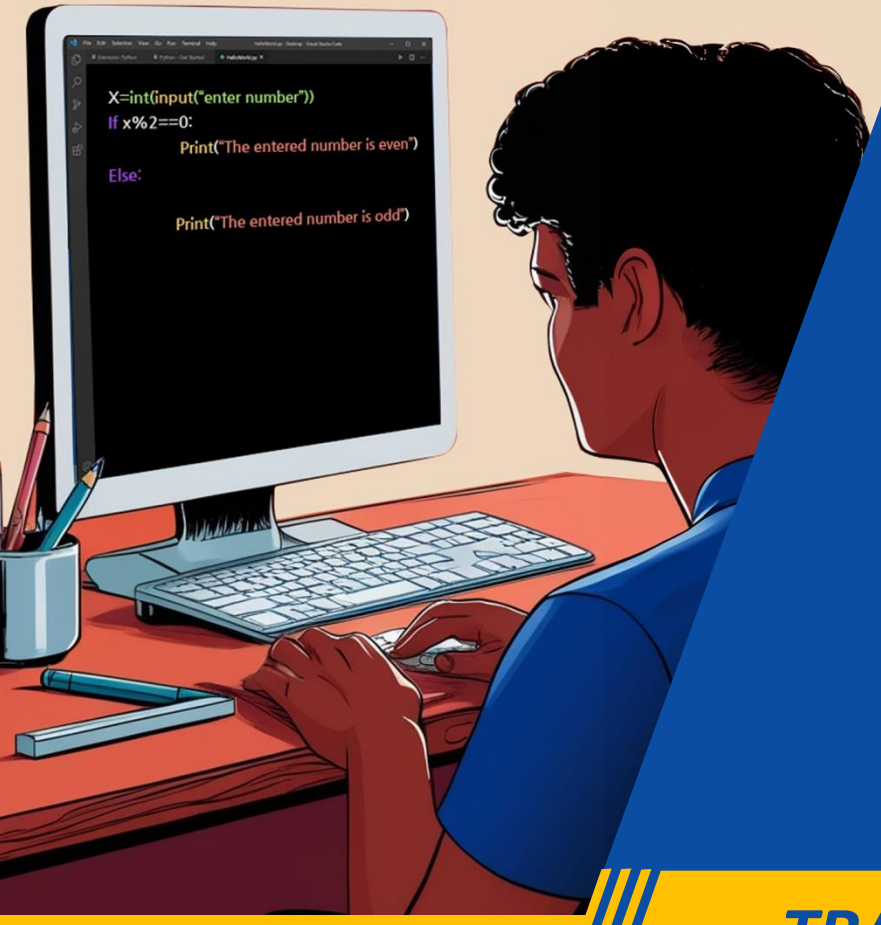
RQF LEVEL 5



GENPP501

SOFTWARE DEVELOPMENT

Python Programming Fundamentals

TRAINEE'S MANUAL

October, 2024

PYTHON PROGRAMMING FUNDAMENTALS

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer and trainee's manuals for the TVET Certificate V in Software Development, specifically for the module "**GENPP501: Python Programming Fundamentals"**

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable
MARIA Bernadette M. Ramos
MUTIJIMA Asher Emmanuel

PRODUCTION TEAM

Authoring and Review

SEKABANZA Jean de la Paix

AKIMANA Gabriel

Validation

HAKIZIMANA Evariste
NYANDWI Rongin

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

NGENDAHAYO HENRY Gabriel

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe
SUA Lim
SAEM Lee
SOYEON Kim
WONYEONG Jeong
HAKIZAYEZU Adrien

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

API: Application Programming Interface
AWS: Amazon Web Services
CHMOD: Change Mode
CI/CD: Continuous Integration/Continuous Deployment
CSV: Comma-Separated Values
DB: Database
GUI: Graphical User Interface
I/O: Input/Output
IANA: Internet Assigned Numbers Authority
ID: Identification
IDE: Integrated Development Environment
IP: Internet Protocol
KOICA: Korea International Cooperation Agency
OOP: Object-Oriented Programming
OS: Operating System
PIP: Package Installer for Python
RAM: Random Access Memory
RTB: Rwanda TVET Board
SDK: Software Development Kit
SQL: Structured Query Language
SSH: Secure Shell
TQUM Project: TVET Quality Management Project
UTC: Coordinated Universal Time
VM: Virtual Machine
YAML: YAML Ain't Markup Language

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in Software Development specifically for the module of **"Python Programming Fundamentals".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: GENPP501 PYTHON PROGRAMMING FUNDAMENTALS

Learning Outcome 1: Prepare python environment
Learning Outcome 2: Write basic python program
Learning Outcome 3: Apply object-driven in python


cmd-python
Microsoft Windown [version 10.0.22621.3593]
(c) Microsoft Corporation. All rights reserved.
C:\Users\samsung>pip install nltk
Collecting nltk
Downloading nltk-3.8.1-py3-none-any.whl (1.5 MB)
Collecting click
Downloading click-3.8.1-py3-none-any.whl (97 kB)
Collecting joblib
Downloading joblib-1.4.2-py3-none-any.whl (301 kB)
Collecting nltk
Downloading nltk-3.8.1-py3-none-any.whl (97 kB)
Collecting regex>=2021.8.3
Downloading regex-2024.10.22-cp311-win_amd64.whl(268 kB)
Collecting colorama
Downloading colorama-0.4.6-py3-none-any.whl (25 kB)
Installing collected packages: regex, joblib, colorama, click, nltk
Successfully installed click-8.1.7 colorama-0.4.6 joblib-1.4.2 nltk-3.8.1 regex-2024.10.22
[notice] A new release of pip available : 22.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip

Indicative contents
1.1 Selection of Python tools
1.2 Installation of Python tools
1.3 Testing python installation

Key Competencies for Learning Outcome 1: Prepare Python Environment

Knowledge	Skills	Attitudes
• Description of python programming. • Identifications of python tools. • Identification of computer system requirements • Description of application of python	• Installing python software tools • Configuring python virtual environment • Running python version command • Checking python interpreter • Testing package manager	• Having teamwork spirit ability • Being critical thinker • Being innovative • Being attentive • Being creative • Problem solving • Being practical oriented

Duration: 15 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly python programming as applied in software development.
2. Identify properly python tools as used in Python programming.
3. Identify properly computer system requirements in line with operating system.
4. Select correctly python tools depending on project to be developed.
5. Install correctly python software tools based on output of python version command.
6. Configure correctly python virtual environment based on operating system.
7. Test correctly python installation based on output of python version command.

Resources

Equipment	Tools	Materials
● Computer	● IDE (PyCharm) ● Python interpreter (Latest version)	● Internet ● Electricity

Indicative content 1.1: Selection of Python Tools

Duration:5 hrs

Theoretical Activity 1.1.1: Descriptionof python programming

Tasks:

1. You are requested to answer the following questions:
    i. Define python
    ii. Discuss on benefits of python
    iii. Explain characteristics of python
    iv. Python programming is applied in different areas, describe each of the following areas:
        a) Data science
        b) Software development
        c) Automation
        d) Data analysis
2. Write your findings on papers
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 1.1.1 and ask questions where necessary.

Key readings 1.1.1. Description of python programming

1. Definition

Python is a high-level, interpreted programming language known for its simplicity and readability. Created by Guido van Rossum and first released in 1991, Python has gained immense popularity due to its versatility and wide range of applications.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- System scripting.

2. Uses of Python

Python is used in the following ways:

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.

- Python can be used for rapid prototyping, or for production-ready software development.

3. Comparison between Python Syntax and other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

4. Benefits of python

Python offers a range of advantages that make it a popular choice for developers, data scientists, and researchers.

4.1. Simplicity and Readability

Python's clear syntax and structure promote easy reading and writing of code, which is especially beneficial for beginners.

4.2. Versatility

Python is suitable for various applications, including web development, data analysis, artificial intelligence, scientific computing, automation, and more.

4.3. Extensive Libraries and Frameworks

Python boasts a rich ecosystem of libraries and frameworks (e.g., NumPy, pandas, Django, Flask) that simplify complex tasks and speed up development.

4.4. Strong Community Support

A large and active community means extensive support, documentation, and resources, making it easier for learners to find help and tutorials.

4.5. Cross-Platform Compatibility

Python can run on various operating systems (Windows, macOS, Linux), allowing for flexibility in development and deployment.

4.6. Dynamic Typing

Variables in Python do not require an explicit declaration of data types, which can speed up coding and reduce boilerplate.

4.7. Interpreted Language

Python code is executed line by line, which simplifies debugging and makes it easier to test code snippets.

4.8. Integration Capabilities

Python can easily integrate with other languages (like C, C++, and Java) and technologies, making it a good choice for projects that require mixed-language support.

4.9. Object-Oriented and Functional Programming

Python supports both programming paradigms, allowing developers to choose

the most suitable approach for their projects.

4.10. Rapid Development

The combination of an easy learning curve, extensive libraries, and rapid prototyping capabilities allows for quicker development cycles.

5. Characteristics of python

Python is a high-level, general-purpose programming language known for its simplicity, readability, and versatility. Here are some of its key characteristics:

5.1. Readability

Clear syntax: Python uses indentation to define code blocks, making it easy to read and understand.

English-like keywords: Python uses words like "if", "else", "for", and "while" to control program flow, making it more intuitive for beginners.

5.2. Versatility

Wide range of applications: Python can be used for web development, data analysis, scientific computing, machine learning, automation, and more.

Cross-platform compatibility: Python code can run on various operating systems like Windows, macOS, and Linux.

5.3. Efficiency

- **Interpreted language:** Python code is executed line by line, making it slower than compiled languages like C++. However, it's often faster than other interpreted languages like Ruby.
- **Large standard library**: Python comes with a rich standard library that provides modules for various tasks, reducing development time.

5.4. Community and Ecosystem

- **Active community**: Python has a large and supportive community of developers, which means you can find plenty of resources, tutorials, and help online.
- **Extensive ecosystem**: Python has a vast ecosystem of third-party libraries and frameworks, such as NumPy, Pandas, TensorFlow, and Django, that extend its capabilities.

5.5. Other characteristics

- **Dynamic typing**: Python automatically determines the data type of variables at runtime.
- **Object-oriented programming**: Python supports object-oriented programming paradigms, allowing you to create modular and reusable code.
- **Memory management**: Python handles memory management automatically, freeing developers from the burden of manual memory allocation and deallocation.

6. Application of python

The applications of Python programming are diverse and span various fields.

6.1. Data Science

- **Data analysis:** Python's libraries like NumPy, Pandas, and Matplotlib facilitate data cleaning, exploration, and visualization.
- **Machine learning:** Frameworks like TensorFlow and PyTorch enable building and training complex machine learning models.
- **Deep learning:** Python is widely used for developing deep neural networks for tasks like image recognition and natural language processing.

6.2. Software Development

- **Web development:** Frameworks like Django and Flask streamline the creation of web applications.
- **Desktop applications:** Libraries like Tkinter, PyQt, and wxPython allow for building graphical user interfaces.
- **Game development:** Pygame is a popular framework for creating simple games.

6.3. Automation:

- **Task automation:** Python can automate repetitive tasks, such as sending emails, downloading files, or interacting with web applications.
- **System administration:** Python scripts can be used to automate system management tasks, like backing up data or monitoring system performance.

6.4. Data Analysis:

- **Statistical analysis:** Python's libraries like SciPy and Stats models provide tools for statistical analysis and modelling.
- **Data mining:** Python can be used for discovering patterns and trends in large datasets.
- **Data visualization**: Matplotlib and Seaborn allow for creating informative and visually appealing charts and graphs.

Practical Activity 1.1.2: Selecting python tools

Task:

1. Read the task bellow

   Ubumwe ltd need to develop a web app that help them in selling their products online and the system will have capabilities to perform automatic updates while products are sold and automatic deployment on the side of system administrator. You are hired as full stack developer responsible for selecting the best tools that will be used.

2. Referring to the provided key reading 1.1.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 1.1.2 Selection of python tools

1. Criteria for selecting tools

Selecting the right Python tools for your project can significantly impact your productivity and the quality of your work.

1.1. Purpose and Requirements

Define the primary goal of your project. Are you building a web application, performing data analysis, or developing a machine learning model?

1.2. Ease of Use

Consider how user-friendly the tool is. Does it have a vertical learning curve, or can you get started quickly? Tools with good documentation and community support can be very helpful.

1.3. Integration

Check if the tool integrates well with your existing systems and workflows. For example, if you're using a specific database or framework, ensure the tool supports it.

1.4. Scalability

Evaluate whether the tool can handle the scale of your project. If you're working with large datasets or high-traffic applications, you'll need tools that can scale accordingly.

1.5. Performance

Assess the performance of the tool. Some tools may be faster or more efficient than others, which can be crucial for performance-critical applications.

1.6. Community and Support

A strong community and good support can make a big difference. Look for tools with active communities, frequent updates, and responsive support.

1.7. Security

Ensure the tool has robust security features, especially if you're dealing with sensitive data.

1.8. Cost

Consider the cost of the tool. While some tools are free and open-source, others may require a subscription or one-time purchase.

1.9. Flexibility

The tool should be flexible enough to adapt to your project's evolving needs. It should allow for customization and extension.

1.10. Compatibility

Ensure the tool is compatible with your operating system and other tools you plan to

use.

N.B: Among the tools you have to select the followings depending on the project to be developed.

2. Python IDE

An IDE is a software application that provides comprehensive facilities for programmers to develop software.

Here are some of the most popular Python IDEs:

2.1. Visual Studio Code (VS Code)

Platform

Windows, macOS, Linux

Features

Lightweight, highly customizable, supports various programming languages, intelligent code completion, debugging, Git integration, and a vast extension marketplace.

2.2. PyCharm

Platform: Windows, macOS, Linux

Features: Specifically designed for Python development, intelligent code completion, refactoring, debugging, unit testing, and integration with popular Python frameworks like Django and Flask.

2.3. Jupiter Notebook

Platform: Web-based, can be used in most browsers

Features: Interactive environment for data analysis, scientific computing, and machine learning. It's great for exploring data, creating visualizations, and sharing results.

2.4. Spyder

Platform: Windows, macOS, Linux

Features: Similar to MATLAB, Spyder is a scientific computing environment with a MATLAB-like interface, suitable for data analysis and scientific programming.

2.5. Sublime Text

Platform: Windows, macOS, Linux

Features: A lightweight, highly customizable text editor with excellent Python support, including syntax highlighting, code completion, and multiple panes.

2.6. Thonny

Platform: Windows, macOS, Linux

Features: A beginner-friendly IDE designed for teaching programming, with a simple interface, step-by-step debugging, and a built-in Python interpreter.

3. Python frameworks

Python frameworks provide a pre-built structure and tools to streamline the development process. They offer reusable code, best practices, and a modular approach, making it easier to build applications efficiently.

3.1. Web Development Frameworks

3.1.1. Django:

A full-stack framework known for its rapid development, scalability, and security features. It's ideal for complex web applications.

3.1.2. Flask

A lightweight, minimalist framework that offers flexibility and control. It's suitable for smaller projects and APIs.

3.1.3. FastAPI

A modern framework that emphasizes performance, ease of use, and developer experience. It's built on top of Starlette and Pedantic.

3.2. Data Science and Machine Learning Frameworks

3.2.1. TensorFlow

A popular open-source platform for machine learning, deep learning, and natural language processing. It's used for building and training neural networks.

3.2.2. PyTorch

Another popular framework for machine learning, known for its flexibility and ease of use. It's often used for research and prototyping.

3.2.3. Scikit-learn

A machine learning library that provides a simple interface for building and training models, including classification, regression, clustering, and dimensionality reduction.

3.3. Scientific Computing and Data Analysis Frameworks

3.3.1. NumPy

A fundamental package for numerical computing, providing multi-dimensional arrays and matrices, along with mathematical functions.

3.3.2. Pandas

A powerful data analysis library offering data structures like Data Frames and Series, which make it easy to manipulate and analyse data.

3.3.3. Matplotlib

A plotting library for creating static, animated, and interactive visualizations.

Other Notable Frameworks:

3.3.4. Kivy

A framework for building cross-platform mobile apps, desktop applications, and web applications using a single codebase.

3.3.5. Twisted

A framework for asynchronous networking, making it suitable for building network-intensive applications like servers and chat clients.

3.3.6. CherryPy

A minimalistic web framework that emphasizes simplicity and performance.

3.4. Automation libraries in python

Python has several powerful libraries for automation that can help streamline tasks across various domains.

Here are some of the most popular ones:

3.4.1. Selenium

Use Case: Automating web browsers.

Description: Selenium allows you to control a web browser programmatically. It can be used for testing web applications and scraping data from websites.

3.4.2. PyAutoGUI

Use Case: GUI automation.

Description: PyAutoGUI lets you simulate mouse movements, clicks, and keyboard inputs to automate interactions with desktop applications.

3.4.3. requests

Use Case: HTTP requests.

Description: While primarily used for making HTTP requests, it can be leveraged for automating interactions with web APIs.

3.4.4. BeautifulSoup

Use Case: Web scraping.

Description: BeautifulSoup is used for parsing HTML and XML documents. It helps in extracting data from web pages.

3.4.5. Pandas

Use Case: Data manipulation and analysis.

Description: Pandas is excellent for automating data processing tasks, such as cleaning, transforming, and analyzing data in tabular formats.

3.4.6. Airflow

Use Case: Workflow automation.

Description: Apache Airflow is a platform to programmatically author, schedule, and monitor workflows, making it ideal for batch data processing.

3.4.7. Celery

Use Case: Distributed task queue.

Description: Celery is used for handling asynchronous tasks and scheduling them, making it great for background job processing.

3.4.8. Paramiko

Use Case: SSH and SFTP.

Description: Paramiko allows you to automate SSH connections and file transfers, useful for server management and automation.

3.4.9. Fabric

Use Case: SSH command execution.

Description: Fabric is a high-level Python library for executing shell commands remotely over SSH, making it easier to deploy applications.

3.4.10. pywinauto

Use Case: Windows GUI automation.

Description: This library allows you to automate GUI interactions on Windows applications.

3.4.11. Schedule

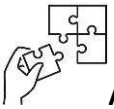
Use Case: Job scheduling.

Description: A simple library for scheduling Python functions to run at specific intervals.

Points to Remember

- Python is a versatile programming language known for its simplicity and readability.
- Python has several benefits including: Simplicity and Readability, Versatility, Extensive Libraries and Frameworks, Strong Community Support, Cross-Platform, Compatibility, Dynamic Typing, Interpreted Language, Integration Capabilities, Object-Oriented and Functional Programming, Rapid Development.
- Key characteristics of python include: Readability, Versatility, Efficiency and Community and Ecosystem.
- Python can be used in data science and machine learning by using the following frameworks: TensorFlow, PyTorch and Scikit-learn.
- Scientific Computing and Data Analysis Frameworks: NumPy, Pandas, Matplotlib.
- In python we can use the following automation Libraries: Selenium, PyAutoGUI, Requests, Beautiful Soup, Airflow, Celery, Paramiko, Fabric, Pywinauto, And Schedule.
- Selecting the right Python tools for your project is crucial for productivity and quality. Start by defining your project's primary goal, whether it's web development, data analysis, or machine learning.
- Consider the tool's ease of use, integration with existing systems, scalability, and performance. A strong community and support, robust security features, and cost are also important factors.
- Ensure the tool is flexible, customizable, and compatible with your operating system and other tools you plan to use.
- Python support the following IDEs: VS Code, PyCharm, Jupyter Notebook, Spyder, Sublime Text and Thonny
- Python support the following web frameworks: Django, Flask and FastAPI
- Python can be used in data science and machine learning by using the following frameworks: TensorFlow, PyTorch and Scikit-learn

- Scientific Computing and Data Analysis Frameworks: NumPy, Pandas and Matplotlib
- In python we can use the following automation Libraries: Selenium, PyAutoGUI, requests, BeautifulSoup, Airflow, Celery, Paramiko, Fabric, Pywinauto and Schedule.



Application of learning 1.1.

HH ltd want to develop a system that will be used while selling their products online and performing some automations once new features have added to their website and performing automatic deployment. You are hired as full stack responsible for selecting the right tools that will be used to develop that software.

Indicative content 1.2: Installation of Python Tools

Duration: 5 hrs

Theoretical Activity 1.2.1: Identification of computer system requirements

Tasks:

1. You are requested to identify the system requirements to install python tools.
2. Write your findings on paper/flipchart
3. Present your findings to the whole class and trainer
4. For more clarification read key reading 1.2.1 and ask questions where necessary.

Key readings 1.2.1.: Identification of Computer System Requirements to Install Python Tools

The following are requirements to install Python tools in Computer System:

1. Hardware Requirements

Processor: A modern processor (Intel or AMD) with at least dual-core is recommended for running Python tools efficiently.

For data science, machine learning, or automation tasks, a multi-core processor or GPU (for TensorFlow or PyTorch) can significantly enhance performance.

Memory (RAM): A minimum of 4 GB of RAM is recommended for general Python development.

For more resource-intensive tasks like data analytics, machine learning, or web development, 8 GB to 16 GB of RAM is ideal.

Storage: At least 1 GB of free disk space is needed to install Python and related tools.

For larger projects or when working with large datasets, SSD storage is preferred for faster read/write speeds.

Graphics Card: If working with AI/ML libraries (e.g., TensorFlow, PyTorch), a dedicated NVIDIA GPU with CUDA support is beneficial for model training.

2. **Software Requirements**

Operating System: Python tools can run on Windows, macOS, or Linux. However, certain tools may have better support on Linux and macOS (e.g., TensorFlow on GPUs). Ensure that the OS version is up-to-date and compatible with Python versions (e.g., Windows 10 or higher).

Python Interpreter: The latest version of Python (3.x) should be installed. Some tools may require specific versions of Python (3.6, 3.8, etc.).

Package Manager: pip (Python's package installer) is required for installing most Python tools. Ensure that pip is installed and up-to-date.

Development Tools: A suitable IDE or text editor (such as PyCharm, VS Code, or Jupyter Notebook) is required for writing and testing Python code.

If working with web development or DevOps, Node.js, Docker, or Git may be necessary for additional setup.

Practical Activity 1.2.2: Installing python software tools

Task:

1. Read the task bellow
   As a full stack developer, you are asked to go to the computer lab to install python and PyCharm in a computer.
2. Referring to the provided key reading 1.2.2, perform the task described above.
3. Present your work to the trainer and whole class.

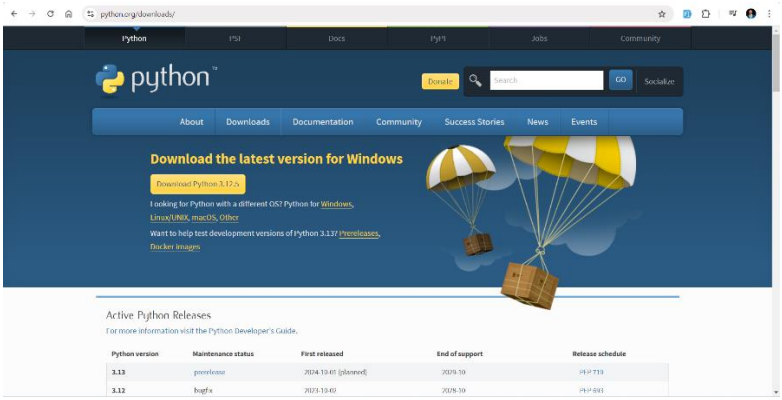
Key readings 1.2.2 Installation of python software tools

Install python software tools

1. Install python

a. Download the Python Installer or use offline from external storage

- ✓ Go to the official Python website: https://www.python.org/downloads/
- ✓ Select the latest stable Python version (e.g., Python 3.12).
- ✓ Click on the appropriate installer for your Windows system (32-bit or 64-bit).

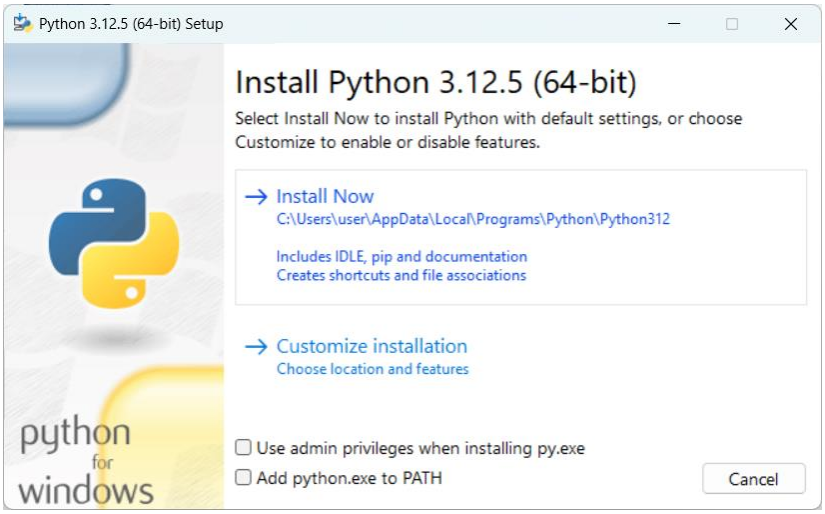


b. RUN the Installer

✓ Double-click the installer file (e.g., python-3.12.exe).

✓ Follow the on-screen instructions: Check the "Add Python to PATH" option to make Python accessible from the command line.

✓ Choose the installation location (default is usually fine).



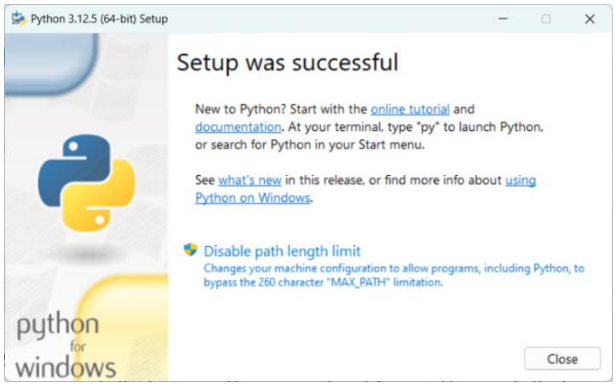
✓ Click "Install Now" or "Customize installation" for more advanced options.

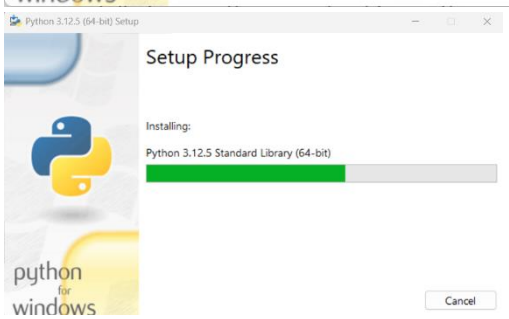
✓ Wait for the installation to complete.

C. Verify the Installation

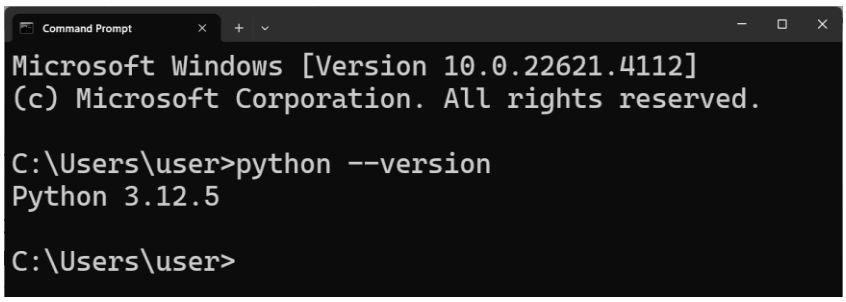
✓ Open a command prompt or terminal window.

✓ Type python --version and press Enter.





✓ If Python is installed correctly, you should see the installed version number displayed.

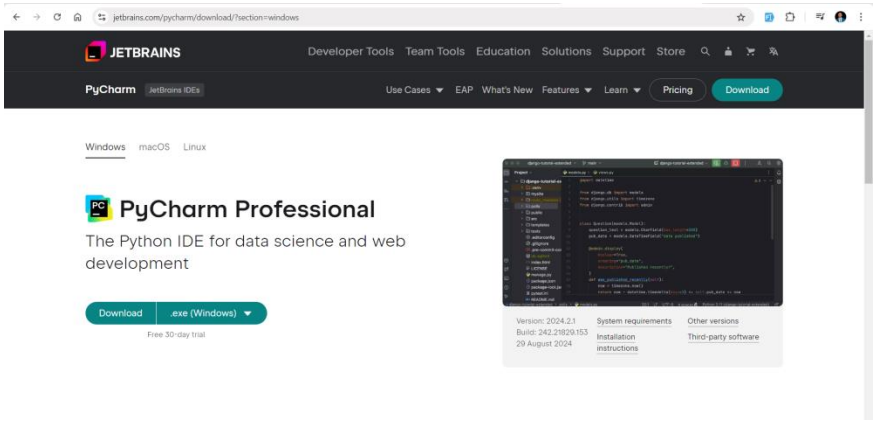


2. Install PyCharm

a. **Download PyCharm or use offline from external storage**

✓ Visit the Jet Brains website: https://www.jetbrains.com/pycharm/download/

✓ Click on the "Download PyCharm" button.

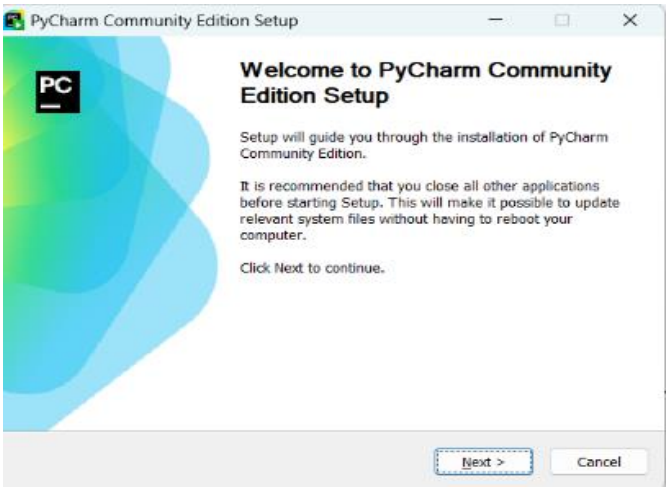


✓ Choose the appropriate edition for your needs (Community or Professional).

✓ Select the Windows installer.

b. **Run the Installer**

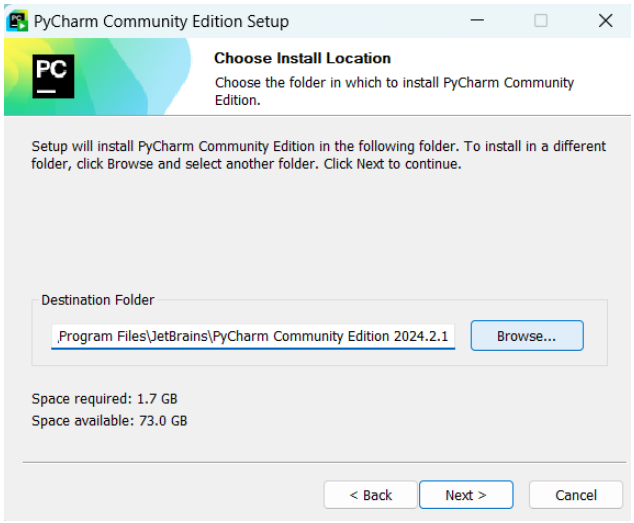
✓ Double-click the downloaded installer file (e.g., pycharm-community-2023.3.exe).



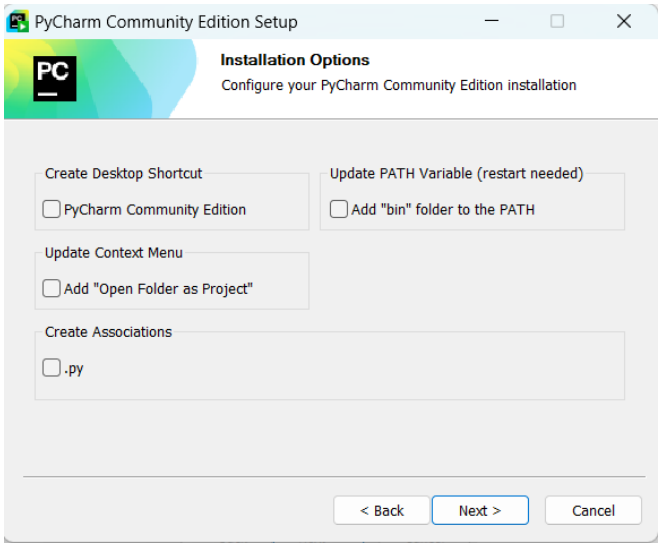
Follow the on-screen instructions

✓ Accept the license agreement.

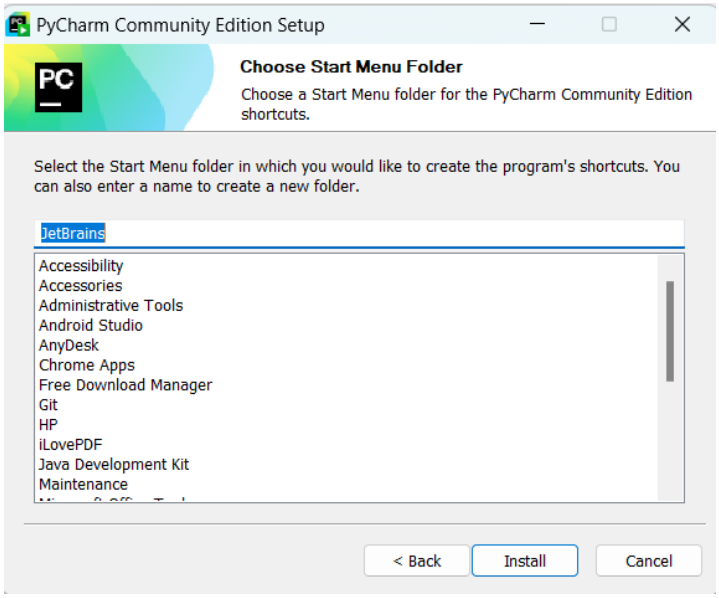
✓ Choose the installation location (default is usually fine).

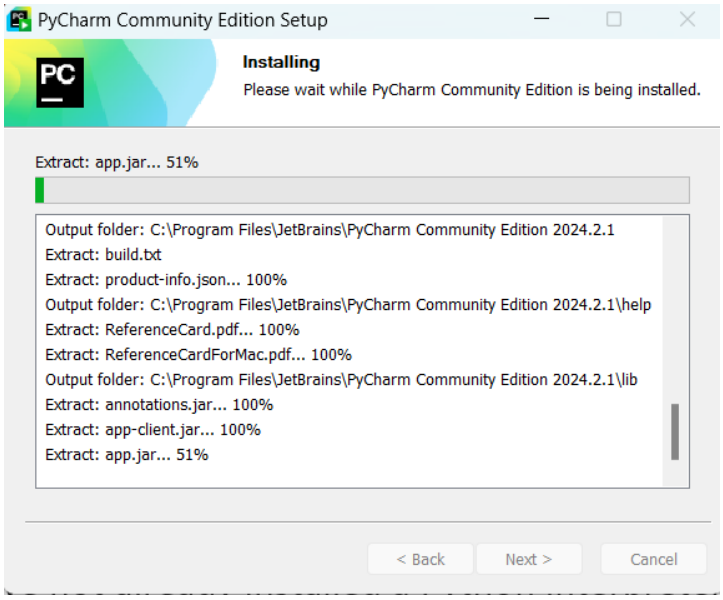


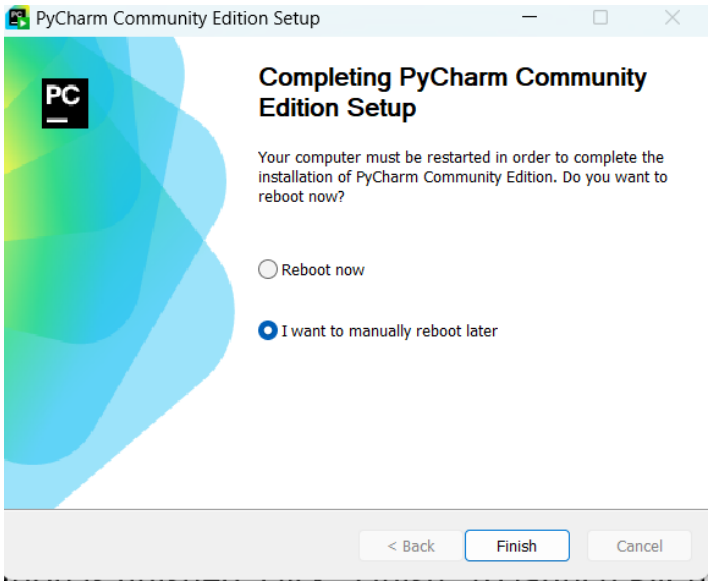
✓ Select the desired components to install (e.g., Python interpreter, web development tools).



✓ Click "Install" and wait for the installation to complete.

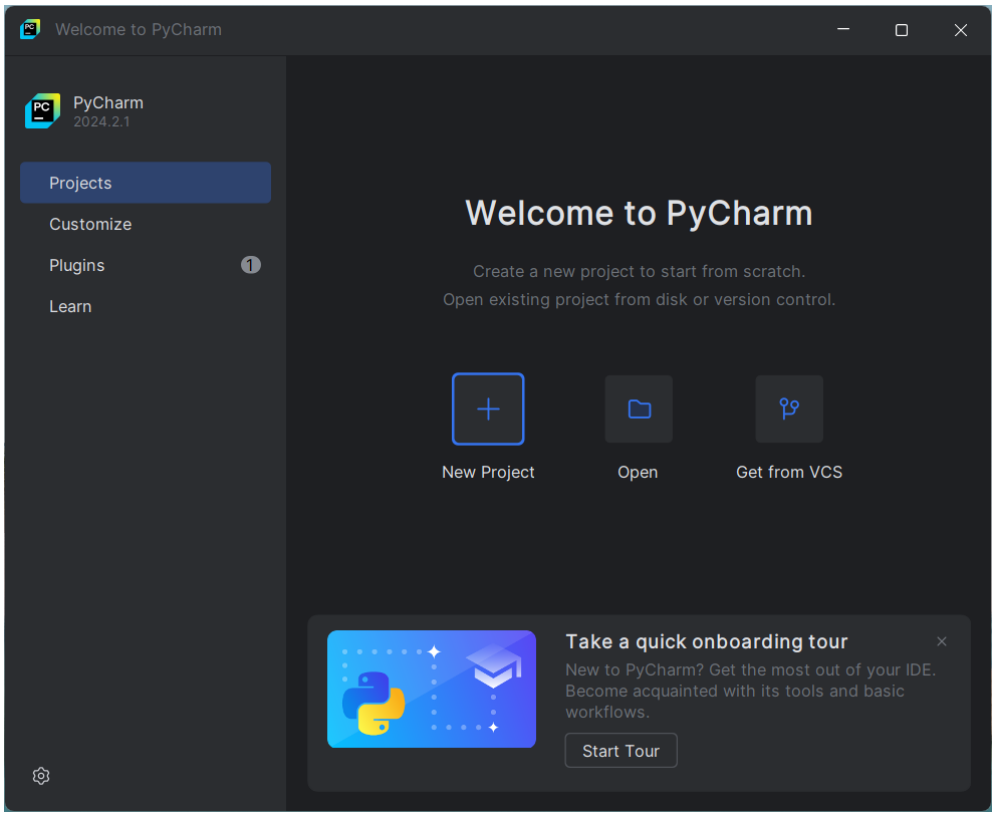






c. **Launch PyCharm**

Once the installation is finished, click "Finish" to launch PyCharm.



Practical Activity 1.2.3: Configuring python virtual environment

Task:

1. Read key reading 1.2.3. **Configuration of python virtual environment**
2. Referring to the Key reading 1.2.3 you are requested to go to the computer lab to configure python virtual environment.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, configure python virtual environment.
5. Present out the steps to configure python virtual environment.

Key readings 1.2.3 Configuration of python virtual environment

1. Introduction

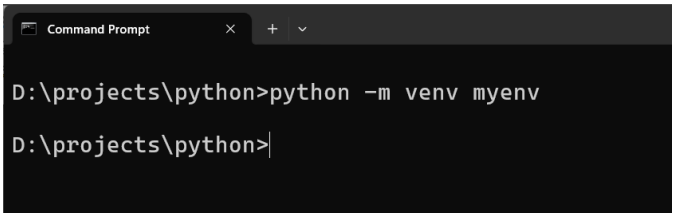
A Python virtual environment is a tool that helps you create an isolated environment for your Python projects. This means you can manage dependencies for each project separately, avoiding conflicts between different projects' requirements.

2. Creating a Virtual Environment

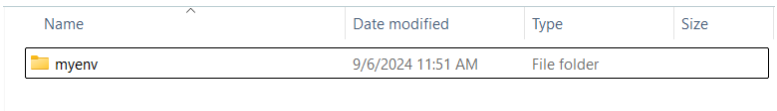
✓ Open your terminal or command prompt.

✓ Navigate to your project directory.

✓ Run the following command "**python -m venv myenv"**



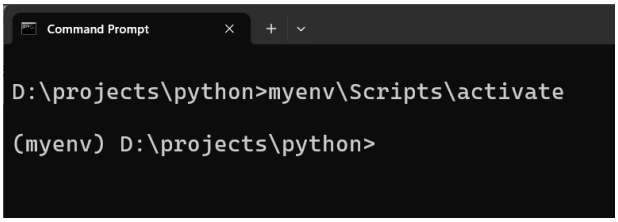
✓ This will create a directory named myenv containing a standalone Python installation.



3. Activating the Virtual Environment

✓ **On Windows**

Run the following command **myenv\Scripts\activate**



✓ **On macOS and Linux**

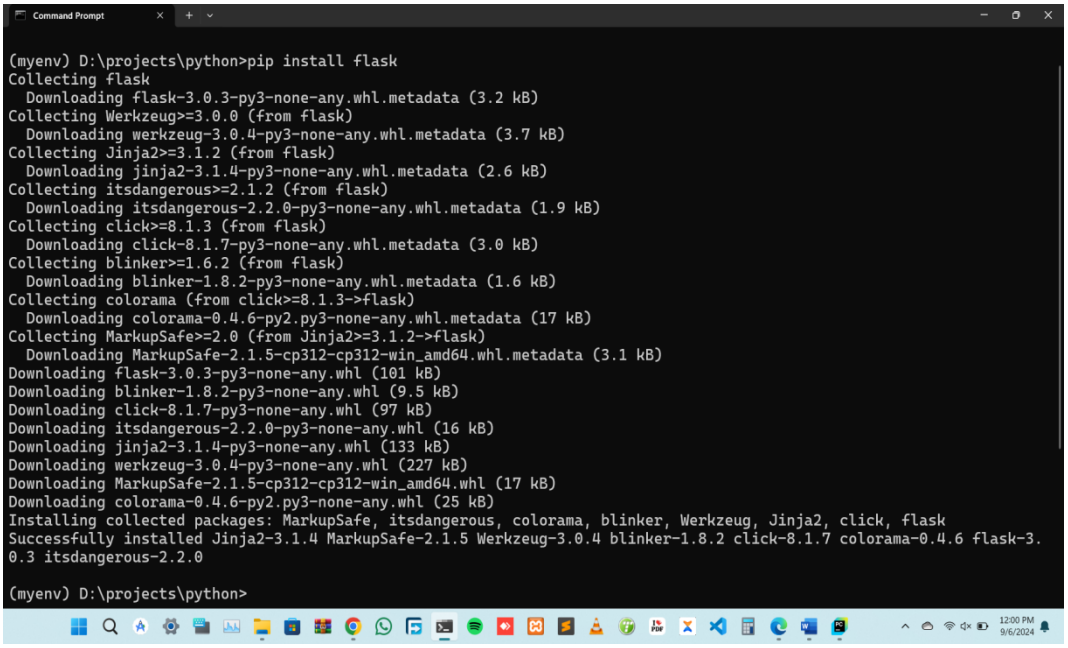
Run the following command **source myenv/bin/activate**

Once activated, your terminal prompt will change to indicate that you are now working within the virtual environment.

3. Installing Packages

With the virtual environment activated, you can install packages using pip

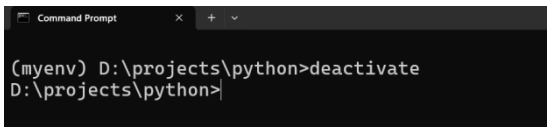
"pip install package name"



4. Deactivating the Virtual Environment

To exit the virtual environment, simply run

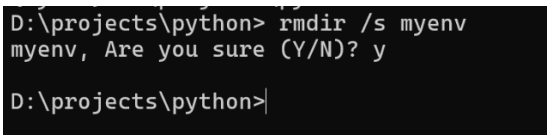
"deactivate"



5. Deleting the Virtual Environment

If you no longer need the virtual environment, you can delete the myenv directory

rmdir /s myenv



6. Freeze packages

The "freeze" command in the context of Python virtual environments is used to create a list of all installed packages and their versions. This is particularly useful for replicating your environment or sharing your project setup with others.

✓ Activate your virtual environment if it's not already activated.

✓ Run the freeze command

```
(myenv) D:\projects\python>pip freeze > requirements.txt

(myenv) D:\projects\python>
```

This command will create a file named requirements.txt in your current directory, containing a list of all installed packages and their versions.

7. **To install packages from a requirements.txt file in another environment**

   This is helpful for setting up identical environments across different machines or for other developers working on the same project

```
(myenv) D:\projects\python>pip install -r requirements.txt
```

Points to Remember

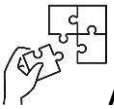
- To install Python tools, your system should meet certain hardware requirements: at least a dual-core processor, 4 GB RAM (preferably 8-16 GB for heavier tasks), 1 GB of free disk space, and optionally, a GPU for machine learning.
- On the software side, ensure you have the latest version of Python 3.x, an appropriate IDE (e.g., VS Code, PyCharm), pip for package management, and necessary dependencies like C++ build tools or Java for specific libraries
- While installing python we follow these steps:

  Step 1: download the Python Installer or use offline from external storage
  Step 2: run the installer
  Step 3: verify the installation

- while installing PyCharm we follow these steps:

  Step 1: Download PyCharm or use offline from external storage
  Step 2: Run the installer
  Step 3: Launch PyCharm

- Python virtual environments create isolated environments for projects, preventing dependency conflicts.
- To create a virtual environment, use python -m venv myenv.
- Activate it using myenv\Scripts\activate.
- Install packages with pip install package_name.
- Deactivate using deactivate.
- Freeze dependencies with pip freeze > requirements.txt.
- Install from requirements.txt using pip install -r requirements.txt.



Application of learning 1.2.

HHT LTD Company located in Kicukiro district, wants to develop a web app that will help in selling their products online and the system will have capabilities to perform automatic deployment on the side of system administrator. You have been hired as full stack developer responsible for installing and configuring all python tools that will be needed to develop that project.

The company will provide all tools, materials and equipment.

Indicative content 1.3: Testing Python Installation

Duration: 5 hrs

Practical Activity 1.3.1: Testing python installation

Task:

1. Read key reading 1.3.1.
2: As full-stack, you are asked to go to the computer lab to Run python version command, Check python interpreter and test package manager.
3. Apply safety precautions
4. Referring to the steps provided in key readings, Run python version command, Check python interpreter and Test package manager.
5. Present out the steps to Run python version command, Check python interpreter and Test package manager.

Key readings 1.3.1 Testing python installation

To test your Python installation, follow these steps:

Step 1: Check Python Version

- ✓ Open a Terminal or Command Prompt:
- ✓ Windows: Press Win + R, type cmd, and press Enter.
- ✓ macOS: Open Finder, go to Applications, then Utilities, and double-click on Terminal.
- ✓ Linux: Open your terminal application.
- ✓ Run the Python Version Command **"python –version"**
- ✓ or, for systems where Python 3 is installed as python3 **"python3 –version"**

Expected Output

```
C:\Users\jeand>python --version
Python 3.7.4

C:\Users\jeand>
```

Step 2: Check Python Interpreter

Open Python Interpreter

In the terminal, type "**python**" or **"python3"**

Verify the Prompt

```
C:\Users\jeand>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Exit the Interpreter

To exit, type **"exit()"** or press Ctrl + D (macOS/Linux) or Ctrl + Z followed by Enter (Windows).

```
>>> exit()

C:\Users\jeand>
```

Step 3: Test the Package Manager (pip)

Check pip Version

In the terminal, type **"pip –version"** or, for systems where pip is installed as pip3 **"pip3 –version"**

Expected Output

```
C:\Users\jeand>pip --version
pip 19.0.3 from c:\users\jeand\appdata\local\programs\python\python37\lib\site-packages\pip (python 3.7)
```

Test Installing a Package

As a test, you can try installing a simple package, such as Django, openpyxl, pytz:

pip install Django // for installing django

pip install openpyxl // for installing openpyxl

pip install pytz // for installing the time zone

Expected Output

You should see messages indicating that the package is being downloaded and installed.

```
Installing collected packages: et-xmlfile, openpyxl
Successfully installed et-xmlfile-1.1.0 openpyxl-3.1.3
You are using pip version 19.0.3, however version 24.0 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\jeand>pip list
```

Verify the Package Installation

After installation, you can verify it by running **"pip list"**

This will display information about the packages that are installed, confirming it is installed.

```
C:\Users\jeand>pip list
Package           Version
----------------- -------
asgiref           3.7.2
Django            3.2.25
et-xmlfile        1.1.0
openpyxl          3.1.3
pip               19.0.3
pytz              2024.1
setuptools        40.8.0
sqlparse          0.4.4
typing-extensions 4.7.1
You are using pip version 19.0.3, however version 24.0 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\jeand>
```

Points to Remember

- To Check Python Version run that command in terminal: **python --version or python3 --version**
- To Check Python Interpreter run that command in terminal: **python or python3**
- To Test Package Manager: Check pip version: **pip –version** and Install a package: **pip install Django**
- To Verify installation run: **pip list**

Application of learning 1.3.

HHT LTD is software development company located in Kicukiro district, that company develop different software of institutions, they want to develop a web app that will help STY ltd in selling their products online and the system will have capabilities to perform automatic deployment on the side of system admistrator. You have been hired as full stack developer responsible for testing the python installation and installing required packages that will be used.

The company will provide all tools, materials and equipment.

Learning outcome 1 end assessment

Written assessment

I. Circle the letter that corresponding to the right anwer

1. **Which of the following is a feature of Python?**
   a) Case sensitivity
   b) Curly-bracket syntax
   c) Indentation-based syntax
   d) Use of semicolons
2. **Python was created by:**
   a) Guido van Rossum
   b) James Gosling
   c) Bjarne Stroustrup
   d) Dennis Ritchie
3. **Which Python framework is known for rapid development and scalability?**
   a) Flask
   b) Django
   c) FastAPI
   d) CherryPy
4. **Python can be used for:**
   a) Web development
   b) Data analysis
   c) Machine learning
   d) All of the above
5. **Which IDE is specifically designed for scientific computing?**
   a) PyCharm
   b) Visual Studio Code
   c) Spyder
   d) Sublime Text
6. **Python is classified as a(n):**
   a) Compiled language
   b) Interpreted language
   c) Assembly language
   d) Machine language
7. **What is the default file extension for Python scripts?**
   a) .pyc
   b) .txt
   c) .py
   d) .exe

8. **The command to install a Python package using pip is:**
    a) python install package_name
    b) pip install package_name
    c) install package_name
    d) pip setup package_name
9. **Which of the following is NOT a Python web framework?**
    a) Django
    b) Flask
    c) NumPy
    d) FastAPI
10. **In Python, indentation is used to:**
    a) Declare variables
    b) Define the scope of loops and functions
    c) Import libraries
    d) Comment on the code
11. **Which Python library is primarily used for numerical computing?**
    a) Pandas
    b) NumPy
    c) Matplotlib
    d) Scikit-learn
12. **Python's memory management is handled by:**
    a) The developer
    b) The Python interpreter automatically
    c) An external tool
    d) Manual memory allocation
13. **Which of the following is an example of a Python text editor?**
    a) PyCharm
    b) Jupyter Notebook
    c) Sublime Text
    d) All of the above
14. **Python was first released in:**
    a) 1989
    b) 1991
    c) 1995
    d) 2000
15. **What is the primary use of the TensorFlow library?**
    a) Web development
    b) Machine learning
    c) Data visualization
    d) Game development
16. **Which of the following is NOT a characteristic of Python?**

a) Dynamic typing
b) Complex syntax
c) Object-oriented programming
d) Readability

17. **What does the command python --version do?**
    a) Runs a Python script
    b) Displays the current Python version installed
    c) Updates Python to the latest version
    d) Installs Python on your system
18. **Which Python library is widely used for data visualization?**
    a) Matplotlib
    b) TensorFlow
    c) Flask
    d) Pandas
19. **What does IDE stand for in the context of Python?**
    a) Integrated Development Environment
    b) Interactive Development Editor
    c) Integrated Debugging Environment
    d) Interactive Design Editor
20. **Which Python command is used to create a virtual environment?**
    a) python create venv
    b) python -m venv
    c) venv create
    d) create venv python
21. **Which Python framework is designed for asynchronous networking?**
    a) Kivy
    b) Twisted
    c) CherryPy
    d) Flask
22. **In Python, a function is defined using the keyword:**
    a) func
    b) function
    c) def
    d) define
23. **Which IDE is web-based and primarily used for data analysis?**
    a) PyCharm
    b) Jupyter Notebook
    c) Visual Studio Code
    d) Thonny
24. **The command to deactivate a Python virtual environment is:**
    a) end venv

b) stop venv
c) deactivate
d) exit venv

25. **What type of language is Python?**
    a) Low-level
    b) High-level
    c) Mid-level
    d) Machine-level

II. Complete the following statements by using one of the keyword listed below; You can use one keyword once or more.

Display installed packages, Kivy, Large, Data type, IDEs, Delete, Functional, TensorFlow, Include libraries, Interpreted, pip, Indentation, FastAPI, Pandas, Beginners

1. Python is an _____ language, meaning it executes code line by line.
2. The Python package manager is called _____.
3. Python relies on _____ to define the scope of loops, functions, and classes.
4. _____ is a popular Python framework for building APIs, known for its performance and ease of use.
5. The Python library _____ is used for data manipulation and analysis.
6. Python's simple and readable syntax makes it especially beneficial for _____.
7. The command pip list is used to _____.
8. _____ is a Python framework used for developing cross-platform mobile and desktop applications.
9. Python's _____ community provides extensive support, documentation, and resources.
10. In Python, variables do not require an explicit _____ declaration.
11. PyCharm and VS Code are examples of _____ used for Python development.
12. The command rmdir /s myenv is used to _____ a Python virtual environment.
13. Python supports both object-oriented and _____ programming paradigms.
14. The Python library _____ is widely used for machine learning and deep learning.
15. In Python, the import statement is used to _____.

III. Matching questions

1. **Match the IDE in column A with their corresponding primary features in column B**

Answers	Column A	Column B
……..	a) PyCharm	(i) Beginner friendly interface
……..	b)Jupyter Notebook	(ii) Intelligent code completion
……..	c) Thonny	(iii) Interactive environment for data analysis
……..	d) Visual Studio Code	(iv) Lightweight and customizable

2. **Match the Python framework in column A with its corresponding description in column B**

Answers	Column A	Column B
……..	a) Django	(i) Framework for cross platform applications
……..	b) Flask	(ii) Lightweight framework for small projects
……..	c) FastAPI	(iii) Full stack framework for web development
……..	d) Kivy	(iv) High performance API framework

3. **Match the Python library of column A with its corresponding application in column B**

Answers	Column A	Column B
……..	a) NumPy	(i) Numerical computing
……..	b) Pandas	(ii) Data manipulation
……..	c) Matplotlib	(iii) Data visualization
……..	d) Scikit learn	(iv) Machine learning

4. **Match the Python characteristic in column A with its corresponding feature of column B**

Answers	Column A	Column B
……..	a) Dynamic typing	(i) Clear syntax and structure
……..	b) Readability	(ii) No explicit data type declaration
……..	c) Efficiency	(iii) Large standard library
……..	d) Versatility	(iv) Wide range of applications

5. **Match the Python command in column A with its corresponding function of column B**

Answers	Column A	Column B
……..	a) python --version	(i) Display the Python version
……..	b) pip install	(iii) Install Python packages
……..	c) deactivate	(iv) Exit the virtual environment
……..	d) pip list	(ii) Display installed packages

6. **Match the Python tool of column A with its corresponding description in column B**

Answers	Column A	Column B
……..	a) PyTorch	(i) Computer vision library
……..	b) TensorFlow	(ii) Flexible machine learning framework
……..	c) SciPy	(iii) Platform for deep learning
……..	d) OpenCV	(iv) Scientific computing and statistics

7. **Match the Python task in Column A with corresponding the suitable library/framework in Column B:**

……..	Column A	Column B
……..	a) Web Development	(i) Scikit learn
……..	b) Data Visualization	(ii) Matplotlib
……..	c) Machine Learning	(iii) Django
……..	d) Task Automation	(iv) Python's standard library

8. **Match the Python version command in Column A with its corresponding corresponding output in Column B:**

Answers	Column A	Column B
……..	a) python --version	(i) Displays the Python version number
……..	b) pip show numpy	(ii) Shows details of the installed NumPy package
……..	c) python -m venv env	(iii) Creates a new virtual environment
……..	d) pip freeze	(iv) Lists installed packages in the virtual environment

9. **Match the Python command in Column A with the corresponding action it performs in Column B:**

Answers	Column A	Column B
……..	a) import	(i) Defines a function
……..	b) def	(ii) Imports a module or library
……..	c) print	(iv) Outputs data to the console
……..	d) class	(iii) Defines a new class

10. **Match the following Python versions in Column A with their corresponding key characteristics in Column B:**

Answers	Column A	Column B
……..	a) Python 2.x	(i) Legacy version with different print syntax
……..	b) Python 3.x	(ii) Current version with updated syntax
……..	c) Python 3.6+	(iv) Introduced f string formatting
……..	d) Python 3.8+	(iii) Introduced assignment expressions

Practical assessment

HHT LTD Company located in Kicukiro district, wants to develop a web application that will help in selling their products online and the system will have capabilities to perform automatic deployment on the side of system administrator and automatic updates once new feature is added. You have been hired as full stack developer responsible for installing, configuring all python tools that will be needed to develop that project, testing the python installation and installing required packages that will be used.
The company will provide all tools, materials and equipment.

References

Books

Alpaydin, E. (2020). Introduction to Machine Learning (Adaptive Computation and Machine Learning series). MIT Press.

Bishop, C. Ms. (2006). Pa ern Recognition and Machine Learning. Springer.

Chollet, F. (2017). Deep Learning with Python. Manning Publications.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep Learning (Adaptive Computation and Machine Learning series). MIT Press.

Web Links

Foundation, P. S. (2001-2025). *downloads/*. Retrieved 12 23, 2024, from python.org: https://www.python.org/downloads/

JetBrains. (2024, 02 20). *configuring-python-interpreter.html#view_list*. Retrieved 01 13, 2025, from JetBrains: https://www.jetbrains.com/help/pycharm/configuring-python-interpreter.html#view_list

JetBrains. (2023, 05 03). *creating-virtual-environment.html*. Retrieved 01 13, 2025, from JejBrains.com: https://www.jetbrains.com/help/pycharm/creating-virtual-environment.html

JetBrains. (2023). *download/?section=windows*. Retrieved 01 13, 2025, from JetBrains.com: https://www.jetbrains.com/pycharm/download/?section=windows

JetBrains. (2022). quick-start-guide.html#-lnysrf_12. Retrieved 01 13, 2024, from JetBrains.com: https://www.jetbrains.com/help/pycharm/quick-start-guide.html#search


X=int(input("enter number"))
If x%2==0:
Print("The entered number is even")
Else:
Print("The entered number is odd")

Indicative contents
2.1 Applying python basic concepts
2.2 Applying python control structures
2.3 Applying functions in Python
2.4 Applying Python Collections
2.5 Performing File handling

Key Competencies for Learning Outcome 2: Write basic python program

Knowledge	Skills	Attitudes
• Description of python basic concepts • Description of function in python • Description of python Collections • Description of file Handling libraries	• Applying python basic concepts • Applying conditional Statements • Applying looping Statements • Using Jump Statements • Creating function in python • Applying special purpose functions • Applying Python Collections • Performing operations on collection • Practicing read file • Performing write/create and delete file • Applying python best practices	• Having teamwork spirit ability while coding • Being critical thinker in logic of coding • Being innovative in coding • Being attentive • Being creative in discovering new logics • Problem solving • Being practical oriented •

Duration: 45 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly python basic concepts based on python standards
2. Describe properly function in python based on python standards
3. Describe correctly python Collections based on python standards
4. Describe properly file Handling libraries based on python standards
5. Apply correctly python basic concepts based on python standards
6. Apply correctly conditional Statements based on python standards
7. Apply correctly looping Statements based on python standards
8. Use correctly Jump Statements based on python standards
9. Create properly function in python based on python standards
10. Apply properly special purpose functions based on python standards
11. Apply correctly Python Collections based on python standards
12. Perform correctly operations on collection based on python standards
13. Practice clearly read file based on python standards
14. Perform correctly write/create and delete file based on python standards
15. Apply correctly python best practices based on python standards

Resources

Equipment	Tools	Materials
● Computer	● Python ● Python IDE (Pycharm)	● Internet

Indicative content 2.1: Applying Python Basic Concepts

Duration: 9 hrs

Theoretical Activity 2.1.1: Description of python basic concepts

Tasks:

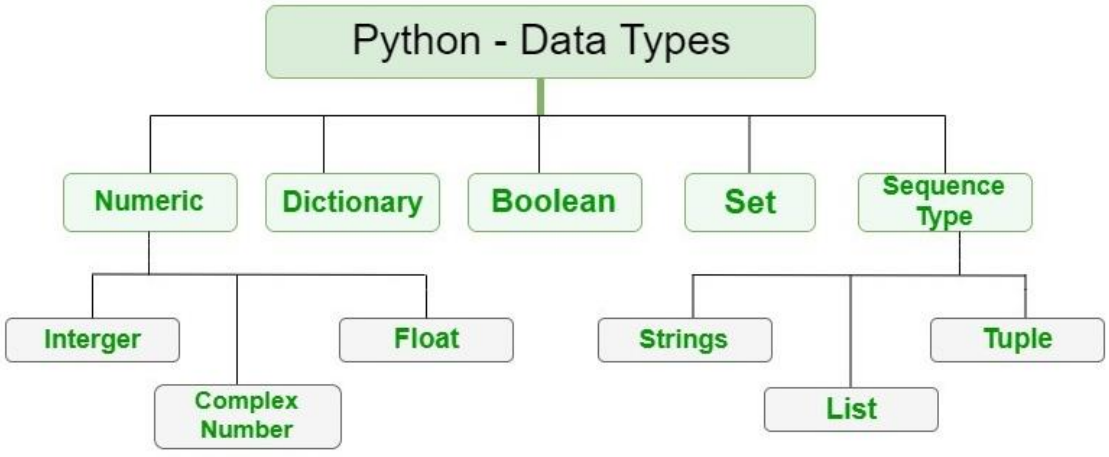
1. You are requested to describe the following python basic concepts:
    i. Data types
    ii. Variable
    iii. Comments
    iv. Operators
2. Write your findings on paper/flipchart
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 2.1.1 and ask questions where necessary.

Key readings 2.1.1. Description of python basic concepts

In python programming, there are most frequently used concepts, some of them are described below:

1. Data Types

In Python, a data type defines the type of a value that a variable can hold. Each data type determines what operations can be performed on that value.



1.1. Built-in data types

Python has several built-in data types, including:

Integers (int): Whole numbers, e.g., 5, -3.

Floating Point Numbers (float): Decimal numbers, e.g., 3.14, -0.001.

Strings (str): Sequences of characters, e.g., "Hello, World!".

Booleans (bool): Represents True or False.

Lists: Ordered, mutable collections, e.g., [1, 2, 3].

Tuples: Ordered, immutable collections, e.g., (1, 2, 3).

Dictionaries (dict): Key-value pairs, e.g., {"name": "Alice", "age": 25}.

Sets: Unordered collections of unique elements, e.g., {1, 2, 3}.

None : Once you have a variable and you didn't assign any value to it

1.2. Features of mapping data types

Key Features of mapping data types are:

- **Unordered**: The elements in a mapping are not stored in a specific order.
- **Mutable**: Elements can be added, modified, or removed after creation.
- **Key-value Pairs**: Each element consists of a key and a value.
- **Efficient Lookup**: Values can be quickly retrieved using their corresponding keys.

Key Characteristics of Boolean datatypes are:

- **Binary**: Boolean values can only be one of two states: true or false.
- **Basic Operations**: Boolean operations include AND, OR, NOT, and XOR.
- **Conditional Statements**: Boolean expressions are used in conditional statements like if, else, and while to control program flow.

Key Features of sets are:

- **Unordered**: The elements in a set are not stored in a specific order.
- **Mutable**: Elements can be added or removed after creation.
- **Unique Elements**: Sets cannot contain duplicate elements.
- **Efficient Membership Testing**: Checking if an element exists in a set is typically very fast.

1.3. Operation that can be done on set datatypes

Operation that can be done on set datatypes includes:

- **Union**: Combines two sets, including all unique elements from both sets.
- **Intersection**: Finds the elements that are common to both sets.
- **Difference**: Finds the elements that are in the first set but not in the second set.
- **Symmetric Difference**: Finds the elements that are in either set but not both.
- **Membership Testing**: Checks if an element exists in the set.
- **Adding/Removing Elements**: Adding or removing elements from a set.

2. Variables

Variables in Python are used to store data. You can create a variable by assigning a value to it using the assignment operator (**=)**.

2.1. Naming rules of python variables.

When naming variables in Python, you must adhere to the following rules:

- Start with a letter or underscore

- Variables cannot begin with a number.
- Consist of letters, numbers, and underscores: Only these characters are allowed in variable names.
- Case-sensitive: Python distinguishes between uppercase and lowercase letters. For example, **myVariable** and **myvariable** are considered different variables.
- Do not use keywords as variable names, as they have special meanings in Python.
- Convention: While not strictly enforced, Python has a common naming convention known as PEP 8. It recommends using lowercase letters with underscores to separate words (e.g., my_variable).

2.2. Variable declaration

Python does not require explicit variable declaration. You can simply assign a value to a variable, and the variable will be created automatically.

For example:

```
x = 10       # Integer
name = "Bob"  # String
is_active = True  # Boolean
```

Variable names should be descriptive and can include letters, numbers, and underscores, but they cannot start with a number.

3. Comments

Comments are used to explain code and are ignored by the Python interpreter. You can create a single-line comment by using the # symbol:

3.1. The types of comments in python are:

- Single-line comment
- Multi-line comment

3.2. The difference between single-line and multi-line comments:

- single-line comments (#) for short explanations'
- multi-line comments (''' ''' or """ """) for longer notes or documentation.

3.3. To write a comment in python

- In python, we use the hash symbol # to write a single-line comment. This line is ignored by the Python interpreter.
- In python, you can use triple quotes (''' or """) to write multi-line comment.

```
# This is a single-line comment
x = 5  # Assign 5 to x
For multi-line comments, you can use triple quotes:
"""
This is a
multi-line comment
"""
```

4. Operators

Operators are special symbols that perform operations on variables and values. Common operators in Python include:

4.1. Arithmetic Operators:

+ (Addition)
- (Subtraction)
* (Multiplication)
/ (Division)
// (Floor Division)
% (Modulus)
** (Exponentiation)

4.2. Comparison Operators:

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

4.3. Logical Operators:

- ✓ And: Returns True if both statements are true
- ✓ Or: Returns True if one of the statements is true
- ✓ Not: Reverse the result, returns False if the result is true

4.4. Assignment Operators:

= (Assign)
+= (Add and assign)
-= (Subtract and assign)
*= (Multiply and assign)
/= (Divide and assign)

4.5. Python Membership Operators are used to test if a sequence is presented in an object:

in : Returns True if a sequence with the specified value is present in the object

not in: Returns True if a sequence with the specified value is not present in the object

4.6. Python Bitwise Operators are used to compare (binary) numbers :

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Practical Activity 2.1.2: Applying python basic concepts

Task:

1. Read the task bellow

As full-stack, you are asked to go to the computer lab to apply python data types, variable, python comments and operators while developing a python program that can perform simple calculation add, substract, multiply, divide and calculate the module and exponent for two entered numbers by user.

2. Refers to provided key reading 2.1.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.1.2 Application of python basic concepts

1. Data Types

Application: Using different data types to store and manipulate various kinds of data.

```
# Different data types
integer_value = 42              # Integer
float_value = 3.14              # Float
string_value = "Hello, Python!"    # String
```

```
boolean_value = True              # Boolean
list_value = [1, 2, 3, 4, 5]      # List
tuple_value = (1, 2, 3)           # Tuple
dict_value = {"name": "Alice", "age": 30}  # Dictionary
set_value = {1, 2, 3}             # Set

print(f"Integer: {integer_value}, Float: {float_value}, String: '{string_value}'")
print(f"Boolean: {boolean_value}, List: {list_value}, Tuple: {tuple_value}")
print(f"Dictionary: {dict_value}, Set: {set_value}")
```

2. Variables

Application: Storing user input and performing operations.

```
# Using variables to store user input
name =str( input("Enter your name: "))
age = int(input("Enter your age: "))

# Displaying the stored variables
print(f"Hello, {name}! You are {age} years old.")
```

3. Comments

Application: Documenting code for better understanding.

```
# This program calculates the area of a rectangle

# Function to calculate area
def calculate_area(length, width):
    return length * width

# Main execution
length = 5  # Length of the rectangle
width = 3   # Width of the rectangle
area = calculate_area(length, width)  # Calculate area

# Display the result
print(f"The area of the rectangle is: {area}")
```

4. Operators

Application: Performing calculations and comparisons.

4.1 A rithmetic Operators

```
# Arithmetic Operators
a = 10
```

```
b = 3

# Addition
addition = a + b
print(f"Addition: {a} + {b} = {addition}")

# Subtraction
subtraction = a - b
print(f"Subtraction: {a} - {b} = {subtraction}")

# Multiplication
multiplication = a * b
print(f"Multiplication: {a} * {b} = {multiplication}")

# Division
division = a / b
print(f"Division: {a} / {b} = {division}")

# Floor Division
floor_division = a // b
print(f"Floor Division: {a} // {b} = {floor_division}")

# Modulus
modulus = a % b
print(f"Modulus: {a} % {b} = {modulus}")

# Exponentiation
exponentiation = a ** b
print(f"Exponentiation: {a} ** {b} = {exponentiation}")
```

4.2 Comparison Operators

```
# Comparison Operators
x = 5
y = 10

# Equal to
print(f"{x} == {y}: {x == y}")

# Not equal to
print(f"{x} != {y}: {x != y}")
```

```
# Greater than
print(f"{x} > {y}: {x > y}")

# Less than
print(f"{x} < {y}: {x < y}")

# Greater than or equal to
print(f"{x} >= {y}: {x >= y}")

# Less than or equal to
print(f"{x} <= {y}: {x <= y}")
```

4.3 Logical Operators

```
# Logical Operators
a = True
b = False

# Logical AND
print(f"a and b: {a and b}")

# Logical OR
print(f"a or b: {a or b}")

# Logical NOT
print(f"not a: {not a}")
```

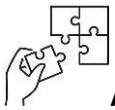
4.4 Assignment Operators

```
# Assignment Operators
num = 10

# Add and assign
num += 5
print(f"After += 5, num = {num}")

# Subtract and assign
num -= 3
print(f"After -= 3, num = {num}")

# Multiply and assign
```

```
num *= 2
print(f"After *= 2, num = {num}")

# Divide and assign
num /= 4
print(f"After /= 4, num = {num}")
```

Points to Remember

- Data types define the kind of values a variable can hold.
- Python supports numeric, sequence, mapping, set, and Boolean data types.
- Variables are containers that store data values.
- Comments are used to explain code.
- Operators are symbols used to perform operations on variables and values.



Application of learning 2.1.

As full-stack developer, you are asked to develop a python program that can perform simple calculation add, substract, multiply, divide and calculate the modulus and exponent for two entered numbers by user. The developed program has to contain comments for better explanation.

Indicative content 2.2: Applying Python Control Structures

Duration: 9 hrs

Practical Activity 2.2.1: Applying Conditional Statements

Task:

1. Read the task bellow:
   As full-stack, you are asked to go to the computer lab to apply if , elif and else statement while developing a python program that can be used when displaying the grade of students depending on entered marks as requested by NESA once analysing marks and student placement.
2. Refers to provided key reading 2.2.1, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.2.1. Applying Conditional Statements

1. Introduction

Conditional statements in Python allow you to execute different blocks of code based on certain conditions.

2. Common frequently used conditional statements

The most common conditional statements are

- ✓ If
- ✓ else.
- ✓ Elif

Here are some applications with program examples:

2.1. Simple if Statement

The if statement in Python is used for decision-making. It allows you to execute a block of code only if a specified condition is true.

Syntax:

```
if condition:
    # Code to execute if the condition is True
```

Key Points:

1. **Condition**: A logical expression that evaluates to either True or False.
2. **Indentation**: The code block inside the if statement must be indented, as Python

uses indentation to define blocks of code.

Application: Checking a condition and executing a block of code if the condition is true.

```
# Simple if statement
age = 18

if age >= 18:
   print("You are eligible to vote.")
```

2.2. if and else Statement

Application: Executing one block of code if the condition is true, and another if it is false.

```
# If-else statement
number = 7

if number % 2 == 0:
   print(f"{number} is even.")
else:
   print(f"{number} is odd.")
```

2.3. if, elif, and else Statement

Application: Checking multiple conditions using elif (short for "else if").

```
# If-elif-else statement
score = 85

if score >= 90:
   grade = 'A'
elif score >= 80:
   grade = 'B'
elif score >= 70:
   grade = 'C'
else:
   grade = 'D'

print(f"Your grade is: {grade}")
```

2.4. Nested Conditional Statements

Application: Using conditional statements within other conditional statements.

```
# Nested if statement
temperature = 30
```

```
if temperature > 0:
   print("The water is liquid.")
   if temperature > 100:
      print("The water is boiling.")
else:
   print("The water is frozen.")
```

2.5. Using Logical Operators in Conditions

Application: Combining conditions using logical operators (and, or, not).

```
# Logical operators in conditions
age = 20
has_id = True
if age >= 18 and has_id:
   print("You can enter the club.")
else:
   print("You cannot enter the club.")
```

Practical Activity 2.2.2: Applying Looping Statements

Task:

1. Read the task bellow:
   As full-stack, you are asked to go to the computer lab to apply looping statement while writing python programs.
2. Refers to provided key reading 2.2.1, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.2.2 Application of Looping Statements

1. Introduction

Looping statements in Python allow you to execute a block of code multiple times. The two primary types of loops are " for loops" and "while loops. "

2. Application of "for loop" and "while loop" statements

Here are some applications with sample examples for each:

2.1. for Loop

- **Application**: Iterating over a sequence (like a list, tuple, or string).

```
# Using a for loop to iterate over a list
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(f"I like {fruit}.")
```

- **for Loop with range()**

Application: Repeating a block of code a specific number of times.

```
# Using a for loop with range()
for i in range(5):  # This will iterate from 0 to 4
    print(f"Iteration {i + 1}")
```

2.2. while Loop

Application: Repeating a block of code as long as a condition is true.

```
# Using a while loop
count = 0

while count < 5:
    print(f"Count is: {count}")
    count += 1  # Increment count
```

3. Nested Loops

Application: Using a loop inside another loop.

```
# Using nested loops
for i in range(3):
    for j in range(2):
        print(f"Outer loop {i}, Inner loop {j}")
```

4. Looping with Conditional Statements

Application: Combining loops with conditional statements to filter results.

```
# Using a loop with a conditional statement
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print("Even numbers:")
for number in numbers:
    if number % 2 == 0:
        print(number)
```

5. Using break and continue Statements

Application: Controlling the flow of loops using break to exit a loop and continue to skip to the next iteration.

```
# Using break and continue
for number in range(1, 11):
    if number == 5:
        print("Breaking the loop at number 5.")
        break  # Exit the loop when number is 5
    print(number)

print("\nUsing continue:")
for number in range(1, 11):
    if number % 2 == 0:
        continue  # Skip even numbers
    print(number)  # Print only odd numbers
```

Practical Activity 2.2.3: Using Jump Statements

Task:

1. Read the task bellow:
   As full-stack, you are asked to go to the computer lab to use jump statement in looping statement while writing python programs.
2. Refers to provided key reading 2.2.3, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.2.3 Using Jump Statements

1. Introduction

Jump statements in Python control the flow of loops and can alter the normal execution sequence.

2. Common frequently used ump statements

The main jump statements are break, continue, and pass. Here are examples of each:

2.1. Break Statement

Application: Exits the nearest enclosing loop when a specified condition is met.

```
# Example of break
for number in range(1, 11):
    if number == 6:
```

```
        print("Breaking the loop at number 6.")
        break  # Exit the loop
    print(number)

# Output will be: 1, 2, 3, 4, 5
```

2.2. Continue Statement

Application: Skips the current iteration of the nearest enclosing loop and continues with the next iteration.

```
# Example of continue
for number in range(1, 11):
    if number % 2 == 0:
        continue  # Skip even numbers
    print(number)  # Print only odd numbers
# Output will be: 1, 3, 5, 7, 9
```

Pass Statement

Application: A null operation; it is syntactically required but does nothing when executed. It's often used as a placeholder.

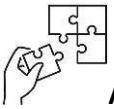
```
# Example of pass
for number in range(1, 6):
    if number == 3:
        pass  # Placeholder for future code
    print(number)

# Output will be: 1, 2, 3, 4, 5
```

Note: All jump statements are written in small letter while developing python program.

Points to Remember

- Conditional statements are used to execute different code blocks based on specific conditions.
- The if statement is used to execute a block of code if a condition is true.
- The else statement is used to execute a block of code if the if condition is false.
- The elif statement is used to test additional conditions if the previous if or elif conditions are false.
- Indentation is crucial for defining the code blocks within conditional statements.
- Comparison operators (e.g., ==, !=, <, >, <=, >=) are used to create conditions.

- Logical operators (e.g., and, or, not) can be used to combine multiple conditions.
- Nested conditional statements can be used to create more complex decision-making logic.
- Proper indentation is essential for ensuring correct code execution.
- Testing different conditions can help verify the correctness of conditional statements.
- Looping statements are used to repeat a block of code multiple times in Python.
- The for loop is used to iterate over a sequence of elements (e.g., lists, tuples, strings).
- The while loop is used to repeat a block of code as long as a condition is true.
- The break statement can be used to exit a loop prematurely.
- The continue statement can be used to skip the current iteration of a loop and move to the next one.
- Nested loops can be used to create more complex looping structures.
- Indentation is crucial for defining the code block within loops.
- Break: Terminates the loop entirely when a condition is met.
- Continue: Skips the current iteration and continues with the next one.
- Pass: Does nothing and is useful for maintaining the structure of code where a statement is syntactically required.



Application of learning 2.2.

Write a python program that can display all even numbers from 0 to 100 and the program skip 60 and stops execution if value is equal to 90.

Indicative content 2.3: Applying Functions in Python

Duration: 9 hrs

Theoretical Activity 2.3.1: Description of function in python

Tasks:

1. You are requested to answer the following questions related to the description of functions in python:
   i. Define function in python
   ii. Differentiate two (2) types of function in python
   iii. Elaborate characteristics and Advantages of using functions in python
2. Write your findings on paper/flipchart
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 2.3.1 and ask questions where necessary.

Key readings 2.3.1. Description of function in python

1. Definition of Function

A function in Python is a block of reusable code that performs a specific task. It is defined using the def keyword, followed by the function name and parentheses containing any parameters.

Functions help organize code, making it more modular and easier to understand.

Example of a Function Definition:

```
def greet(name):
    """This function greets the person passed as a parameter."""
    print(f"Hello, {name}!")
```

2. Characteristics of Functions

1. **Modularity**: Functions allow you to break your program into smaller, manageable pieces.
2. **Reusability**: Once defined, functions can be reused multiple times throughout the code.
3. **Parameters and Return Values**: Functions can accept parameters and return values, making them flexible.
4. **Encapsulation**: Functions encapsulate the logic of a task, which can improve code clarity.

5. **Scope**: Variables defined inside a function are local to that function unless specified otherwise.

3. Advantages of Functions

1. **Improved Readability**: Breaking code into functions improves readability and organization.
2. **Easier Maintenance**: Functions can be modified independently, making maintenance simpler.
3. **Code Reusability**: Functions can be reused across different parts of a program or even in different programs.
4. **Debugging**: Isolating functionality into functions simplifies debugging since you can test each function independently.
5. **Abstraction**: Functions allow you to abstract complex operations, making it easier to understand and use.

4. Types of Functions

4.1. Built-in Functions

These are functions that are pre-defined in Python and can be used without any additional code.

Examples include:

- **print():** Outputs data to the console.
- **len():** Returns the length of an object.
- **type():** Returns the type of an object.
- **sum():** Returns the sum of a collection of numbers.

Example of a Built-in Function:

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)  # Using the built-in sum function
print(f"The total is: {total}")
```

4.2. User-Defined Functions

These are functions that you define yourself to perform specific tasks. You can create them using the def keyword.

Example of a User-Defined Function

```
def add(a, b):
    """This function returns the sum of two numbers."""
    return a + b
result = add(5, 3)  # Calling the user-defined function
print(f"The sum is: {result}")
```

Practical Activity 2.3.2: Creating function in python

Task:

1. Read the task bellow:
   As full-stack developer, you are requested to go to the computer lab to create a function with arguments, default parameter value, passing a list as an argument and calling a function.
2. Refers to provided key reading 2.3.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.3.2 Creation of Function in Python

1. Defining a Function

In Python, you can define a function using the def keyword. A function is a block of reusable code that performs a specific task. It can take inputs (called parameters), execute code, and optionally return a result.

```
def function_name(parameters):
    """
    Optional docstring: Describes the function's purpose.
    """
    # Code block to execute
    return value  # Optional
```

Example

```
def greet(name):
    """This function greets the person passed as a parameter."""
    print(f"Hello, {name}!")
```

2. Arguments

Arguments are the values you pass to a function when calling it. You can define functions with different types of arguments:

Positional Arguments:

These must be provided in the correct order.

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

result = add(5, 3)  # 5 and 3 are positional arguments
print(f"The sum is: {result}")
```

Keyword Arguments:

You can specify arguments by name, allowing you to pass them in any order.

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet(pet_name="Buddy", animal_type="dog")  # Using keyword arguments
```

3. Default Parameter Value

You can set default values for parameters in a function. If a value is not provided during the function call, the default value will be used.

Example:

```
def greet(name="Guest"):
    """This function greets the person passed as a parameter with a default value."""
    print(f"Hello, {name}!")

greet()        # Uses default value
greet("Alice")  # Overrides default value
```

4. Passing a List as an Argument

You can pass a list (or any other collection) as an argument to a function. Inside the function, you can manipulate it as needed.

Example:

```
def print_fruits(fruits):
    """This function prints each fruit in the list."""
    for fruit in fruits:
        print(fruit)

fruit_list = ["apple", "banana", "cherry"]
print_fruits(fruit_list)  # Passing a list as an argument
```

5. Calling a Function

To call a function, simply use its name followed by parentheses. If the function requires arguments, provide them within the parentheses.

Example:

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

# Calling the function with arguments
result = multiply(4, 5)
print(f"The product is: {result}")  # Outputs: The product is: 20
```

Practical Activity 2.3.3: Applying special purpose functions

Task:

1. Read the task bellow:

   As full-stack developer, you are requested to go to the computer lab to apply special purpose functions such as Lambda, Python Generators, Python Closures, Python Decorators, Recursive function, and Higher-order function.
2. Refers to provided key reading 2.3.3, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.3.3 Applying special purpose functions

Lambda Functions

What is a Lambda Function?
Lambda functions are similar to user-defined functions but without a name.

They're commonly referred to as anonymous functions.

Lambda functions are efficient whenever you want to create a function that will only contain simple expressions – that is, expressions that are usually a single line of a statement. They're also useful when you want to use the function once.

How to Define a Lambda Function

You can define a lambda function like this:

lambda argument(s) : expression

1. lambda is a keyword in Python for defining the anonymous function.
2. argument(s) is a placeholder, that is a variable that will be used to hold the value you want to pass into the function expression. A lambda function can have multiple variables depending on what you want to achieve.
3. expression is the code you want to execute in the lambda function.
   Notice that the anonymous function does not have a return keyword. This is because the anonymous function will automatically return the result of the expression in the function once it is executed.

   Let's look at an example of a lambda function to see how it works. We'll compare it

to a regular user-defined function.

Assume I want to write a function that returns twice the number I pass it. We can define a user-defined function as follows:

```
def f(x):
  return x * 2
f(3)
>> 6
```

Now for a lambda function. We'll create it like this:

```
lambda x: x * 3
```

As we explained above, the lambda function does not have a return keyword. As a result, it will return the result of the expression on its own. The x in it also serves as a placeholder for the value to be passed into the expression. You can change it to whatever you want.

Now if you want to call a lambda function, you will use an approach known as immediately invoking the function. That looks like this:

```
(lambda x : x * 2)(3)
>> 6
```

The reason for this is that since the lambda function does not have a name you can invoke (it's anonymous), you need to enclose the entire statement when you want to call it.

When Should You Use a Lambda Function?

You should use the lambda function to create simple expressions. For example, expressions that do not include complex structures such as if-else, for-loops, and so on.

So, for example, if you want to create a function with a for-loop, you should use a user-defined function.

Common Use Cases for Lambda Functions

How to Use a Lambda Function with Iterables

An iterable is essentially anything that consists of a series of values, such as characters, numbers, and so on.

In Python, iterables include strings, lists, dictionaries, ranges, tuples, and so on. When working with iterables, you can use lambda functions in conjunction with two common functions: filter() and map().

Filter()

When you want to focus on specific values in an iterable, you can use the filter function. The following is the syntax of a filter function:

filter(function, iterable)

As you can see, a filter function requires another function that contains the expression or operations that will be performed on the iterable.

For example, say I have a list such as [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Now let's say that I'm only interested in those values in that list that have a remainder of 0 when divided by 2. I can make use of filter() and a lambda function.
Firstly I will use the lambda function to create the expression I want to derive like this:

lambda x: x % 2 == 0
Then I will insert it into the filter function like this:

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filter(lambda x: x % 2 == 0, list1)
list(filter(lambda x: x % 2 == 0, list1))
>> [2, 4, 6, 8, 10]
```

Map()

You use the map() function whenever you want to modify every value in an iterable.

map(function, iterable)

For example, let's say I want to raise all values in the below list to the power of 2. I can easily do that using the lambda and map functions like this:

```
list1 = [2, 3, 4, 5]
list(map(lambda x: pow(x, 2), list1))
>> [4, 9, 16, 25]
```

Pandas Series

Another place you'll use lambda functions is in data science when creating a data frame from Pandas.

A series is a data frame column. You can manipulate all of the values in a series by

using the lambda function.

For example, if I have a data frame with the following columns and want to convert the values in the name column to lower case, I can do so using the Pandas apply function and a Python lambda function like this:

```
import pandas as pd

df = pd.DataFrame(
   {"name": ["IBRAHIM", "SEGUN", "YUSUF", "DARE", "BOLA", "SOKUNBI"],
    "score": [50, 32, 45, 45, 23, 45]
   }
)
```

	name	score
0	IBRAHIM	50
1	SEGUN	32
2	YUSUF	45
3	DARE	45
4	BOLA	23
5	SOKUNBI	45

```
df["lower_name"] = df["name"].apply(lambda x: x.lower())
```

The apply function will apply each element of the series to the lambda function. The lambda function will then return a value for each element based on the expression you passed to it. In our case, the expression was to lowercase each element.

	name	score	lower_name
0	IBRAHIM	50	ibrahim
1	SEGUN	32	segun
2	YUSUF	45	yusuf
3	DARE	45	dare
4	BOLA	23	bola
5	SOKUNBI	45	sokunbi

1. Python Generators

Definition: Generators are a type of iterable, like lists or tuples. Unlike lists, they do not store their contents in memory; instead, they generate items on-the-fly using the yield keyword.

Example:

```
# Generator function to yield numbers
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
# Using the generator
for number in count_up_to(5):
    print(number)  # Outputs: 1, 2, 3, 4, 5
```

2. Python Closures

Definition: A closure is a function that remembers its enclosing lexical scope even when the program flow is no longer in that scope.

Example:

```
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

# Create a closure
my_greeting = outer_function("Hello, World!")
my_greeting()  # Outputs: Hello, World!
```

3. Python Decorators

Definition: Decorators are a way to modify or enhance functions or methods without changing their code. They are applied using the @decorator syntax.

Example:

```
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed before {}".format(original_function.__name__))
        return original_function()
    return wrapper_function
@decorator_function
def display():
    print("Display function executed.")
# Calling the decorated function
display()
# Outputs:
# Wrapper executed before display
# Display function executed.
```

4. Recursive Function

Definition: A recursive function is a function that calls itself in order to solve a problem.

Example:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
# Calling the recursive function
result = factorial(5)
print(f"The factorial of 5 is: {result}")  # Outputs: The factorial of 5 is: 120
```

5. Higher-Order Function

Definition: A higher-order function is a function that takes one or more functions as arguments or returns a function as its result.

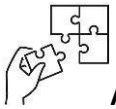
Example:

```
def apply_function(func, value):
    return func(value)

# Function to square a number
def square(x):
```

```
    return x * x

# Using a higher-order function
result = apply_function(square, 4)
print(f"The square of 4 is: {result}")  # Outputs: The square of 4 is: 16
```

Points to Remember

- Functions are fundamental building blocks in Python programming. They promote code reuse, readability, and maintainability.
- Built-in Functions: These are functions that are pre-defined in Python and can be used without any additional code.
- User-Defined Functions: These are functions that you define yourself to perform specific tasks. You can create them using the def keyword.
- **Characteristics of Functions include:** Modularity, Reusability, Parameters and Return Values, Encapsulation and Scope.
- **Advantages of Functions include:** Improved Readability, Easier Maintenance, Code Reusability, Debugging and Abstraction.
- Creation of Function in Python you can follow the following steps:
    - **Step 1:** **Defining a Function**
    - **Step 2:** Arguments
    - **Step 3:** Default Parameter Value
    - **Step 4:** Passing a List as an Argument
    - **Step 5:** Calling a Function
- Special purpose functions offer unique capabilities and can be used to solve specific problems.
- Lambda functions are concise and often used for short, simple expressions.
- Generators provide efficient ways to generate values on demand.
- Closures can be used to create functions with state.
- Decorators can modify the behavior of other functions without directly changing their code.
- Recursive functions can be used to solve problems that can be broken down into smaller, similar sub problems.
- Higher-order functions can be used to create more flexible and reusable code.



Application of learning 2.3.

Write a Python program that defines a function that generates Fibonacci numbers using a generator, uses a lambda function to filter even numbers from the generated sequence then prints the first 10 even Fibonacci numbers.

Indicative content 2.4: Applying of Python Collections

Duration: 9 hrs

Theoretical Activity 2.4.1: Description of python Collections

Tasks:

1. You are requested to answer the following questions related to the description of collections in python:
    i. Describe collection Types in Python
    ii. Explain Tools from the Collections Module
2. Write your findings on paper/flipchart
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 2.4.1 and ask questions where necessary.

Key readings 2.4.1 Description of python Collections

1. Collection Types in Python

1.1. Lists

Definition: Lists are ordered, mutable collections that can hold a variety of object types. Elements can be added, removed, or modified.

Syntax:

```
list_name = [item1, item2, item3, ...]
```

- Lists are defined using square brackets [].
- Items in the list are separated by commas,.
- A list can contain elements of any data type (e.g., integers, strings, other lists, etc.).

Example:

```
fruits = ['apple', 'banana', 'cherry']
fruits.append('orange')  # Add an item
print(fruits)  # Outputs: ['apple', 'banana', 'cherry', 'orange']
```

1.2. Tuples

Definition: Tuples are ordered, immutable collections. Once created, their elements cannot be changed.

Syntax:

```
tuple_name = (item1, item2, item3, ...)
```

- Tuples are defined using parentheses ().
- Items in the tuple are separated by commas ,.
- Tuples can contain elements of any data type.
- A tuple with a single element requires a trailing comma `(value,)` to differentiate it from a regular value.

Example:

```
colors = ('red', 'green', 'blue')
print(colors[1])  # Outputs: green
```

1.3. Dictionaries

Definition: Dictionaries are unordered collections of key-value pairs. Keys must be unique and immutable, while values can be of any type.

Syntax: my_dict = {'key1': 'value1', 'key2': 'value2'}

- Dictionaries are defined using curly braces {}.
- Each key-value pair is separated by a colon :.
- Pairs are separated by commas ,

Example:

```
student = {'name': 'Alice', 'age': 25}
student['age'] = 26  # Modify value
print(student)  # Outputs: {'name': 'Alice', 'age': 26}
```

1.4. Sets

Definition: Sets are unordered collections of unique elements. They are mutable and do not allow duplicate values.

Syntax: my_set = {item1, item2, item3, ...}

- Sets are defined using curly braces {} or the set() constructor.
- Items in a set are separated by commas ,.
- Sets do not allow duplicate elements.
- Sets are unordered, so they do not support indexing or slicing.

Example:

```
unique_numbers = {1, 2, 2, 3}
print(unique_numbers)  # Outputs: {1, 2, 3}
```

1.5. Frozen Set

Definition: A frozen set is an immutable version of a set. Once created, its elements cannot be changed.

Syntax: my_frozenset = frozenset(iterable)

- Use the frozenset() constructor to create a frozen set.
- The iterable can be any iterable object like a list, tuple, set, or string.

Example:

```
immutable_set = frozenset([1, 2, 3, 4])
print(immutable_set)  # Outputs: frozenset({1, 2, 3, 4})
```

1.6. ChainMaps

Definition: A ChainMap groups multiple dictionaries into a single view. It allows for searching through multiple dictionaries as if they were one.

Syntax: from collections import ChainMap

Example:

```
from collections import ChainMap

dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
combined = ChainMap(dict1, dict2)

print(combined['b'])  # Outputs: 2 (from dict1)
```

1.7. Deques

Definition: Deques (double-ended queues) are mutable sequences that allow fast appends and pops from both ends.

Syntax: from collections import deque

Example:

```
from collections import deque

my_deque = deque(['a', 'b', 'c'])
my_deque.append('d')  # Add to the right
my_deque.appendleft('z')  # Add to the left
print(my_deque)  # Outputs: deque(['z', 'a', 'b', 'c', 'd'])
```

2. Specialized Tools from the Collections Module

2.1. Counter

Definition: A Counter is a dictionary subclass for counting hashable objects. It makes it easy to count occurrences of elements.

Example:

```
from collections import Counter

count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'banana'])
print(count)  # Outputs: Counter({'banana': 3, 'apple': 2, 'orange': 1})
```

2.2. OrderedDict

Definition: An OrderedDict is a dictionary subclass that maintains the order of keys based on their insertion order.

Example:

```
from collections import OrderedDict

ordered_dict = OrderedDict()
ordered_dict['apple'] = 1
ordered_dict['banana'] = 2
ordered_dict['cherry'] = 3
print(ordered_dict)  # Outputs: OrderedDict([('apple', 1), ('banana', 2), ('cherry', 3)])
```

2.3. Defaultdict

Definition: A defaultdict is a dictionary subclass that provides a default value for a nonexistent key. It avoids KeyErrors.

Example:

```
from collections import defaultdict

default_dict = defaultdict(int)  # Default value is 0
default_dict['a'] += 1
default_dict['b'] += 2
print(default_dict)  # Outputs: defaultdict(<class 'int'>, {'a': 1, 'b': 2})
```

Practical Activity 2.4.2: Performing common operations on collection

Task:

1. Read the task bellow:

As full-stack developer, you are requested to go to the computer lab to perform the following operations on collections:

a) Adding and Removing Elements
b) Accessing and Iterating Over Elements
c) Filtering and Sorting
d) Set Operations and Counting
e) Stack and Queue Operations

2. Refers to provided key reading 2.4.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.4.2 Perform common operations on collection

1. Adding and Removing Elements

1.1. Lists

Adding:

```
fruits = ['apple', 'banana']
fruits.append('cherry')  # Add to the end
fruits.insert(1, 'orange')  # Add at index 1
print(fruits)  # Outputs: ['apple', 'orange', 'banana', 'cherry']
```

Removing:

```
fruits.remove('banana')  # Remove by value
popped_fruit = fruits.pop()  # Remove last item and return it
print(fruits)  # Outputs: ['apple', 'orange']
print(f"Popped fruit: {popped_fruit}")  # Outputs: Popped fruit: cherry
```

1.2. Dictionaries

Adding:

```
student = {'name': 'Alice'}
student['age'] = 25  # Add new key-value pair
print(student)  # Outputs: {'name': 'Alice', 'age': 25}
```

Removing:

```
del student['age']  # Remove key-value pair by key
print(student)  # Outputs: {'name': 'Alice'}
```

2. Accessing and Iterating Over Elements

2.1. Lists

```
# Accessing elements
print(fruits[0])  # Outputs: apple

# Iterating over elements
for fruit in fruits:
    print(fruit)
```

2.2. Dictionaries

```
# Accessing values
print(student['name'])  # Outputs: Alice

# Iterating over keys and values
for key, value in student.items():
    print(f"{key}: {value}")
```

3. Filtering and Sorting

3.1. Sorting

Sorting Lists

1. Using sorted() (Returns a New Sorted List)

```
numbers = [5, 2, 9, 1, 7]
# Ascending Order
ascending = sorted(numbers)
print("Ascending:", ascending)  # Output: [1, 2, 5, 7, 9]
# Descending Order
descending = sorted(numbers, reverse=True)
print("Descending:", descending)  # Output: [9, 7, 5, 2, 1]
```

2. Using .sort() (Modifies the List In-Place)

```
numbers = [5, 2, 9, 1, 7]
# Sort in Ascending Order
numbers.sort()
print("Ascending:", numbers)  # Output: [1, 2, 5, 7, 9]
# Sort in Descending Order
numbers.sort(reverse=True)
print("Descending:", numbers)  # Output: [9, 7, 5, 2, 1]
```

Sorting Tuples

Tuples are **immutable**, so you can only use sorted() (returns a new sorted list).

```
numbers = (5, 2, 9, 1, 7)
ascending = sorted(numbers)  # Output: [1, 2, 5, 7, 9]
descending = sorted(numbers, reverse=True)  # Output: [9, 7, 5, 2, 1]
print("Ascending:", ascending)
print("Descending:", descending)
```

Sorting Dictionaries

You can sort dictionaries based on **keys** or **values**.

1. Sorting by Keys

```
students = {'Alice': 85, 'Bob': 92, 'Charlie': 78}
# Ascending by key
ascending = dict(sorted(students.items()))
print("Ascending by keys:", ascending)
# Descending by key
descending = dict(sorted(students.items(), reverse=True))
```

```
print("Descending by keys:", descending)
```

2. Sorting by Values

```
# Ascending by value
ascending = dict(sorted(students.items(), key=lambda x: x[1]))
print("Ascending by values:", ascending)
# Descending by value
descending = dict(sorted(students.items(), key=lambda x: x[1], reverse=True))
print("Descending by values:", descending)
```

Sorting Sets

Sets are unordered collections, so you must convert them into a list first.

```
numbers = {5, 2, 9, 1, 7}
ascending = sorted(numbers)  # [1, 2, 5, 7, 9]
descending = sorted(numbers, reverse=True)  # [9, 7, 5, 2, 1]
print("Ascending:", ascending)
print("Descending:", descending)
```

Sorting Custom Objects

If you have a list of dictionaries or objects, use the key parameter.

```
students = [
   {"name": "Alice", "score": 85},
   {"name": "Bob", "score": 92},
   {"name": "Charlie", "score": 78}
]

# Sort by score (ascending)
ascending = sorted(students, key=lambda x: x["score"])
print("Ascending:", ascending)

# Sort by score (descending)
descending = sorted(students, key=lambda x: x["score"], reverse=True)
print("Descending:", descending)
```

Summary

Collection Type	Sorting Method	Ascending	Descending
List	.sort() (modifies) or sorted()	sorted(list)	sorted(list, reverse=True)
Tuple	sorted()	sorted(tuple)	sorted(tuple, reverse=True)
Dictionary (Keys)	sorted(dict.items())	sorted(dict.items())	sorted(dict.items(), reverse=True)
Dictionary (Values)	sorted(dict.items(), key=lambda x: x[1])	sorted(dict.items(), key=lambda x: x[1])	sorted(dict.items(), key=lambda x: x[1], reverse=True)
Set	sorted(set)	sorted(set)	sorted(set, reverse=True)
Custom Objects	sorted(objects, key=lambda x: x['property'])	sorted(objects, key=lambda x: x['property'])	sorted(objects, key=lambda x: x['property'], reverse=True)

3.2. Filtering

Filtering involves selecting specific elements from collections (list, tuple, dictionary) based on a condition. Below is how you can filter these collections effectively.

1. Filtering a List

Using filter():

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

# Filter even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

Using List Comprehension:

```
# Filter even numbers
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)  # Output: [2, 4, 6]
```

2. Filtering a Tuple

Using filter():

```
numbers = (1, 2, 3, 4, 5, 6)
```

```
# Filter even numbers
even_numbers = tuple(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: (2, 4, 6)
```

Using Generator Expression:

```
# Filter even numbers
even_numbers = tuple(x for x in numbers if x % 2 == 0)
print(even_numbers)  # Output: (2, 4, 6)
```

3. Filtering a Dictionary

Filtering Based on Values:

```
data = {"a": 1, "b": 2, "c": 3, "d": 4}

# Keep only items with values greater than 2
filtered_data = {k: v for k, v in data.items() if v > 2}
print(filtered_data)  # Output: {'c': 3, 'd': 4}
```

Filtering Based on Keys:

```
# Keep only keys that start with 'b'
filtered_data = {k: v for k, v in data.items() if k.startswith('b')}
print(filtered_data)  # Output: {'b': 2}
```

Filtering Both Keys and Values:

```
# Keep items where the key starts with 'c' and the value is greater than 2
filtered_data = {k: v for k, v in data.items() if k.startswith('c') and v > 2}
print(filtered_data)  # Output: {'c': 3}
```

Comparison of Filtering Techniques

Collection	Method	Example
List	filter()	Filter even numbers: filter(lambda x: x % 2 == 0, numbers)
List	List Comprehension	[x for x in numbers if x % 2 == 0]
Tuple	filter()	tuple(filter(lambda x: x % 2 == 0, numbers))
Tuple	Generator Expression	tuple(x for x in numbers if x % 2 == 0)
Dictionary	Dictionary Comprehension	{k: v for k, v in data.items() if v > 2}

Each method has its use case:

- Use filter() for functional-style filtering.
- Use comprehensions for a concise, readable approach.
- For dictionaries, comprehensions are the primary way to filter based on keys or values.

4. Set Operations and Counting

4.1. Sets

Basic Set Operations:

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
union_set = set_a | set_b  # or set_a.union(set_b)
print(union_set)  # Outputs: {1, 2, 3, 4, 5}

# Intersection
intersection_set = set_a & set_b  # or set_a.intersection(set_b)
print(intersection_set)  # Outputs: {3}

# Difference
difference_set = set_a - set_b  # or set_a.difference(set_b)
print(difference_set)  # Outputs: {1, 2}
```

4.2. **Counting Elements** (Using Counter):

```
from collections import Counter

elements = ['apple', 'banana', 'apple', 'orange', 'banana', 'banana']
count = Counter(elements)
print(count)  # Outputs: Counter({'banana': 3, 'apple': 2, 'orange': 1})
```

5. Stack and Queue Operations

5.1. Stack Operations (Using Lists)

Stack: LIFO (Last In, First Out)

```
stack = []

# Push
stack.append('A')
stack.append('B')
stack.append('C')

# Pop
```

```
top_element = stack.pop()
print(top_element)  # Outputs: C
print(stack)  # Outputs: ['A', 'B']
```

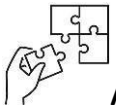
Queue Operations (Using deque)

Queue: FIFO (First In, First Out)

```
from collections import deque

queue = deque()

# Enqueue
queue.append('A')
queue.append('B')
queue.append('C')

# Dequeue
first_element = queue.popleft()
print(first_element)  # Outputs: A
print(queue)  # Outputs: deque(['B', 'C'])
```

Points to Remember

- Python provides various built-in collection types, each serving different purposes: Lists, Tuples, dictionaries, sets, frozen sets, ChainMaps and deques.
- The collections module enhances functionality with specialized tools like: Counter, OrderedDict and defaultdict.
- These are common operations you can perform on various collection types in Python. Lists, dictionaries, sets: Adding and Removing Elements, Accessing and Iterating over Elements, Filtering and Sorting, Set Operations and Counting and Stack and Queue Operations.
- Specialized tools such as deques provide powerful ways to manage and manipulate data, making Python a versatile language for handling collections.



Application of learning 2.4.

Write a Python program that creates a list of numbers, filters out even numbers, sorts the remaining numbers in ascending order and then prints the sorted list.

Indicative content 2.5: Performing File Handling

Duration: 9 hrs

Theoretical Activity 2.5.1: Description of file handling libraries

Tasks:

1. You are requested to answer the following questions related to the description of file handling libraries in python:
   i.Describe the following file handling libraries
   a) Os module
   b) Pathlib module
   c) Shutil module
   d) Pandas library
2. Write your findings on paper/flipchart
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 2.5.1 and ask questions where necessary.

Key readings 2.5.1. Description of file Handling libraries

File handling libraries in Python, include os, pathlib, shutil, and pandas. Each library serves different purposes and offers various functionalities for file and directory manipulation.

1. os Module

Description: The os module provides a way to use operating system-dependent functionality like reading or writing to the file system, working with directories, and handling environment variables.

Common Functions:

- ✓ os.listdir(path): Returns a list of files and directories in the specified path.
- ✓ os.mkdir(path): Creates a directory at the specified path.
- ✓ os.remove(path): Deletes a file at the specified path.
- ✓ os.rename(src, dst): Renames a file or directory.
- ✓ os.path: Contains functions to manipulate pathnames (e.g., os.path.join, os.path.exists).

Example:

```
import os

# List files in the current directory
files = os.listdir('.')
print(files)

# Create a new directory
os.mkdir('new_folder')

# Remove a file (make sure it exists)
# os.remove('file_to_delete.txt')
```

2. pathlib Module

Description: The pathlib module offers an object-oriented approach to file system paths. It allows easier manipulation of paths and provides a more intuitive syntax.

Common Classes and Methods:

- ✓ Path: Represents a filesystem path.
- ✓ Path.exists(): Checks if the path exists.
- ✓ Path.mkdir(): Creates a new directory.
- ✓ Path.rmdir(): Removes a directory.
- ✓ Path.read_text(): Reads the contents of a text file.

Example:

```
from pathlib import Path

# Create a Path object
path = Path('example.txt')

# Check if the file exists
if path.exists():
    print(f"{path} exists.")
else:
    # Create a new file
    path.write_text("Hello, World!")

# Read the file
content = path.read_text()
print(content)
```

3. shutil Module

Description: The shutil module provides a higher-level interface for file operations, particularly for copying and removing files and directories. It simplifies tasks like file and directory management.

Common Functions:

- ✓ shutil.copy(src, dst): Copies a file from src to dst.
- ✓ shutil.move(src, dst): Moves a file or directory from src to dst.
- ✓ shutil.rmtree(path): Deletes an entire directory tree.
- ✓ shutil.make_archive(base_name, format, root_dir): Creates a zip or tar archive.

Example:

```
import shutil
# Copy a file
shutil.copy('source.txt', 'destination.txt')
# Move a directory
shutil.move('old_directory', 'new_directory')
# Remove a directory tree
# shutil.rmtree('directory_to_delete')
```

4. pandas Library

Description: While primarily a data analysis library, pandas provides powerful tools for reading from and writing to various file formats, including CSV, Excel, JSON, and more. It simplifies data manipulation and analysis.

Common Functions:

- ✓ pandas.read_csv(filepath): Reads a CSV file into a DataFrame.
- ✓ DataFrame.to_csv(filepath): Writes a DataFrame to a CSV file.
- ✓ pandas.read_excel(filepath): Reads an Excel file into a DataFrame.
- ✓ DataFrame.to_excel(filepath): Writes a DataFrame to an Excel file.

Example:

```
import pandas as pd
# Read a CSV file into a DataFrame
df = pd.read_csv('data.csv')
# Display the first few rows
print(df.head())
# Write DataFrame to a new CSV file
df.to_csv('new_data.csv', index=False)
```

Practical Activity 2.5.2: Practicing open and read file

Task:

1. Read the task bellow:

As full-stack developer, you are requested to go to the computer lab to perform the following operations on file:

a) Open a File
b) Read File Permissions

2. Refers to provided key reading 2.5.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.5.2 Practice to open and read file

Practice reading a file in Python, covering how to open a file and check file permissions.

1. Open a File

To open a file in Python, you use the built-in open() function. This function takes at least one argument: the path to the file. You can also specify a second argument to indicate the mode in which you want to open the file.

Common File Modes:

- 'r': Read (default mode) - Opens a file for reading.
- 'w': Write - Opens a file for writing (creates a new file or truncates an existing file).
- 'a': Append - Opens a file for appending (data will be written at the end).
- 'b': Binary - Opens a file in binary mode.
- 't': Text - Opens a file in text mode (default).

Example:

```
# Open a file for reading
file_path = 'example.txt'  # Ensure this file exists before running
try:
    with open(file_path, 'r') as file:
        content = file.read()  # Read the entire file
        print(content)  # Print the file content
except FileNotFoundError:
    print(f"The file {file_path} does not exist.")
except IOError:
    print("An error occurred while reading the file.")
```

2. Read File Permissions

Before opening a file, you may want to check its permissions to ensure you have the appropriate access rights. You can use the os module to check file permissions.

Example:

```
import os
file_path = 'example.txt'  # Ensure this file exists
# Check if the file exists
if os.path.exists(file_path):
 # Get file permissions
   permissions = os.stat(file_path).st_mode
   # Check read permission
   can_read = bool(permissions & 0o400)  # Owner can read
   can_write = bool(permissions & 0o200)  # Owner can write
   can_execute = bool(permissions & 0o100)  # Owner can execute
   print(f"Read permission: {can_read}")
   print(f"Write permission: {can_write}")
   print(f"Execute permission: {can_execute}")
else:
   print(f"The file {file_path} does not exist.")
```

Practical Activity 2.5.3: Performing write/create and delete file

Task:

1. Read the task bellow:

As full-stack developer, you are requested to go to the computer lab to perform the following operations on file:

- Write
- Create
- Delete file

2. Refers to provided key reading 2.5.3, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.5.3 Performing write/create and delete file

1. Create a New File

To create a new file in Python, you can use the open() function with the 'w' (write) or 'x' (exclusive creation) mode. The 'w' mode will create a new file or overwrite an existing file, while 'x' will raise an error if the file already exists.

Example:

```
# Create a new file
file_path = 'new_file.txt'

try:
    with open(file_path, 'w') as file:
        file.write("This is a new file created with Python.\n")
    print(f"File {file_path} created successfully.")
except IOError:
    print("An error occurred while creating the file.")
```

2. Write to an Existing File

To write to an existing file, you can open it in 'a' (append) or 'w' mode. The 'a' mode will add content to the end of the file without deleting the current content.

Example:

```
# Write to an existing file
existing_file_path = 'new_file.txt'

try:
    with open(existing_file_path, 'a') as file:
        file.write("Appending new content to the existing file.\n")
    print(f"Content appended to {existing_file_path} successfully.")
except IOError:
    print("An error occurred while writing to the file.")
```

3. Remove a File

To delete a file, you can use the os.remove() function from the os module.

Example:

```
import os

# Remove a file
file_to_remove = 'new_file.txt'

try:
    os.remove(file_to_remove)
```

```
    print(f"File {file_to_remove} deleted successfully.")
except FileNotFoundError:
    print(f"The file {file_to_remove} does not exist.")
except PermissionError:
    print(f"Permission denied to delete the file {file_to_remove}.")
except Exception as e:
    print(f"An error occurred: {e}")
```

4. Delete a Folder

To delete a folder, you can use os.rmdir() for empty directories or shutil.rmtree() for directories that contain files.

Example (Deleting an Empty Folder):

```
# Delete an empty folder
folder_to_remove = 'empty_folder'

try:
    os.rmdir(folder_to_remove)
    print(f"Folder {folder_to_remove} deleted successfully.")
except FileNotFoundError:
    print(f"The folder {folder_to_remove} does not exist.")
except OSError:
    print(f"The folder {folder_to_remove} is not empty or cannot be deleted.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Example (Deleting a Non-Empty Folder):

```
import shutil

# Delete a non-empty folder
non_empty_folder = 'non_empty_folder'

try:
    shutil.rmtree(non_empty_folder)
    print(f"Folder {non_empty_folder} deleted successfully.")
except FileNotFoundError:
    print(f"The folder {non_empty_folder} does not exist.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Practical Activity 2.5.4: Applying python best practices

Task:

1. Read the task bellow:

As full-stack developer, you are requested to go to the computer lab to apply the following:

i. Readability and Style
ii. Use of Built-in Features
iii. Efficiency and Memory Usage
iv. Error Handling and Testing

2. Refers to provided key reading 2.5.4, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 2.5.4 Application of python best practices

Best practices for writing Python code, focusing on readability and style, the use of built-in features, efficiency and memory usage, and error handling and testing.

1. Readability and Style

PEP 8: Follow the Python Enhancement Proposal (PEP) 8 style guide, which outlines conventions for writing clean and readable code.

- ✓ **Indentation**: Use 4 spaces per indentation level.
- ✓ **Line Length**: Limit lines to 79 characters.
- ✓ **Naming Conventions**: Use descriptive variable and function names. Use snake_case for variables and functions, and CamelCase for classes.

Example:

```
def calculate_area(radius):
    """Calculate the area of a circle given its radius."""
    return 3.14 * radius ** 2
```

- **Docstrings**: Use docstrings to describe the purpose of functions and classes. This helps others understand your code.

2. Use of Built-in Features

Leverage Built-in Functions: Use Python's built-in functions and libraries whenever possible, as they are optimized and thoroughly tested.

Example:

```
# Instead of manually calculating the sum of a list
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)  # Use built-in sum function
```

List Comprehensions: Use list comprehensions for creating lists in a concise and readable way.

Example:

```
squares = [x ** 2 for x in range(10)]  # List comprehension for squares
```

3. Efficiency and Memory Usage

Use Generators: When working with large datasets, use generators to save memory. Generators yield items one at a time and do not load everything into memory.

Example:

```
def generate_numbers(n):
    for i in range(n):
        yield i * 2

# Using the generator
for number in generate_numbers(10):
    print(number)
```

Avoid Unnecessary Copies: Be mindful of operations that create unnecessary copies of data. For instance, use in-place modifications where applicable.

Example:

```
# Instead of creating new lists
my_list = [1, 2, 3]
my_list.append(4)  # Modify in place
```

4. Error Handling and Testing

Use Exceptions: Use try and except blocks for error handling to make your code robust. Handle specific exceptions rather than using a bare except.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

Assertions: Use assertions to enforce conditions that must be true for your program to work correctly.

Example:

```
def divide(a, b):
    assert b != 0, "The denominator cannot be zero."
    return a / b
```

Unit Testing: Write unit tests using the unittest or pytest framework to ensure that your code works as expected. Testing helps catch bugs early and improves code reliability.

Example:

```
import unittest

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

class TestMathFunctions(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(1, 2), 3)

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

Summary

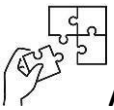
By following these best practices, you enhance the quality of your Python code:

- ✓ **Readability and Style**: Adhere to PEP 8 guidelines and use meaningful names and docstrings.
- ✓ **Use of Built-in Features**: Take advantage of Python's built-in functions and libraries for optimized performance.
- ✓ **Efficiency and Memory Usage**: Use generators and avoid unnecessary copies to manage memory effectively.
- ✓ **Error Handling and Testing**: Implement robust error handling and write unit tests to ensure code reliability.

Points to Remember

- Os used for interacting with the operating system and file system.
- Pathlib used for an object-oriented approach to path manipulation.
- Shutil used for high-level file operations such as copying and moving files.
- Pandas used for reading and writing data in various formats, primarily used for data analysis.
- Creating a New File: Use open (file_path, 'w') or open(file_path, 'x') to create a new file.
- Writing to an Existing File: Use open (file_path, 'a') to append or 'w' to overwrite.
- Removing a File: sUse os.remove () to delete a file.
- Deleting a Folder: Use os.rmdir () for empty directories and shutil.rmtree() for non-empty directories.

- Adhering to Python best practices can improve code quality, readability, and maintainability.
- Following PEP 8 guidelines promotes consistent coding style.
- Using built-in features can make code more concise and efficient.
- Writing efficient code can reduce resource consumption.
- Proper error handling can prevent unexpected program failures.
- Testing can help identify and fix bugs early in the development process.
- Regular code reviews can help improve coding practices and catch potential issues.



Application of learning 2.5.

Develop a python program that can create a file in excel format "list of TVET schools" and attach the following as header (District,School name,Trade,Number of students) and attach sample data in row 1 (Your_district, Your_schoolname, SWD,23). The created file have to be saved on desktop of your computer in directory works if that directory don't exist it has to create it before saving that file.

Learning outcome 2 end assessment

Theoretical assessment

I.Circle the letter that correspond to the right answer:

1. What is the correct syntax to create a list in Python?

A) list = (1, 2, 3)
B) list = [1, 2, 3]
C) list = {1, 2, 3}
D) list = <1, 2, 3>

2. Which of the following is an immutable data type in Python?
    A) List
    B) Dictionary
    C) Tuple
    D) Set
3. What will the output of the following code be?

```
x = 5
y = 10
print(x > y)
```

    A) True
    B) False
    C) 5
    D) 10
4. Which operator is used for exponentiation in Python?
    A) ^
    B) **
    C) //
    D) ***
5. Which keyword is used to define a function in Python?
    A) function
    B) def
    C) define
    D) func
6. What does the 'break' statement do in a loop?
    A) Skips the current iteration
    B) Exits the loop
    C) Repeats the loop
    D) Terminates the program
7. How do you create a dictionary in Python?
    A) dict = [key: value]
    B) dict = {key: value}

C) dict = (key: value)
D) dict = <key: value>

8. Which function is used to read a CSV file into a DataFrame using pandas?
   A) pd.read_csv()
   B) pd.load_csv()
   C) pd.import_csv()
   D) pd.open_csv()
9. What will len([1, 2, 3]) return?
   A) 2
   B) 3
   C) 4
   D) None
10. Which of the following is a built-in function in Python?
    A) print()
    B) show()
    C) display()
    D) output()
11. In Python, what data type is used to represent True or False values?
    A) int
    B) float
    C) bool
    D) str
12. What will the output of the following code?
    print("Hello, World!"[7])
    A) H
    B) e
    C) W
    D) o
13. Which of the following is NOT a valid way to comment in Python?
    A) # This is a comment
    B) /* This is a comment */
    C) """ This is a comment """
    D) #!
14. What does the 'continue' statement do in a loop?
    A) Exits the loop
    B) Skips to the next iteration
    C) Restarts the loop
    D) Ends the program
15. Which of the following is used to create a set in Python?
    A) []
    B) ()

C) {}
D) <>

16. What is the correct way to define a lambda function in Python?
    A) lambda x, y: x + y
    B) function x, y: x + y
    C) def x, y: x + y
    D) x, y -> x + y
17. What is the output of print(type(3.14))?
    A) <class 'int'>
    B) <class 'float'>
    C) <class 'str'>
    D) <class 'bool'>
18. Which of the following statements is used to import the os module?
    A) import os
    B) include os
    C) using os
    D) require os
19. What will the following code output?

```
x = 20
if x < 10:
print("Small")
else:
print("Large")
```

    A) Small
    B) Large
    C) 20
    D) None
20. Which method can be used to add an item to a list in Python?
    A) add()
    B) append()
    C) insert()
    D) Both B and C

II.Complete thefollowing statements by correct word, operator, or keword from the listed ones :

(function, ==, loop,lambda, os, defaultdict, break, tuple, open, dictionary)

1. A _____ is a block of reusable code that performs a specific task in Python.
2. The _____ operator is used to compare two values for equality.
3. In Python, a _____ allows you to iterate over a sequence.
4. A _____ function can take any number of arguments but can only have one expression.

5. The _____ module provides a way to use operating system-dependent functionality in Python.
6. A _____ is a dictionary subclass that provides a default value for a non-existent key.
7. The _____ statement is used to exit the nearest enclosing loop in Python.
8. A _____ is an immutable collection that can hold a variety of object types.
9. The _____ function is used to read the contents of a text file.
10. A _____ is a collection of key-value pairs where keys must be unique.

i. Respond to the followings by True or False

1. A list in Python is immutable.
2. A function in Python can return multiple values.
3. The 'else' clause can be used with a 'for' loop.
4. The pandas library is primarily used for file handling.
5. Sets in Python can contain duplicate elements.
6. The 'pass' statement in Python does nothing when executed.
7. You can use the 'with' statement for file handling in Python.
8. A frozen set is a mutable version of a set.
9. The 'elif' keyword is used to check multiple conditions in Python.
10. Variables in Python do not require a declaration before use.

III. Match the following Python data types with their corresponding descriptions:

ANSWER	Data Type	Description
……	A. List	1. Unordered collection of unique elements
……	B. Tuple	2. Ordered, mutable collection
……	C. Dictionary	3. Key-value pairs
……	D. Set	4. Ordered, immutable collection

Practical assessment

CODEX DEV LTD, a Kigali-based software development company, is seeking a full-stack developer to implement a shopping cart feature in their existing system. The developer will be responsible for receiving a list of items to be purchased, allowing users to remove items from the cart, and providing an option to empty the cart entirely. And app must store those cart information to file named cart.csv The project will be implemented using Python.

References

James, G., Wi en, D., Has e, T., & Tibshirani, R. (2017). An Introduction to Statistical Learning: with Applications in R. Springer

LazyProgrammer. (2016). Deep Learning: Recurrent Neural Networks in Python: LSTM, GRU, and more RNN machine learning architectures in Python and Theano (Machine Learning in Python).

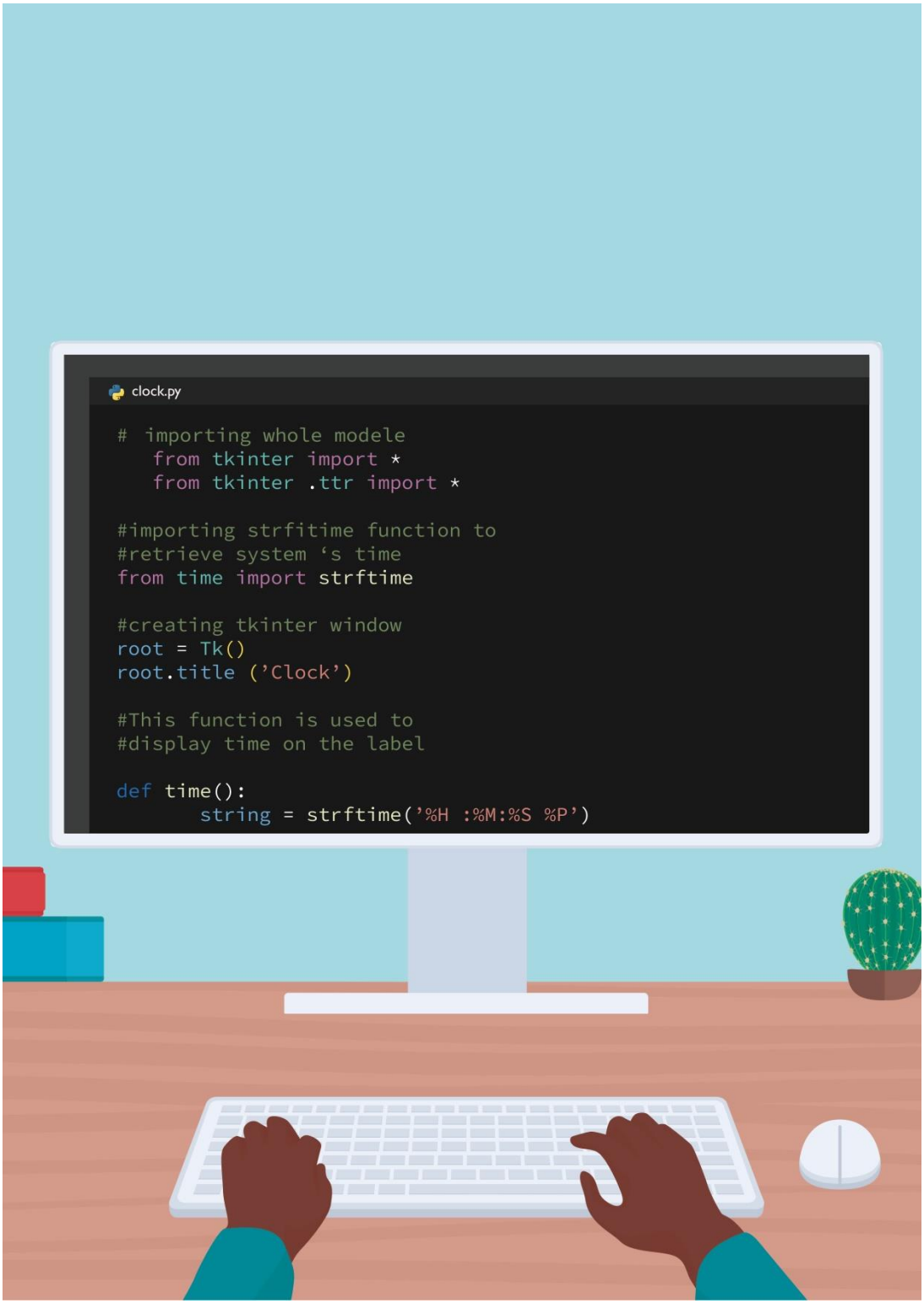
Mar n, K., Hi mana, E., Ngabonziza, J., Hanyurwimfura, D., Musabe, R., Uwamahoro, A., . . . Mutonga, K. (2023). Crop Yield Prediction Using Machine Learning Models: Case of Irish Potato and Maize. Agriculture, 20.

Moolayil, J. J. (2019). Learn Keras for Deep Neural Networks: A Fast-Track Approach to Modern Deep Learning with Python.

Morgan, P. (2018). Data Analysis from Scratch With Python: Beginner Guide, Pandas, NumPy, Scikit-Learn, IPython, TensorFlow, and Matplotlib.

Russell, R. (2018). Machine Learning: Step-by-Step Guide To Implement Machine Learning Algorithms with Python.

Sarkar, D., Raghav, B., & Tushar, S. (2017). Practical Machine Learning with Python: A Problem-Solver's Guide to Building Real-World Intelligent Systems.


clock.py
# importing whole modele
from tkinter import *
from tkinter .ttr import *
#importing strfitime function to
#retrieve system 's time
from time import strftime
#creating tkinter window
root = Tk()
root.title ('Clock')
#This function is used to
#display time on the label
def time():
string = strftime('%H :%M:%S %P')

Indicative contents
3.1 Applying OOP Concepts
3.2 Applying python Date and time concepts
3.3 Applying Python Libraries
3.4 Applying system Automation

Key Competencies for Learning Outcome 3: Apply object-driven in python

Knowledge	Skills	Attitudes
• Description of date and time • Description of Python Library • Understanding Scope of library according to the name space • Identification of tasks to automate • Identification of tasks to be prioritized	• Applying OOP Concepts • Setting Time zones • Formatting and parsing dae and time • Performing relative timedeltas • Using python libraries • Selecting Python Automation Library • Developing Python Script • Integrating script with Deployment Process • Testing and Monitoring the automated task	• Having teamwork spirit ability • Being critical thinker • Being innovative • Being attentive • Being creative • Problem solving • Being practical oriented

Duration: 20 hrs

Learning outcome 3 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly date and time as according to python standards
2. Describe correctly Python Library in accordance with python standards
3. Understand correctly Scope of library according to the name space
4. Identify clearly tasks to automate based on specific task
5. Identify correctly tasks to be prioritized based on specific task
6. Apply correctly OOP Concepts in line with python standards
7. Set properly Time zones in line with python standards
8. Use correctly python libraries in accordance with python standards
9. Select correctly Python Automation Library based on specific task
10. Develop correctly Python Script based on specific task to be automated
11. Integrate properly script with Deployment Process based on specific task
12. Test and monitor correctly automated task in accordance with python standards

Resources

Equipment	Tools	Materials
● Computer	● Python (latest and stable version) ● IDE (Jupyter notebook)	● Internet

Indicative content 3.1: Applying OOP Concepts

Duration: 5 hrs

Theoretical Activity 3.1.1: Description of OOP concepts

Tasks:

1: You are requested to answer the following questions related to the introduction to Object Oriented Programming:

Describe the following terms as applied in OOP:

i. Object
ii. Python class
iii. Inheritance
iv. Polymorphism
v. Encapsulation

2: Write your findings on paper/flipchart

3: Present your findings to the whole class or trainer

4: For more clarification read key reading 3.1.1 and ask questions where necessary.

Key readings 3.1.1. Description of OOP concepts

1. Object

Python is an object-oriented programming language.

Almost everything in Python is an object, with its properties and methods.

2. Python class

A Class is like an object constructor, or a "blueprint" for creating objects.

To create a class, use the keyword class:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

3. Inheritance

3.1. Definition

Inheritance in Python is a fundamental concept in object-oriented programming that allows a class (known as a subclass or derived class) to inherit attributes and methods from another class (known as a superclass or base class).

This mechanism promotes code reuse and establishes a natural hierarchy between classes.

Inheritance in Python is a fundamental concept of object-oriented programming (OOP) that allows a new class (child class) to inherit the properties and behaviors (attributes and methods) from an existing class (parent class).

This promotes code reusability and hierarchy, as the child class can extend or override the functionalities of the parent class. In Python, inheritance is defined by passing the parent class as an argument to the child class.

There are several types of inheritance, such as single, multiple, multilevel, and hierarchical inheritance.

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

3.2. Key Features of Inheritance

- **Code Reusability:** Inheritance allows subclasses to use methods and properties of the superclass, reducing redundancy.
- **Method Overriding:** A subclass can provide a specific implementation of a method that is already defined in its superclass.
- **Multiple Inheritance:** Python supports multiple inheritance, allowing a subclass to inherit from more than one superclass.
- **Types of Inheritance**
- **Single Inheritance:** A subclass inherits from one superclass.

```
class Animal:
    def speak(self):
        return "Animal speaks"

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

dog = Dog()
print(dog.speak())  # Output: Bark
```

Multiple Inheritance: A subclass inherits from multiple superclasses.

```
class Flyer:
def fly(self):
 return "Flying"

class Swimmer:
def swim(self):
return "Swimming"

class Duck(Flyer, Swimmer):
pass

duck = Duck()
print(duck.fly())  # Output: Flying
print(duck.swim())  # Output: Swimming
```

Multilevel Inheritance: A class inherits from a subclass, creating a chain of inheritance.

```
class Animal:
def speak(self):
return "Animal speaks"

class Dog(Animal):
def bark(self):
return "Bark"
```

```
class Puppy(Dog):
    def weep(self):
        return "Wee wee"

puppy = Puppy()
print(puppy.speak())  # Output: Animal speaks
print(puppy.bark())   # Output: Bark
```

Hierarchical Inheritance: Multiple subclasses inherit from a single superclass.

```
class Shape:
    def area(self):
        return "Area calculation"

class Circle(Shape):
    def area(self):
        return "Area of Circle"

class Square(Shape):
    def area(self):
        return "Area of Square"

circle = Circle()
square = Square()
print(circle.area())  # Output: Area of Circle
print(square.area())  # Output: Area of Square
```

4. Polymorphism

4.1. Definition

Polymorphism in Python is a core concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables methods to do different things based on the object it is acting upon, even if they share the same name.

4.2. Types of Polymorphism

Method Overriding: A subclass can provide a specific implementation of a method that is already defined in its superclass.

```
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
```

```
    def sound(self):
        return "Bark"

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

def make_sound(animal):
    print(animal.sound())

make_sound(Dog())  # Output: Bark
make_sound(Cat())  # Output: Meow
```

4.3. Duck Typing:

In Python, the type or class of an object is less important than the methods it defines. If an object behaves like a certain type (i.e., has the necessary methods), it can be used as that type.

```
class Bird:
    def fly(self):
        return "Flying"

class Airplane:
    def fly(self):
        return "Jetting through the sky"

def let_it_fly(flyable):
    print(flyable.fly())

let_it_fly(Bird())     # Output: Flying
let_it_fly(Airplane())  # Output: Jetting through the sky
```

4.4. Benefits of Polymorphism

- **Flexibility:** Functions or methods can operate on objects of different types.
- **Code Reusability:** Common interfaces can be used across different classes.
- **Simplification:** Code becomes easier to read and maintain by using a consistent interface.

5. Encapsulation

5.1. Definition

Encapsulation in Python is an object-oriented programming principle that restricts

direct access to certain components of an object. This is done to protect the internal state of the object and to control how data is accessed and modified. Encapsulation promotes modularity and helps maintain the integrity of the data within an object.

5.2. Key Features of Encapsulation

- **Data Hiding:** By restricting access to some of an object's attributes, encapsulation helps prevent unintended interference and misuse of the data. This is typically achieved using private or protected access modifiers.
- **Controlled Access:** Encapsulation allows the use of getter and setter methods to access and modify private attributes, enabling validation and control over how data is manipulated.
- **Improved Maintainability:** With encapsulation, changes to the internal implementation of a class can be made without affecting external code that relies on it.
- **Implementation of Encapsulation in Python**
- In Python, encapsulation is implemented through naming conventions:
- **Public Attributes:** Accessible from outside the class.
- **Protected Attributes:** Indicated by a single underscore (_), suggesting that they should not be accessed directly outside the class.
- **Private Attributes:** Indicated by a double underscore (__), making them harder to access from outside the class.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner          # Public attribute
        self.__balance = balance    # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew: {amount}")
        else:
            print("Insufficient funds or invalid amount.")
```

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

# Usage
account = BankAccount("Alice", 100)
account.deposit(50)           # Deposited: 50
print(account.get_balance())    # Output: 150
account.withdraw(30)          # Withdrew: 30
print(account.get_balance())    # Output: 120

# Attempting direct access to the private attribute
# print(account.__balance)      # AttributeError
```

Practical Activity 3.1.2: Applying classes and objects

Task:

1. Read key reading 3.1.2

As a full stack developer, you have been asked to develop a python program to perform simple calculation (add, substract, divide and multiply). That program have to let numbers to be entered by user using keyboard, and after entering numbers it have to let user to select the operation to be performed and display the results according to selected operation.

2. Apply safety precautions.
3. Referring to the steps provided in key readings 3.1.2, develop the required program.
4. Present out the steps to be followed.

Key readings 3.1.2 Application of classes and objects

1. Introduction

Classes and objects in Python enable developers to model real-world entities and their behavior. This object-oriented approach promotes code organization, reusability, and abstraction.

2. Application of classes and object

Here are some common applications of classes and objects, along with examples.

2.1. Modeling Real-World Entities

Classes can represent real-world entities with attributes and methods.

Example: **Car Class**

```
class Car:
 def __init__(self, make, model, year):
self.make = make
self.model = model
self.year = year
def display_info(self):
return f"{self.year} {self.make} {self.model}"
# Creating an object of the Car class
my_car = Car("Toyota", "Corolla", 2020)
print(my_car.display_info())  # Output: 2020 Toyota Corolla
```

2.2. Encapsulation

Classes can encapsulate data and provide controlled access through methods.

Example: Bank Account

```
class BankAccount:
def __init__(self, owner, balance=0):
self.owner = owner
self.__balance = balance  # Private attribute
def deposit(self, amount):
if amount > 0:
self.__balance += amount
def withdraw(self, amount):
if 0 < amount <= self.__balance:
self.__balance -= amount
def get_balance(self):
return self.__balance
# Creating a bank account object
account = BankAccount("Alice", 100)
account.deposit(50)
print(account.get_balance())  # Output: 150
```

2.3. Inheritance

Classes can inherit attributes and methods from other classes, promoting code reuse.

Example: Employee and Manager Classes

```
class Employee:
 def __init__(self, name, salary):
self.name = name
self.salary = salary
```

```
def display_info(self):
 return f"Employee: {self.name}, Salary: {self.salary}"
class Manager(Employee):
def __init__(self, name, salary, department):
 super().__init__(name, salary)
self.department = department
def display_info(self):
return f"Manager: {self.name}, Salary: {self.salary}, Department: {self.department}"
# Creating objects
emp = Employee("John", 50000)
mgr = Manager("Alice", 70000, "HR")
print(emp.display_info())  # Output: Employee: John, Salary: 50000
print(mgr.display_info())  # Output: Manager: Alice, Salary: 70000, Department: HR
```

2.4. Polymorphism

Classes can define methods with the same name, allowing different behaviors based on the object type.

Example: Shape Class

```
class Shape:
def area(self):
pass
class Rectangle(Shape):
def __init__(self, width, height):
self.width = width
self.height = height
def area(self):
eturn self.width * self.height
class Circle(Shape):
def __init__(self, radius):
self.radius = radius
def area(self):
 return 3.14 * (self.radius ** 2)
# Function to calculate area
def print_area(shape):
 print(f"Area: {shape.area()}")
# Creating objects
rect = Rectangle(10, 5)
circle = Circle(7)
```

```
print_area(rect)   # Output: Area: 50
print_area(circle) # Output: Area: 153.86
```

2.5. Creating Frameworks and Libraries

Classes allow for the creation of modular code that can be reused across different projects.

Example: Simple Web Framework

```
class WebApp:
def __init__(self, name):
self.name = name
 self.routes = {}
def route(self, path):
def wrapper(func):
self.routes[path] = func
return func
return wrapper
def run(self):
for path, func in self.routes.items():
print(f"Route: {path}, Response: {func()}")
# Creating a web app object
app = WebApp("MyApp")
@app.route("/")
def home():
return "Welcome to the homepage!"

@app.route("/about")
def about():
  return "This is the about page."
# Running the web app
app.run()
# Output:
# Route: /, Response: Welcome to the homepage!
# Route: /about, Response: This is the about page.
```

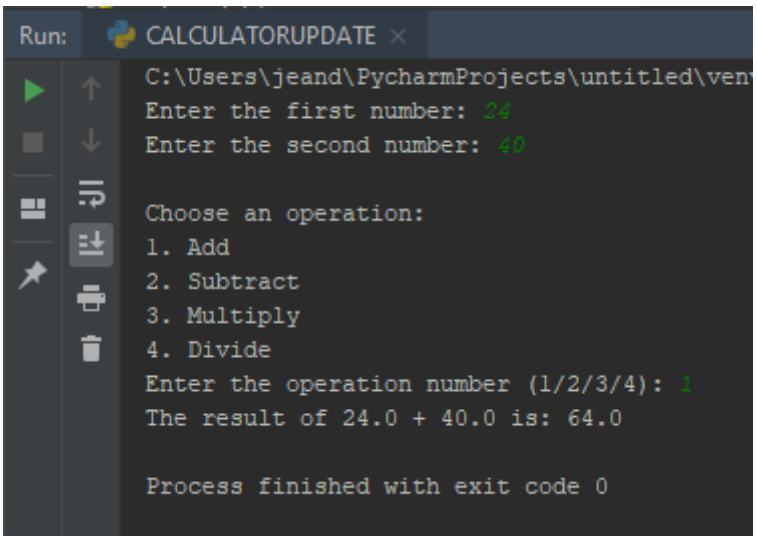
Sample program of simple calculator in python that can perform simple calculation:

```
class Calculator:
  def add(self, a, b):
    return a + b

  def subtract(self, a, b):
    return a - b
```

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

    def divide(self, a, b):
        if b == 0:
            return "Cannot divide by zero!"
        return a / b

# User input for numbers
a = float(input("Enter the first number: "))
b = float(input("Enter the second number: "))

# Creating an object of the Calculator class
calc = Calculator()

# Display possible operations
print("\nChoose an operation:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")

# User chooses an operation
operation = input("Enter the operation number (1/2/3/4): ")

# Perform the chosen operation and display the result
if operation == '1':
    result = calc.add(a, b)
    print(f"The result of {a} + {b} is: {result}")
elif operation == '2':
    result = calc.subtract(a, b)
    print(f"The result of {a} - {b} is: {result}")
elif operation == '3':
    result = calc.multiply(a, b)
    print(f"The result of {a} * {b} is: {result}")
elif operation == '4':
    result = calc.divide(a, b)
    print(f"The result of {a} / {b} is: {result}")
else:
```

```
print("Invalid operation. Please enter a number between 1 and 4.")
```

The output of that program



Practical Activity 3.1.3: Applying inheritance in python

Task:

1. Read key reading 3.1.3
2. As a full stack developer, you have been asked to develop a python program to perform bank transactions including deposit and withdraw, for withdrow you cannot let the account to be empty means it have to let 100 Rfw on the account and for all transactions it have to display the messages.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, develop the required program.
5. Present out the steps to be followed.

Key readings 3.1.3: Applying inheritance in python

Application of inheritance in python

Inheritance in Python is a powerful feature that allows one class (the subclass) to inherit attributes and methods from another class (the superclass). This promotes code reuse, simplifies maintenance, and creates a clear hierarchical relationship between classes.

Here are some common applications of inheritance in Python, along with examples.

1. Code Reusability

Inheritance enables subclasses to use existing code from superclasses, reducing redundancy.

Example: Basic Shapes

```
class Shape:
def area(self):
 pass
class Rectangle(Shape):
def __init__(self, width, height):
self.width = width
 self.height = height
def area(self):
return self.width * self.height
class Circle(Shape):
def __init__(self, radius):
self.radius = radius
def area(self):
return 3.14 * (self.radius ** 2)
# Using the classes
shapes = [Rectangle(10, 5), Circle(7)]
for shape in shapes:
print(f"Area: {shape.area()}")
```

2. Method Overriding

Subclasses can provide specific implementations of methods defined in their superclasses.

Example: Employees

```
class Employee:
def calculate_salary(self):
return 50000
class Manager(Employee):
def calculate_salary(self):
return super().calculate_salary() + 20000  # Base salary + bonus
class Developer(Employee):
def calculate_salary(self):
return super().calculate_salary() + 10000  # Base salary + bonus
# Using the classes
employees = [Manager(), Developer()]
for emp in employees:
print(f"Salary: {emp.calculate_salary()}")
```

3. Creating a Hierarchical Structure

Inheritance allows the creation of a hierarchy where subclasses can inherit from a common superclass.

Example: Animal Kingdom

```
class Animal:
def speak(self):
return "Some sound"
class Dog(Animal):
def speak(self):
 return "Bark"
class Cat(Animal):
def speak(self):
return "Meow"
# Using the classes
animals = [Dog(), Cat()]
for animal in animals:
print(animal.speak())
```

4. Multiple Inheritance

Python supports multiple inheritance, allowing a subclass to inherit from multiple superclasses.

Example: **Vehicles**

```
class Flyer:
def fly(self):
return "Flying"
class Swimmer:
def swim(self):
return "Swimming"
class Duck(Flyer, Swimmer):
def quack(self):
return "Quack!"
# Using the class
duck = Duck()
print(duck.fly())   # Output: Flying
print(duck.swim())  # Output: Swimming
print(duck.quack()) # Output: Quack!
```

5. Framework and Library Development

Inheritance allows for the creation of extensible frameworks where users can

subclass base classes to implement specific functionality.

Example: GUI Framework

```
class Widget:
def draw(self):
pass
class Button(Widget):
def draw(self):
return "Drawing a button"
class TextBox(Widget):
def draw(self):
return "Drawing a text box"
# Using the classes
widgets = [Button(), TextBox()]
for widget in widgets:
print(widget.draw())
```

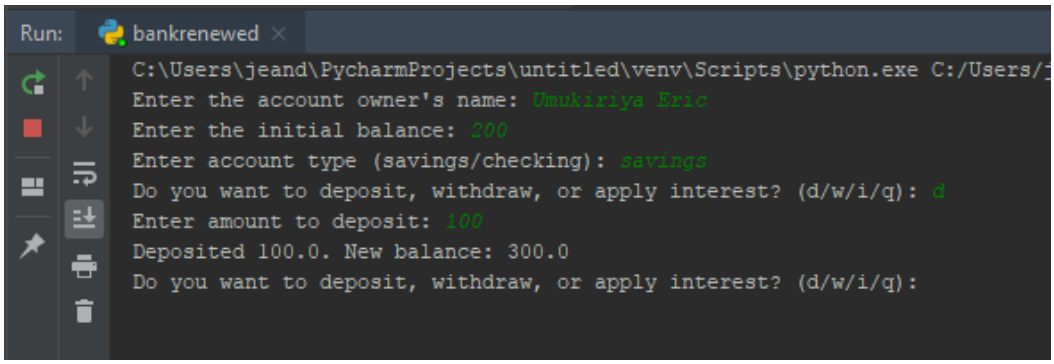
Python program for bank account management

```
# Base class for a bank account
class BankAccount:
def __init__(self, owner, balance=0):
# Initialize the account with owner and starting balance
self.owner = owner
self.balance = balance

def deposit(self, amount):
  # Add amount to the balance
self.balance += amount
return f"Deposited {amount}. New balance: {self.balance}"
def withdraw(self, amount):
# Calculate the maximum amount that can be withdrawn
if self.balance - amount < 100:
max_withdrawable = self.balance - 100
return f"You can only withdraw up to {max_withdrawable}."
self.balance -= amount
return f"Withdrew {amount}. New balance: {self.balance}"
# Derived class for a savings account
class SavingsAccount(BankAccount):
def __init__(self, owner, balance=0, interest_rate=0.02):
# Initialize the savings account with owner, balance, and interest rate
super().__init__(owner, balance)  # Call the constructor of the base
```

```
class
      self.interest_rate = interest_rate

def apply_interest(self):
# Apply interest to the current balance
interest = self.balance * self.interest_rate
self.balance += interest
return f"Interest applied. New balance: {self.balance}"

# Derived class for a checking account
class CheckingAccount(BankAccount):
def __init__(self, owner, balance=0, overdraft_limit=100):
 # Initialize the checking account with owner, balance, and overdraft
limit
 super().__init__(owner, balance)
self.overdraft_limit = overdraft_limit

def withdraw(self, amount):
 # Calculate the maximum amount that can be withdrawn
if self.balance - amount < 100:
 max_withdrawable = self.balance - 100
return f"You can only withdraw up to {max_withdrawable}."
# Allow withdrawal up to the overdraft limit
if amount > self.balance + self.overdraft_limit:
return "Insufficient funds, even with overdraft."
self.balance -= amount
return f"Withdrew {amount}. New balance: {self.balance}"

# User input for account details
owner = input("Enter the account owner's name: ")
initial_balance = float(input("Enter the initial balance: "))

# Ensure the initial balance is above 100
while initial_balance <= 100:
print("Initial balance must be greater than 100.")
initial_balance = float(input("Enter the initial balance: "))

# Choose account type
account_type = input("Enter account type (savings/checking):
").lower()
```

```
# Creating an object of the appropriate account class
if account_type == 'savings':
account = SavingsAccount(owner, initial_balance)
elif account_type == 'checking':
account = CheckingAccount(owner, initial_balance)
else:
print("Invalid account type. Defaulting to BankAccount.")
account = BankAccount(owner, initial_balance)

# User operations
while True:
action = input("Do you want to deposit, withdraw, or apply interest?
(d/w/i/q): ").lower()
if action == 'd':
amount = float(input("Enter amount to deposit: "))
  print(account.deposit(amount))
elif action == 'w':
amount = float(input("Enter amount to withdraw: "))
print(account.withdraw(amount))
elif action == 'i' and isinstance(account, SavingsAccount):
print(account.apply_interest())
elif action == 'q':
break
else:
print("Invalid option. Please try again.")
```

Output of the program



Practical Activity 3.1.4: Applying polymorphism in python

Task:

1. Read key reading 3.1.4
2. As a full stack developer, you have been asked to develop a python program to perform bank transactions including deposit and withdraw, for withdrow you cannot let the account to be empty means it have to let 100 Rfw on the account and for all transactions it have to display the messages the program have to include the application of polymorphism.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, develop the required program.
5. Present out the steps to be followed.

Key readings 3.1.4 Applying polymorphism in python

1. Introduction

Polymorphism in Python is a powerful feature that allows different classes to be treated as instances of the same class through a common interface. It enables methods to be defined in multiple forms, allowing for flexible and interchangeable code.

2. common applications of polymorphism in Python

Here are some common applications of polymorphism in Python, along with examples.

2.1. Method Overriding

Polymorphism allows subclasses to provide specific implementations of methods defined in their superclasses.

Example: Shape Classes

```
class Shape:
def area(self):
pass
class Rectangle(Shape):
def __init__(self, width, height):
self.width = width
self.height = height

def area(self):
 return self.width * self.height
class Circle(Shape):
def __init__(self, radius):
```

```
    self.radius = radius

    def area(self):
    return 3.14 * (self.radius ** 2)

    # Using polymorphism
    shapes = [Rectangle(10, 5), Circle(7)]
    for shape in shapes:
    print(f"Area: {shape.area()}")
```

2.2. Duck Typing

Python's dynamic typing allows objects to be used based on their behavior rather than their actual type, a concept known as duck typing.

Example: Flying Objects

```
class Bird:
def fly(self):
return "Flapping wings"

class Airplane:
def fly(self):
return "Jetting through the sky"
def let_it_fly(flyable):
print(flyable.fly())
# Using duck typing
let_it_fly(Bird()) # Output: Flapping wings
let_it_fly(Airplane())  # Output: Jetting through the sky
```

2.3. Operator Overloading

Polymorphism can also be achieved through operator overloading, allowing different types of objects to interact using standard operators.

Example: Custom Vector Class

```
class Vector:
def __init__(self, x, y):
self.x = x
self.y = y
def __add__(self, other):
return Vector(self.x + other.x, self.y + other.y)
def __repr__(self):
return f"Vector({self.x}, {self.y})"
# Using the custom Vector class
```

```
v1 = Vector(2, 3)
v2 = Vector(5, 7)
result = v1 + v2
print(result)  # Output: Vector(7, 10)
```

2.4. Function Overloading

Although Python doesn't natively support function overloading, polymorphism allows functions to accept different types of arguments.

Example: Print Function

```
def print_info(data):
if isinstance(data, str):
 print(f"String: {data}")
elif isinstance(data, int):
 print(f"Integer: {data}")
elif isinstance(data, list):
print(f"List: {data}")
# Using the function
print_info("Hello")# Output: String: Hello
print_info(42)# Output: Integer: 42
print_info([1, 2, 3])    # Output: List: [1, 2, 3]
```

2.5. Frameworks and Libraries

Polymorphism is widely used in frameworks and libraries, allowing for flexible design patterns where components can be easily interchanged.

Example: GUI Framework

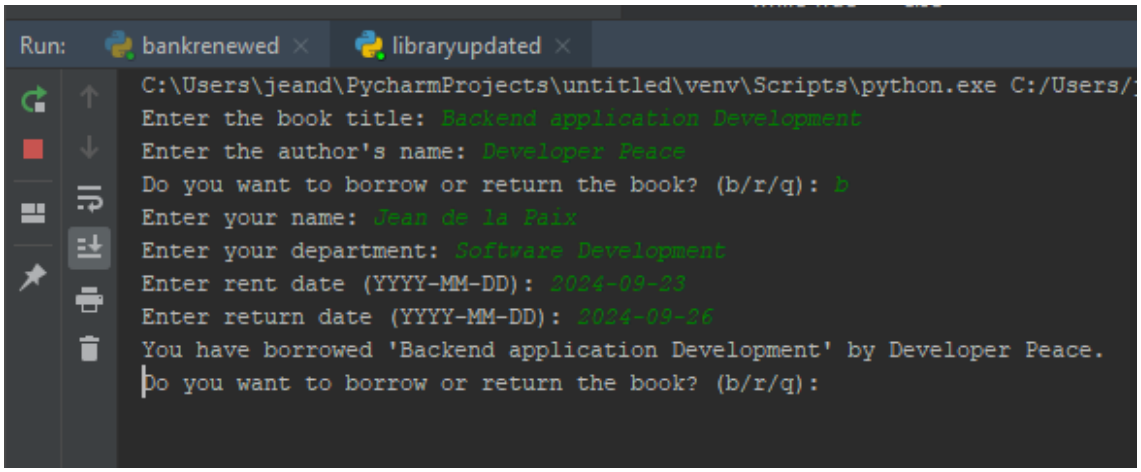
```
class Widget:
    def draw(self):
        pass
class Button(Widget):
    def draw(self):
        return "Drawing a button"
class TextBox(Widget):
    def draw(self):
        return "Drawing a text box"
# Using the GUI components
widgets = [Button(), TextBox()]
for widget in widgets:
    print(widget.draw())
```

Example sample library program

```
from datetime import datetime
class Book:
def __init__(self, title, author):
self.title = title
self.author = author
self.is_borrowed = False
self.borrower_name = None
self.department = None
self.rent_date = None
self.return_date = None
def borrow(self, borrower_name, department, rent_date, return_date):
if not self.is_borrowed:
 self.is_borrowed = True
self.borrower_name = borrower_name
self.department = department
self.rent_date = rent_date
self.return_date = return_date
return f"You have borrowed '{self.title}' by {self.author}."
return f"'{self.title}' is already borrowed."
def return_book(self):
if self.is_borrowed:
self.is_borrowed = False
details = f"Returned '{self.title}' by {self.author}."
self.borrower_name = None
self.department = None
self.rent_date = None
self.return_date = None
return details
return f"'{self.title}' was not borrowed."
def get_borrow_info(self):
if self.is_borrowed:
return (f"Borrower: {self.borrower_name}, "
f"Department: {self.department}, "
f"Rent Date: {self.rent_date}, "
f"Return Date: {self.return_date}")
return "This book is not currently borrowed."
# User input for book details
title = input("Enter the book title: ")
author = input("Enter the author's name: ")
```

```
# Creating an object of the Book class
book1 = Book(title, author)
# User actions
while True:
action = input("Do you want to borrow or return the book? (b/r/q): ").lower()
if action == 'b':
borrower_name = input("Enter your name: ")
department = input("Enter your department: ")
rent_date = input("Enter rent date (YYYY-MM-DD): ")
return_date = input("Enter return date (YYYY-MM-DD): ")
# Validating date format
try:
rent_date = datetime.strptime(rent_date, '%Y-%m-%d').date()
return_date = datetime.strptime(return_date, '%Y-%m-%d').date()
print(book1.borrow(borrower_name, department, rent_date, return_date))
except ValueError:
print("Invalid date format. Please enter dates in YYYY-MM-DD format.")
elif action == 'r':
print(book1.return_book())
elif action == 'info':
print(book1.get_borrow_info())
elif action == 'q':
break
else:
print("Invalid option. Please try again.")
```

Program output



Practical Activity 3.1.5: Applying Encapsulation in python

Task:

1. Read key reading 3.1.5
2. As a full stack developer, you have been asked to develop a python program to perform simple calculation (add, substract, divide and multiply). That program have to let numbers to be entered by user using keyboard, and after entering numbers it have to let user to select the operation to be performed and display the results according to selected operation. In addition, the program have to show the application of encapsulation
3. Apply safety precautions.
4. Referring to the steps provided in key readings, develop the required program.
5. Present out the steps to be followed.

Key readings 3.1.5 Applying Encapsulation in python

1. Introduction

Encapsulation in Python is a fundamental object-oriented programming principle that restricts direct access to certain attributes and methods of a class. This approach is used to protect the internal state of an object and to control how data is accessed and modified.

2. Common applications of encapsulation in Python

Here are some common applications of encapsulation in Python, along with examples.

2.1. Data Hiding

Encapsulation helps in hiding the internal state of an object from the outside world, ensuring that only specified methods can modify it. This prevents unintended interference and misuse.

Example: Bank Account

```
class BankAccount:
def __init__(self, owner, balance=0):
self.owner = owner
self.__balance = balance  # Private attribute
def deposit(self, amount):
if amount > 0:
self.__balance += amount
else:
```

```
print("Deposit amount must be positive.")

 def withdraw(self, amount):
if 0 < amount <= self.__balance:
self.__balance -= amount
else:
print("Insufficient funds or invalid amount.")
def get_balance(self):
return self.__balance
# Usage
account = BankAccount("Alice", 100)
account.deposit(50)
print(account.get_balance())  # Output: 150
account.withdraw(30)
print(account.get_balance())  # Output: 120
```

2.2. Controlled Access

Encapsulation allows the use of getter and setter methods to control access to private attributes. This enables validation and ensures that the internal state remains valid.

Example: Employee Class

```
class Employee:
def __init__(self, name, salary):
self.__name = name
self.__salary = salary
def set_salary(self, salary):
if salary < 0:
print("Salary must be positive.")
else:
self.__salary = salary

def get_salary(self):
return self.__salary
# Usage
emp = Employee("John", 50000)
print(emp.get_salary())  # Output: 50000
emp.set_salary(-1000)     # Output: Salary must be positive.
emp.set_salary(60000)
print(emp.get_salary())  # Output: 60000
```

2.3. Implementation Hiding

Encapsulation allows for changes in the internal implementation of a class without affecting external code that relies on it. This makes the code easier to maintain and evolve.

Example: Temperature Converter

```
class Temperature:
def __init__(self, celsius):
elf.__celsius = celsius  # Private attribute
def to_fahrenheit(self):
return (self.__celsius * 9/5) + 32
def to_kelvin(self):
return self.__celsius + 273.15
# Usage
temp = Temperature(25)
print(temp.to_fahrenheit())  # Output: 77.0
print(temp.to_kelvin())      # Output: 298.15
```

2.4. API Design

Encapsulation is vital in designing APIs, where you can expose a clean interface while hiding the underlying complexity.

Example: Simple Game

```
class Game:
def __init__(self):
self.__score = 0  # Private attribute
def increase_score(self, points):
if points > 0:
self.__score += points
 def get_score(self):
return self.__score
# Usage
game = Game()
game.increase_score(10)
print(game.get_score())  # Output: 10
```

2.5. Security

Encapsulation enhances security by restricting access to sensitive data. This is particularly important in applications handling confidential information.

Example: User Credentials

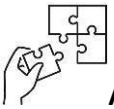
```
class User:
   def __init__(self, username, password):
```

```
    self.username = username
    self.__password = password  # Private attribute
    def authenticate(self, password):
    return self.__password == password
    # Usage
    user = User("Alice", "securepassword")
    print(user.authenticate("securepassword"))  # Output: True
    print(user.authenticate("wrongpassword"))    # Output: False
```

Points to Remember

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes that encapsulate data and behaviors.
- Classes serve as blueprints for creating objects, defining attributes and methods.
- Inheritance allows one class (subclass) to inherit properties and methods from another class (superclass), promoting code reuse and establishing a hierarchical relationship.
- Polymorphism enables objects of different classes to be treated as instances of a common superclass, allowing methods to be defined in multiple forms.
- Encapsulation restricts direct access to an object's internal state, exposing only necessary components through public methods, thereby enhancing data protection and modularity.
- Together, these concepts form the foundation of OOP, facilitating organized, reusable, and maintainable code.
- Classes and objects in Python provide a powerful way to model complex systems, encapsulate data, and promote code reuse. From simple data structures to complex frameworks, object-oriented programming enhances the clarity and maintainability of code.
- Classes and object in python are most applicable in different ways like the followings:In development of calculators programs, In banking transaction programs and In library management programs
- Inheritance in Python is a powerful feature that allows one class (the subclass) to inherit attributes and methods from another class (the superclass).
- Common applications of inheritance in Python: Code Reusability, Method Overriding, Creating a Hierarchical Structure, Multiple Inheritance and Framework and Library Development.

- Polymorphism in Python is a powerful feature that allows different classes to be treated as instances of the same class through a common interface.
- Polymorphism can be applied in different ways including: Method Overriding, Duck Typing, Operator Overloading, Function Overloading and Frameworks and Libraries
- Encapsulation in Python is a fundamental object-oriented programming principle that restricts direct access to certain attributes and methods of a class.
- Encapsulation can be used in different ways including: Data Hiding, Controlled Access, Implementation Hiding, API Design and Security



Application of learning .3.1.:

You are requested to develop a simple bank account transaction python program where there is application of inheritance in which the system have to accept to input the account owner and choose the action to be performed including withdraw, deposit and the system has to show the amount to be withdrow base on settings of bank.
The program has to include the followings:

- ✓ Object
- ✓ Class
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Encapsulation

Indicative content 3.2: Applying python Date and Time Concepts

Duration: 5 hrs

Theoretical Activity 3.2.1: Description of date and time

Tasks:

1. You are requested to describe the following date and time concepts in python programming:
    i. Datetime
    ii. Dateutil
    iii. Arrow
    iv. Pendulum
    v. Python-tzdata
2. Write your findings on paper/flipchart
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 3.2.1 and ask questions where necessary.

Key readings 3.2.1.: Description of date and time

1. Datetime

The datetime module is part of Python's standard library and provides classes for manipulating dates and times.

Key Classes

datetime: Combines date and time into a single object.
date: Represents a date (year, month, day).
time: Represents a time (hour, minute, second, microsecond).
timedelta: Represents the difference between two dates or times.

Example:

```
from datetime import datetime, timedelta
# Current date and time
now = datetime.now()
print("Current date and time:", now)
# Creating a specific date
new_year = datetime(2024, 1, 1)
print("New Year:", new_year)
# Date arithmetic
```

```
tomorrow = now + timedelta(days=1)
print("Tomorrow:", tomorrow)
```

2. Dateutil

dateutil is a powerful extension to the standard datetime module. It provides additional features for parsing, formatting, and manipulating dates and times.

Key Features

- ✓ Powerful parsing of dates using parse().
- ✓ Support for relative deltas (e.g., adding months, years).
- ✓ Time zone handling.

Example:

```
from dateutil import parser, relativedelta
from datetime import datetime
# Parsing a date string
date_str = "2024-01-01T12:00:00"
parsed_date = parser.parse(date_str)
print("Parsed Date:", parsed_date)
# Adding months
new_date = parsed_date + relativedelta.relativedelta(months=1)
print("One month later:", new_date)
```

3. Arrow

Arrow is a library that provides a more intuitive way to work with dates and times. It is designed for better readability and easier manipulation.

Key Features

- ✓ Easy timezone handling.
- ✓ Human-friendly formatting.
- ✓ Natural language support.

Example:

```
import arrow
# Current time
now = arrow.now()
print("Current time:", now)
# Shift time
shifted = now.shift(days=3)
print("Three days later:", shifted)
# Formatting
formatted = now.format('YYYY-MM-DD HH:mm:ss')
print("Formatted:", formatted)
```

4. Pendulum

Pendulum is another library for date and time manipulation that focuses on simplicity and flexibility. It includes features for working with time zones, durations, and intervals.

Key Features:

- ✓ UTC and local time zone support.
- ✓ Duration and period handling.
- ✓ Human-friendly methods for manipulating dates.

Example:

```
import pendulum
# Current time
now = pendulum.now()
print("Current time:", now)
# Adding time
future = now.add(days=7)
print("One week later:", future)
# Time zone support
utc_time = pendulum.now('UTC')
local_time = utc_time.in_tz('America/New_York')
print("Local time:", local_time)
```

5. Python-tzdata

python-tzdata is a library that provides the IANA Time Zone Database, making it easier to handle time zones in Python applications. It is often used in combination with datetime or other date libraries.

Key Features:

- ✓ Access to time zone data for accurate conversions.
- ✓ Works well with pytz for timezone-aware datetime objects.

Example:

```
import pytz
from datetime import datetime
# Time zone conversion
utc_zone = pytz.utc
new_york_zone = pytz.timezone('America/New_York')
# Current time in UTC
utc_time = datetime.now(utc_zone)
print("Current UTC time:", utc_time)
# Convert to New York time
ny_time = utc_time.astimezone(new_york_zone)
print("New York time:", ny_time)
```

Practical Activity 3.2.2: Applying python Date and time concepts

Task:

1. Read the task bellow
   As a full stack developer, you are asked to go to the computer lab apply date and time concepts while performing different operations on date and time calculating the difference between date, calculating the age depending on current date and date of birth.
2. Refers to provided key reading 3.2.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 3.2.2 Applying python Date and time concepts

1. Python Dates

A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

Example

```
Import the datetime module and display the current date:
import datetime
x = datetime.datetime.now()
print(x)
```

1.1. Date Output

When we execute the code from the example above the result will be:

2024-09-10 12:25:43.517354

The date contains year, month, day, hour, minute, second, and microsecond.

The datetime module has many methods to return information about the date object.

Example

Return the year and name of weekday:

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

1.2. Creating Date Objects

To create a date, we can use the datetime() class (constructor) of the datetime module.

The datetime() class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

The strftime() Method

The datetime object has a method for formatting date objects into readable strings. The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

A reference of all the legal format codes:

Directive	**Description**	**Example code**	**Output**
%a	Weekday, short version	import datetime x =datetime.datetime.now() print(x.strftime("%a"))	Tuey
%A	Weekday, full version	import datetime x = datetime.datetime.now() print(x.strftime("%A"))	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	import datetime x = datetime.datetime.now() print(x.strftime("%w"))	2
%d	Day of month 01-31	import datetime x = datetime.datetime.now() print(x.strftime("%d"))	23
%b	Month name, short version	import datetime x = datetime.datetime.now() print(x.strftime("%b"))	Dec
%B	Month	import datetime	Decemb

	name, full version	x = datetime.datetime.now() print(x.strftime("%B"))	er
%m	Month as a number 01-12	import datetime x = datetime.datetime.now() print(x.strftime("%m"))	12
%y	Year, short version, without century	import datetime x = datetime.datetime.now() print(x.strftime("%y"))	18
%Y	Year, full version	import datetime x = datetime.datetime.now() print(x.strftime("%Y"))	2018
%H	Hour 00-23	import datetime x = datetime.datetime.now() print(x.strftime("%H"))	17
%p	AM/PM	import datetime x = datetime.datetime.now() print(x.strftime("%p"))	PM
%x	Local version of date	import datetime x = datetime.datetime.now() print(x.strftime("%x"))	12/31/18

Setting Time Zones

To work with time zones, you can use the pytz library, which provides access to the IANA time zone database.

Example:

```
from datetime import datetime
import pytz
# Define time zones
utc_zone = pytz.utc
new_york_zone = pytz.timezone('America/New_York')

# Current time in UTC
utc_time = datetime.now(utc_zone)
print("Current UTC time:", utc_time)
# Convert to New York time
ny_time = utc_time.astimezone(new_york_zone)
print("New York time:", ny_time)

# Get current time in New York
ny_now = datetime.now(new_york_zone)
```

```
print("Current New York time:", ny_now)
```

2. Formatting and Parsing

You can format and parse dates and times using the strftime and strptime methods in the datetime module, along with dateutil for more complex parsing.

Example:

```
from datetime import datetime
from dateutil import parser
# Current date and time
now = datetime.now()
# Formatting date and time
formatted_now = now.strftime('%Y-%m-%d %H:%M:%S')
print("Formatted current time:", formatted_now)
# Parsing a date string
date_str = "2024-01-01 15:30:00"
parsed_date = datetime.strptime(date_str, '%Y-%m-%d %H:%M:%S')
print("Parsed date:", parsed_date)

# Parsing using dateutil
date_str2 = "January 1, 2024, 3:30 PM"
parsed_date2 = parser.parse(date_str2)
print("Parsed date with dateutil:", parsed_date2)
```

3. Performing Relative Timedeltas

You can use the timedelta class from the datetime module to perform arithmetic with dates and times, allowing you to easily calculate relative dates.

Example:

```
from datetime import datetime, timedelta
# Current date and time
now = datetime.now()
print("Current date and time:", now)
# Calculate relative dates
tomorrow = now + timedelta(days=1)
print("Tomorrow:", tomorrow)

next_week = now + timedelta(weeks=1)
print("Next week:", next_week)
# Subtracting time
yesterday = now - timedelta(days=1)
print("Yesterday:", yesterday)
```

```
# Using relativedelta from dateutil for more complex operations
from dateutil.relativedelta import relativedelta
# Adding months and years
next_month = now + relativedelta(months=1)
print("Next month:", next_month)
next_year = now + relativedelta(years=1)
print("Next year:", next_year)
```

Points to Remember

- Python provides robust support for date and time manipulation through its built-in datetime module and several powerful third-party libraries like dateutil, Arrow, Pendulum, and python-tzdata.
- The datetime module is part of the standard library and provides classes for manipulating dates and times. It includes functionalities for creating, formatting, and performing arithmetic on dates and times.
- Dateutil is a powerful extension of the datetime module that provides additional features, such as parsing dates from strings and handling time zones. It simplifies date manipulation with utilities for relative deltas, recurrence rules, and more.
- Arrow is a lightweight library that simplifies working with dates and times in Python. It provides an intuitive API for creating, formatting, and converting dates, along with built-in timezone handling and human-friendly features.
- Pendulum is a robust datetime library that extends datetime with advanced features like duration calculations, timezone conversions, and natural language support. It offers immutable instances, making it easier to work with dates and times without side effects.
- python-tzdata is a package that provides the IANA time zone database for Python applications. It allows for accurate timezone conversions and offsets, ensuring applications handle date and time correctly across different regions.
- In Python, you can effectively manage time zones using pytz, format and parse dates using strftime and strptime, and perform arithmetic with dates using timedelta and relativedelta.

Application of learning 3.2.:

As full stack developer you are asked to develop a python program that can be used while calculating the age of students in order to know if they are allowed to take national id card depending on entered age and the current date.
The program has to tell user if the entered student is allowed or not.

Indicative content 3.3: Applying Python Libraries

Duration: 5 hrs

Theoretical Activity 3.3.1: Description of Python Libraries

Tasks:

1: You are requested to describe the following python libraries:

i. Matplotlib
ii. Numpy
iii. Pandas

2: Write your findings on paper/flipchart

3: Present your findings to the whole class or trainer

4: For more clarification read key reading 3.3.1 and ask questions where necessary.

Key readings 3.3.1. Description of Python Library

1. Definition

A Python library is a collection of modules that provide reusable code to perform specific tasks. Libraries can be either part of the standard library (which comes bundled with Python) or third-party libraries that can be installed via package managers like pip. Python libraries promote code reuse and modular programming, making it easier to develop complex applications.

2. Python Standard Library

The Python Standard Library is a collection of modules and packages that come with Python. It provides a wide range of functionalities, including file I/O, system calls, data manipulation, and more.

The standard library allows developers to perform common programming tasks without needing to install additional packages.

Key Features

- ✓Comprehensive documentation.
- ✓Built-in modules for various programming tasks.
- ✓Ensures consistency across Python installations.

3. Matplotlib

Description: Matplotlib is a plotting library for Python that enables the creation of static, animated, and interactive visualizations. It is highly customizable and works well with NumPy and **Pandas.**

Key Features

✓Support for a wide variety of plots (line, scatter, bar, histogram, etc.).
✓Extensive customization options (colors, labels, fonts).
✓Integration with Jupyter notebooks for interactive plotting.

Example:

```
import matplotlib.pyplot as plt
import numpy as np

# Data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Plot
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```

4. NumPy

Description: NumPy (Numerical Python) is a library for numerical computing in Python. It provides support for arrays, matrices, and a wide range of mathematical functions to operate on these data structures.

Key Features:

✓N-dimensional arrays (ndarray) for efficient storage and manipulation.
✓Mathematical functions for linear algebra, statistical analysis, and more.
✓Broadcasting capabilities for arithmetic operations on arrays of different shapes.

Example:

```
import numpy as np

# Creating an array
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Basic operations
print("Array:\n", arr)
print("Sum:", np.sum(arr))
print("Mean:", np.mean(arr))
print("Transpose:\n", arr.T)
```

5. Pandas

Description: Pandas is a powerful data manipulation and analysis library built on top of NumPy. It provides data structures like Series and DataFrame to handle labeled data efficiently.

Key Features:

- ✓DataFrame for handling two-dimensional labeled data.
- ✓Powerful data manipulation functions (grouping, merging, reshaping).
- ✓Built-in support for reading and writing data from various file formats (CSV, Excel, SQL, etc.).

Example:

```
import pandas as pd

# Creating a DataFrame
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)

# DataFrame operations
print("DataFrame:\n", df)
print("Mean Age:", df['Age'].mean())
print("Filtered:\n", df[df['Age'] > 28])
```

Practical Activity 3.3.2: Using python libraries

Task:

1. Read the task bellow
   As a full stack developer, you are asked to go to the computer lab to use python libraries described in key readings 3.3.1.
2. Refers to provided key reading 3.3.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 3.3.2 Using python libraries

1. Importing Libraries

To use any library in Python, you need to import it. This is typically done at the beginning of your script or notebook.

Example of Importing Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

- ✓ numpy is imported as np for brevity.
- ✓ pandas is imported as pd.
- ✓ matplotlib.pyplot is imported as plt.

2. Accessing Functionality

NumPy

NumPy provides a powerful array object and a range of functions for numerical operations.

Example:

```
# Create a NumPy array
array = np.array([1, 2, 3, 4, 5])
# Basic operations
mean_value = np.mean(array)
sum_value = np.sum(array)
print("Array:", array)
print("Mean:", mean_value)
print("Sum:", sum_value)
```

Pandas

Pandas is used for data manipulation and analysis, primarily with its DataFrame structure.

Example:

```
# Create a DataFrame
data = {
'Name': ['Alice', 'Bob', 'Charlie'],
 Age': [25, 30, 35]
}
df = pd.DataFrame(data)
# Accessing DataFrame functionalities
mean_age = df['Age'].mean()
filtered_df = df[df['Age'] > 28]
print("DataFrame:\n", df)
print("Mean Age:", mean_age)
```

```
print("Filtered DataFrame:\n", filtered_df)
```

Matplotlib

Matplotlib is used for creating static, animated, and interactive visualizations.

Example:

```
# Simple plot using Matplotlib
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```

3. Understanding Scope According to the Namespace

In Python, a namespace is a container that holds a set of identifiers (names) and ensures that all names are unique within that namespace. When you import libraries, their functions and classes are accessible within the current namespace.

Example of Scope and Namespace

```
# Importing libraries
import numpy as np
import pandas as pd
# Defining a function that uses NumPy
def calculate_statistics(data):
mean = np.mean(data)
return mean
# Using the function
data_array = np.array([1, 2, 3, 4, 5])
mean_value = calculate_statistics(data_array)
print("Mean Value:", mean_value)
# Accessing a Pandas function directly
df = pd.DataFrame({'A': [1, 2, 3]})
print("DataFrame:\n", df)
```

Scope Considerations

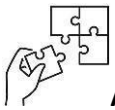
- **Global Scope:** Functions and variables defined outside any function have a global scope and can be accessed anywhere in the file.
- **Local Scope:** Variables defined inside a function are local to that function and cannot be accessed outside of it.
- **Namespace Access:** When you import libraries, you access their functions using the specified alias (like np for NumPy). If you try to access a function without the alias, you will get a NameError.

Example of Scope Errors

```
# Attempting to access a function without the namespace
try:
np.mean([1, 2, 3])  # Correct usage
mean([1, 2, 3])     # Incorrect, will raise NameError
except NameError as e:
print(e)  # Output: name 'mean' is not defined
```

Points to Remember

- The Python Standard Library provides a robust foundation for programming tasks, while libraries like Matplotlib, NumPy, and Pandas significantly enhance Python's capabilities in data visualization, numerical computing, and data analysis. Together, these libraries make Python a powerful tool for scientific computing, data analysis, and more.
- To use any library in Python, you need to import it. This is typically done at the beginning of your script or notebook.
- NumPy provides a powerful array object and a range of functions for numerical operations.
- Pandas is used for data manipulation and analysis, primarily with its DataFrame structure.
- Matplotlib is used for creating static, animated, and interactive visualizations.



Application of learning 3.3.

You're working for a retail store located in Nyanza District. You are tasked with analysing the monthly sales data for a retail store. The sales data is manually defined in a Python dictionary. The objective is to calculate total and average sales, and visualize the sales trend over the months using Python libraries.

Indicative content 3.4: Applying System Automation

Duration: 5 hrs

Theoretical Activity 3.4.1: Identification of tasks to automate and to be prioritized

Tasks:

1. You are requested to identify tasks to automate and to be prioritized
2. Write your findings on paper/flipchart
3. Present your findings to the whole class or trainer
4. For more clarification read key reading 3.1.1 and ask questions where necessary.

Key readings 3.4.1.: Identification of tasks to automate and to be prioritized

1. Identification of Tasks to Automate

1.1. Database Migrations

Automate the process of updating database schemas.

Use tools like Alembic or Django migrations to streamline the transition between database versions.

1.2. Configuration File Updates

Automate updates to configuration files across environments (e.g., development, staging, production).

Use scripts or configuration management tools like Ansible or Chef.

1.3. Service Restarts

Automate the restarting of services after deployments or configuration changes.

Implement health checks and automated scripts to ensure services are running as expected.

1.4. Testing and Verification

Automate unit tests, integration tests, and end-to-end tests.

Use CI/CD tools like Jenkins, GitHub Actions, or Travis CI to run tests automatically upon code changes.

1.5. Logging and Notifications

Automate the collection and monitoring of logs.

Set up notifications for critical events using tools like Slack, email, or monitoring dashboards (e.g., Prometheus).

2. Identification of Tasks to Be Prioritized

2.1. Repetitive

Tasks that are performed frequently (e.g., daily, weekly).

Examples: data backups, report generation, and code deployments.

2.2. Time-Consuming

Tasks that take a significant amount of manual time.

Examples: manual data entry, resource provisioning, and batch processing.

2.3. Error-Prone

Tasks that are susceptible to human error, leading to inconsistencies or failures.

Examples: manual configuration updates, data migrations, and testing processes.

2.4. Critical for Deployment Speed

Tasks that impact the speed and efficiency of the deployment pipeline.

Examples: automated testing and deployment scripts that need to be executed quickly to reduce downtime.

Practical Activity 3.4.2: Installing Python Automation Libraries

Task:

1. Read the task bellow

As a full stack developer, you are asked to go to the computer lab to install the following python automation libraries:

i. fabric
ii. ansible
iii. salt
iv. boto3
v. vsphere-automation-sdk

2. Refers to provided key reading 3.4.2, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 3.4.2 Installation of Python Automation Library

1. introduction

When selecting a Python automation library, it's important to consider various factors such as ease of use, community support, features, and specific use cases. Below is a comparison of several automation libraries, along with key considerations for each.

1. Fabric

Fabric is a simple and lightweight Python library for streamlining the use of SSH for application deployment and system administration tasks.

Key Features:

- ✓ SSH command execution and file transfer.

- ✓ Easy to define tasks in Python code.
- ✓ Supports task parallelism.

Use Cases:

- ✓ Deploying applications to remote servers.
- ✓ Running administrative commands across multiple servers.

Considerations:

- ✓ Best for small to medium-scale deployments.
- ✓ Requires familiarity with SSH.

2. Ansible

Ansible is a powerful automation tool for configuration management, application deployment, and orchestration, using a simple YAML syntax.

Key Features:

- ✓ Agentless architecture.
- ✓ Extensive module library for various tasks.
- ✓ Strong community and enterprise support.

Use Cases:

- ✓ Automating configuration management.
- ✓ Orchestrating complex deployments across multiple environments.

Considerations:

- ✓ Requires YAML knowledge.
- ✓ More suited for larger environments with complex setups.

3. SaltStack

SaltStack is a configuration management and orchestration tool that allows for real-time automation and monitoring.

Key Features:

- ✓ Event-driven automation.
- ✓ High scalability with a master-minion architecture.
- ✓ Supports a wide range of operating systems.

Use Cases

- ✓ Managing large infrastructures.
- ✓ Real-time monitoring and automation.

Considerations:

- ✓ More complex setup compared to other tools.
- ✓ **Best for environments requiring real-time updates.**

4. Boto3

Boto3 is the Amazon Web Services (AWS) SDK for Python, allowing Python developers to write software that makes use of AWS services.

Key Features:

- ✓ Comprehensive access to AWS services.
- ✓ Simple and intuitive API.

- ✓ Supports resource management and automation.

Use Cases:

- ✓ Automating AWS resource provisioning.
- ✓ Managing AWS services like S3, EC2, and Lambda.

Considerations:

- ✓ Specific to AWS environments.
- ✓ Requires understanding of AWS services.

5. vSphere Automation SDK for Python

This SDK provides a Python client for automating VMware vSphere environments, enabling easy interaction with vSphere APIs.

Key Features:

- ✓ Access to vSphere APIs for managing virtualized environments.
- ✓ Supports operations such as VM lifecycle management, networking, and storage.

Use Cases:

- ✓ Automating VM provisioning and management.
- ✓ Integrating with existing VMware workflows.

Considerations:

- ✓ Best suited for organizations using VMware vSphere.
- ✓ Requires familiarity with VMware's architecture and APIs.

Factors to Consider

- **Ease of Use**

How easy is it to get started with the library?

Does it require extensive setup or configuration?

- **Community Support**

Is there a strong community or documentation available?

Are there frequent updates and active maintenance?

- **Features**

Does the library provide the necessary features for your specific automation tasks?

How well does it integrate with other tools or environments?

- **Scalability**

Can the library handle the scale of your infrastructure?

Is it suitable for both small and large environments?

- **Specific Use Cases**

Does the library cater to your specific needs (e.g., cloud automation, configuration management)?

Are there any dependencies on certain platforms or services?

Practical Activity 3.4.3: Developing python script

Task:

1. Read the task bellow

   As a full stack developer, you are asked to go to the computer lab to develop python scripts.

2. Refers to provided key reading 3.4.3, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 3.4.3 Developing python script

1. Introduction

Developing a Python script that demonstrates the use of library functions, logical structuring, and logging. For this example, we'll create a script that automates the deployment of a simple web application using Fabric for SSH tasks, alongside logging to track its operations.

Example Script: Web Application Deployment

2. Install Required Library

Ensure you have Fabric installed. You can install it using pip:

pip install fabric

3. Script Structure

The script will:

- ✓ Connect to a remote server.
- ✓ Upload application files.
- ✓ Restart the web server.
- ✓ **Log the deployment process.**

4. Python Script

```
import logging
from fabric import Connection
# Set up logging
logging.basicConfig(
level=logging.INFO,
format='%(asctime)s - %(levelname)s - %(message)s',
handlers=[
logging.FileHandler("deployment.log"),
 logging.StreamHandler()
]
```

```
)
# Define the deployment function
def deploy_application(host, user, app_path, remote_path):
   try:
      logging.info("Starting deployment to %s", host)
# Create a connection to the remote server
with Connection(host=host, user=user) as conn:
logging.info("Connected to %s", host)
 # Upload application files
logging.info("Uploading files from %s to %s", app_path, remote_path)
conn.put(app_path, remote=remote_path)
# Restart the web server
logging.info("Restarting the web server")
conn.run("sudo systemctl restart apache2")  # Change this command based on
your web server
logging.info("Deployment completed successfully to %s", host)
except Exception as e:
logging.error("Deployment failed: %s", e)

# Main execution
if __name__ == "__main__":
# Configuration
HOST = "your_remote_server_ip"
USER = "your_username"
APP_PATH = "./path_to_your_application/*"  # Local path to your application
files
REMOTE_PATH = "/var/www/html"  # Remote path where files will be
deployed
   deploy_application(HOST, USER, APP_PATH, REMOTE_PATH)
```

Breakdown of the Script

5. Logging Setup

The logging module is configured to log messages to both a file (deployment.log) and the console.

Log levels (INFO, ERROR) help track the flow and any issues.

6. Function Definition

The deploy_application function handles the deployment process.

It connects to the remote server using Fabric, uploads files, and restarts the web server.

Each step is logged for transparency and debugging.

7. Main Execution Block

The script's entry point sets configuration variables (host, user, app paths).
It calls the deploy_application function with these variables.

Conclusion

This Python script demonstrates how to use library functions from Fabric, structure code logically, and implement logging for tracking operations. Adjust the server commands and paths according to your specific needs to make it functional in your environment.

Practical Activity 3.4.4: Integrating script with Deployment Process

Task:

1. Read the task bellow
   As a full stack developer, you are asked to go to the computer lab to integrate the developed script on activity 3.4.3 with deployment process.
2. Refers to provided key reading 3.4.4, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 3.4.4 Integrating script with Deployment Process

Integrating a Python script into a deployment process for system automation involves several strategies. Below are methods to trigger the script post-deployment, along with considerations for security and integration into CI/CD pipelines.

Step1 : Trigger Method to Initiate the Python Script Post-Deployment

After the main deployment tasks are completed, the script can be triggered automatically. This can be done using a simple command in a shell script or a deployment tool.

Example:

```
# Shell script to deploy an application and trigger the Python script
#!/bin/bash
# Deploy the application (pseudo-command)
echo "Deploying application..."
# (Your deployment commands here)
# Trigger the Python deployment script
python3 deploy_script.py
```

Step2 : Direct Execution After Deployment Completion

You can directly call the Python script at the end of your deployment process, ensuring it runs only after the main deployment is successful.

Example in a Shell Script:

```
# After deployment commands
echo "Deployment completed successfully."
python3 deploy_script.py
```

Step3 : Integration with CI/CD Pipelines

Integrate the Python script into your CI/CD pipeline using tools like Jenkins, GitHub Actions, or GitLab CI. You can add a step in your pipeline configuration file to run the script.

Example for GitHub Actions:

```
name: Deploy Application
on:
  push:
branches:
- main
jobs:
  deploy:
runs-on: ubuntu-latest
steps:
- name: Checkout code
uses: actions/checkout@v2
- name: Deploy application
 run: |
# Your deployment commands here
- name: Run deployment script
run: python3 deploy_script.py
```

Step 4 : Scheduled Execution at Specific Intervals

You can schedule the execution of the script using cron jobs on Linux or Task Scheduler on Windows.

Example of a Cron Job:

```
# Open crontab
crontab -e
# Schedule the script to run daily at midnight
0 0 * * * /usr/bin/python3 /path/to/deploy_script.py
```

2. Implement Security Measures

To ensure that sensitive information is protected and access to the script is controlled, consider the following measures:

a. Restrict Script Access

File Permissions: Set appropriate file permissions to restrict access to the script.

```
chmod 700 deploy_script.py  # Only the owner can read, write, and execute
```

User Roles: Limit access to the script to specific user roles that need to execute it.

b. Environment Variables for Sensitive Information

Store sensitive information (like passwords, API keys) in environment variables instead of hardcoding them in the script.

Example:

```
import os
DB_PASSWORD = os.getenv('DB_PASSWORD')
```

You can set the environment variables in your shell or CI/CD pipeline configuration.

c. Use Secrets Management Tools

Utilize secrets management tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault to store and manage sensitive information securely.

Example Using AWS Secrets Manager:

```
import boto3
# Load secret
def get_secret():
 client = boto3.client('secretsmanager')
secret_value = client.get_secret_value(SecretId='my_secret_id')
return secret_value['SecretString']
```

Conclusion

Integrating your Python script into the deployment process can enhance automation and efficiency. By triggering the script post-deployment, integrating it into CI/CD pipelines, scheduling it, and implementing robust security measures, you can ensure a smooth and secure deployment workflow. Always prioritize securing sensitive information and controlling access to your automation scripts.

Practical Activity 3.4.5: Testing and monitoring the automated tasks

Task:

1. Read the task bellow
   As a full stack developer, you are asked to go to the computer lab to test and monitor the automated tasks of activity 3.4.4.
2. Refers to provided key reading 3.4.5, perform the task described above.
3. Present your work to the trainer and whole class.

Key readings 3.4.5 Testing and monitoring the automated tasks

Approach to testing and monitoring your Python automation scripts, ensuring they operate effectively and efficiently.

1. Thorough Testing

To ensure your automation script functions as intended, implement various testing strategies:

a. Unit Testing

Write unit tests for individual functions within your script using the unittest or pytest frameworks.

Example:

```
import unittest
from your_script import deploy_application  # Import your function
class TestDeployment(unittest.TestCase):
def test_deploy_application(self):
 # Mock parameters
host = "test_host"
user = "test_user"
app_path = "test_path"
remote_path = "test_remote_path"
# Call the function and assert expected results
result = deploy_application(host, user, app_path, remote_path)
self.assertIsNone(result)  # Assuming the function returns None on success
if __name__ == "__main__":
unittest.main()
```

b. Integration Testing

Test the script in a staging environment that mimics production. This will help identify issues in the interaction between components.

Example:

Deploy to a staging server and run the script, verifying that all steps complete successfully and the application behaves as expected.

c. End-to-End Testing

Conduct end-to-end tests that cover the entire deployment process, from code commit to deployment.

2. Monitor Script Logs Execution

Monitoring is crucial for identifying issues and ensuring smooth operation. Implement logging and monitoring practices:

a. Logging

Use Python's built-in logging module to record important events, errors, and execution flow.

Example:

```
import logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("deployment.log"),
        logging.StreamHandler()
    ]
)
# Log an example message
logging.info("Starting deployment...")
```

b. Log Monitoring

Use log monitoring tools like ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, or Papertrail to analyze logs in real-time.

Set up alerts for errors or critical warnings in your logs.

4. Refine and Improve

Continuous improvement is key to maintaining a robust automation script:

a. Code Review

Regularly review the code for improvements, ensuring it adheres to best practices.

Peer reviews can help catch potential issues and provide feedback for enhancements.

b. Performance Monitoring

Analyze the script's performance, noting execution time and resource usage.

Use profiling tools like cProfile to identify bottlenecks.

Example:

```
import cProfile
def main():
    # Your main deployment function
    deploy_application(...)
cProfile.run('main()')
```

c. Feedback Loop

Collect feedback from users or stakeholders regarding the deployment process.

Identify pain points or areas that require improvement.

d. Regular Updates

Keep dependencies up to date and review the latest features or improvements in libraries used.

Regularly test the script against new versions of relevant libraries.

Conclusion

By implementing thorough testing, actively monitoring script execution, and refining

your automation process, you can ensure that your Python scripts remain robust, efficient, and effective. Continuous improvements based on testing outcomes and user feedback will help maintain a high level of reliability in your deployment automation efforts.

Step-by-step guide to testing a Python automation script, complete with a real-life example that illustrates each step clearly.

Example Scenario: Deploying a Web Application

Assume you have a Python script that automates the deployment of a web application to a server. The script includes functions for uploading files, restarting services, and logging actions.

Steps to Follow for Testing

Step 1: Write Unit Tests

Objective: Test individual functions to ensure they work correctly.

Example Function:

```
def upload_files(local_path, remote_path):
    # Simulate file upload
    if not local_path or not remote_path:
        raise ValueError("Both paths are required")
    return True  # Simulate successful upload
```

Unit Test

```
import unittest
class TestDeploymentFunctions(unittest.TestCase):
    def test_upload_files_success(self):
        result = upload_files("local/file/path", "remote/file/path")
        self.assertTrue(result)
    def test_upload_files_failure(self):
        with self.assertRaises(ValueError):
            upload_files("", "remote/file/path")
```

Run the Tests

python -m unittest test_deployment.py

Step 2: Perform Integration Testing

Objective: Test the interaction between components. Ensure that the entire deployment process works as expected.

Integration Test Example:

```
class TestDeploymentIntegration(unittest.TestCase):
    def test_deployment_process(self):
        # Simulate the entire deployment process
        result = deploy_application("host", "user", "local/path", "remote/path")
        self.assertIsNone(result)  # Assuming it returns None on success
```

Run the Tests

python -m unittest test_integration.py

Step 3: Conduct End-to-End Testing

Objective: Test the complete workflow from start to finish in a staging environment.

Set up a Staging Environment: Create a replica of your production environment.

Deploy the Application: Run the deployment script in the staging environment.

Verify Functionality: Check that the application is running as expected and all features work.

Example:

```
# Deploy to staging server
python deploy_script.py
# Verify the application
curl http://staging-server-url
```

Step 4: Monitor Script Logs Execution

Objective: Keep track of the script's execution and identify any issues.

Log Important Events: Use the logging module to capture key actions.

import logging

logging.basicConfig(filename='deployment.log', level=logging.INFO)

logging.info("Starting deployment process.")

Analyze Logs: After running the script, check the logs for any errors or warnings.

cat deployment.log

Step 5: Collect Feedback and Refine

Objective: Gather input from stakeholders and improve the script based on real-world usage.

Collect User Feedback: After deployment, ask users about their experience.

Identify Pain Points: Look for any issues they encountered during the process.

Make Improvements: Update the script based on feedback.

Example: If users report that the deployment takes too long, you might optimize file uploads or reduce the number of services restarted.

Step 6: Regular Updates and Regression Testing

Objective: Ensure that updates to the script or dependencies do not introduce new issues.

Update Dependencies: Periodically check for updates to libraries and tools used in your script.

Run All Tests: Execute unit, integration, and end-to-end tests after any changes.

python -m unittest discover

Conclusion

By following these steps—writing unit tests, performing integration and end-to-end testing, monitoring logs, collecting feedback, and regularly updating your script—you can ensure a reliable and effective automation process for your deployment tasks. This structured approach helps identify issues early and enhances the overall quality of your automation efforts.

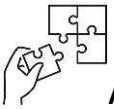
Points to Remember

- Automating certain tasks can greatly improve efficiency, reduce errors, and speed up deployment processes.
- By focusing on repetitive, time-consuming, error-prone tasks that are critical for deployment speed, teams can maximize their productivity and ensure a smoother workflow.
- Automating these tasks not only saves time but also enhances the reliability of the development and deployment processes.
- To install python automation libraries you can use “pip install library name” and remember to replace the library name with the one that you are installing. E.g pip install fabric, pip install ansible, pip install salt, pip install boto3 and pip install vsphere-automation-sdk.
- To develop Python scripts effectively, plan your goals, import necessary libraries, structure your code logically, write functions for reusable code, use library functions, handle errors, and document your work.
- Consider using a virtual environment to isolate dependencies and leverage logging for debugging.
- By following these steps and incorporating best practices, you can create efficient, maintainable, and well-structured Python scripts.
- While Integrating script with Deployment Process you follow the fllowing steps:

  Step 1: Trigger method to initiate the Python script post-deployment
  Step 2: Direct execution after deployment completion
  Step 3: Integration with CI/CD pipelines
  Step 4: Scheduled execution at specific intervals
  Step 5: Implement security measures to restrict script access and control sensitive information.
- Selecting the right Python automation library depends on your specific needs, existing infrastructure, and the complexity of the tasks you want to automate. By evaluating the features, ease of use, scalability, and community support of each

option, you can make an informed decision that aligns with your automation goals.

- While testing and monitoring the automated task you have to: Thorough testing, Monitor script logs execution, Refine and improve.



Application of learning 3.4.:

Manzi as a full-stack developer, has built a web application for BERWA School. After each deployment of the application, certain tasks must be automated to ensure the application is correctly configured and ready for use. These tasks include configuring the server, backing up the database, and sending notifications upon completion. To enhance automation, Manzi decides to integrate a Python script into the deployment process, ensuring it runs after each successful deployment and meets necessary security measures. As a full stack developer, you are tasked to develop python script that will help Manzi to perform the desired requirements for his system.

Learning outcome 3 end assessment

Written assessment

I. Match the following terms with their corresponding definitions as applied in functions:

Answers	Items	Definitions
……	1. Class	A. A blueprint for creating objects.
……	2. Object	B. A specific instance of a class.
……	3. Inheritance	C. The ability to use a method in different ways.
……	4. Polymorphism	D. Restricting access to certain components of an object.
……	5. Encapsulation	E. A class from which another class inherits.
……	6. Method Overriding	F. A derived class that inherits properties from another class.
……	7. Constructor	G. A method that replaces the implementation of a method in the superclass.
……	8. Data Hiding	H. A special method used to initialize objects.
……	9. Superclass	I. The ability to hide data from outside access.
……	10. Subclass	J. The mechanism of a class acquiring properties from another class.

II. Match the following Python libraries with their corresponding primary use cases:

ANSWER	Python libraries	Primary use cases
……	1. Matplotlib	A. Data manipulation and analysis
……	2. NumPy	B. Numerical operations on arrays and matrices
……	3. Pandas	C. Data visualization
……	4. Ansible	D. Automation and configuration management
……	5. datetime	E. Date and time handling

III. Select the correct answer from the listed one

1. What keyword is used to define a class in Python?
   A) define
   B) class
   C) object
   D) function
2. Which of the following allows a class to inherit properties from multiple classes?
   A) Single Inheritance
   B) Multiple
   C) Inheritance
   D) Multilevel Inheritance

E) Hierarchical Inheritance

3. What does the __init__() method do?
    A) It creates a new class.
    B) It initializes an object's attributes.
    C) It defines a new method.
    D) It overrides a method.
4. What is the primary benefit of encapsulation?
    A) It increases redundancy.
    B) It hides the internal state of an object.
    C) It allows multiple inheritance.
    D) It simplifies code readability.
5. In Python, what is polymorphism primarily used for?
    A) To create new classes.
    B) To allow different classes to be treated as instances of the same class.
    C) To hide data.
    D) To define class methods.

IV. State whether the following statements are True or False.

1. An object is an instance of a class.
2. In Python, all classes must inherit from a superclass.
3. Encapsulation allows for direct access to an object's private attributes.
4. Polymorphism can be achieved through method overloading.
5. A subclass can override methods from its superclass.
6. Inheritance promotes code reuse.
7. The self keyword is used to refer to an instance of a class.
8. All attributes in a class are public by default.
9. The __str__() method is used to represent an object as a string.
10. Method overloading is directly supported in Python.

6. **Complete the following sentences with correct keyword chosen in box below .**

(class, inheritance, class, constructor, Method overriding , encapsulation, object, hierarchical, polymorphism, Encapsulation)

1. A _____ is a blueprint for creating objects in Python.
2. The process of a class inheriting properties from another class is called _____.
3. In Python, you use the keyword _____ to define a class.
4. The _____ method is automatically called when an object is created.
5. _____ allows a subclass to provide a specific implementation of a method defined in its superclass.
6. Data hiding is a feature of _____ that restricts direct access to some attributes.
7. A _____ is an instance of a class.

8. Inheritance allows for the creation of a _____ structure among classes.
9. The ability to treat objects of different classes as objects of a common superclass is known as _____.
10. _____ is achieved when a subclass inherits methods and properties from a superclass.

Practical assessment

As a full stack devoloper, you have been assigned the task of developing and managing a Health Management System for GIRUBUZIMA HOSPITAL. The system will handle patient data, automate deployment tasks, and analyze health trends to ensure smooth operation and efficient service delivery. This project involves applying concepts from Object-Oriented Programming (OOP), date and time handling, Python libraries, and automation of post-deployment tasks.

Tasks:

1. Create a Patient class with attributes like name, age, gender, and disease.
2. Create specialized classes like in_patient and Outpatient inheriting from Patient.
3. Use methods that behave differently based on patient type.
4. Keep patient information private and control access through methods.
5. Ensure records are aligned with Rwanda's timezone (Central Africa Time).
6. Format timestamps for patient records.
7. Calculate days between tests and monitor recovery progress.
8. Visualize patient recovery trends.
9. Perform statistical operations on patient records.
10. Manage large datasets with patient info.
11. Use Fabric or Ansible to automate deployment-related tasks such as **restarting services** and **updating configurations**

References

Alpaydin, E. (2020). Introduction to Machine Learning (Adaptive Computation and Machine Learning series). MIT Press.

Bishop, C. Ms. (2006). Pa ern Recognition and Machine Learning. Springer.

Chollet, F. (2017). Deep Learning with Python. Manning Publications.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep Learning (Adaptive Computation and Machine Learning series). MIT Press.

Has e, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer.

Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective. MIT Press.

October 2024