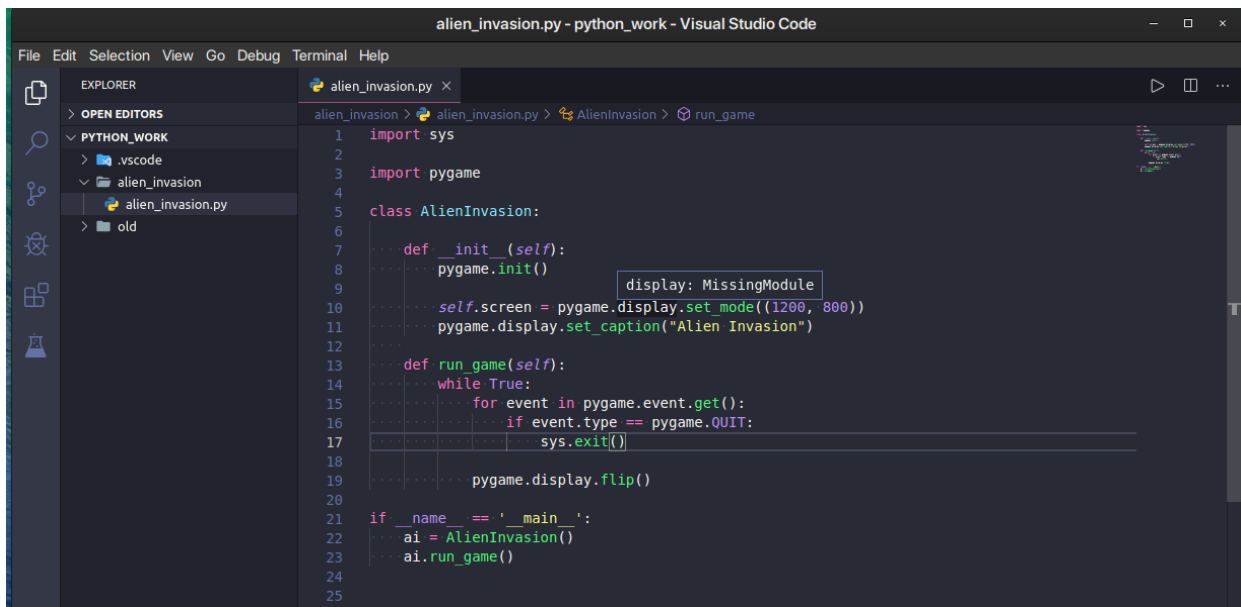


Web Development and Database Administration Level-IV

Based on November 2023, Curriculum Version II



```
1 import sys
2
3 import pygame
4
5 class AlienInvasion:
6
7     def __init__(self):
8         pygame.init()
9
10        self.screen = pygame.display.set_mode((1200, 800))
11        pygame.display.set_caption("Alien Invasion")
12
13    def run_game(self):
14        while True:
15            for event in pygame.event.get():
16                if event.type == pygame.QUIT:
17                    sys.exit()
18
19            pygame.display.flip()
20
21 if __name__ == '__main__':
22     ai = AlienInvasion()
23     ai.run_game()
24
25
```

Module Title: Object-oriented programming language

Module code: EIS WDDBA4 M01 1123

Nominal duration: 80 Hours

Prepared by: Ministry of Labor and Skill

November, 2023

Addis Ababa, Ethiopia

Table of Contents

Unit One: Identification of content needs.....	7
1.1 Introduction to Object Oriented Programming language.....	8
1.2 Basic language syntax rules and Best practices	10
1.3 Data-types, Operators and Expression	15
1.4 Sequence, Selection and Iteration constructs	27
1.5 Modular Programming Approach.....	32
1.6 Arrays and Arrays of objects.....	35
Self-Check -1.....	41
Operation sheet 1.1: Installation of Python Program for windows computer	42
Operation sheet 1.2: Installation of VS Code IDE.....	43
Operation sheet 1.3: Set Up VS Code for Python Development	44
Operation sheet 1.4: Python Selection	45
Operation sheet 1.5: Python Sequence	46
Operation sheet 1.6: Python Iteration	47
Operation sheet 1.7: Python Data types, Operators and Expression	48
Operation sheet 1.8: Operators and If-Elif-Else Statement.....	50
Operation sheet 1.9: sequence, selection and iteration.....	52
Operation sheet 1.10: Arrays and Arays Object	54
Lap Tests.....	56
Unit Two: Basic OO principles	57
2.1 Primitive Member Variables in Class Implementation.....	58
2.2 Flexible Object Construction.....	62
2.3 User-Defined Aggregation in Class Design	63
2.4 Navigating Hierarchical Inheritance	65
2.5 Code Extension through Versatile Polymorphism	68
Self-check: 3.....	69
Operation sheet 2.1 : Object Construction	71
Operation sheet 2.2: Primitive member of variables.....	72
Operation sheet 2.3: Object Construction and User-Defined Aggregation.....	74
Operation sheet 2.4: Inheritance.....	75
Operation sheet 2.5: Polymorphism	77

Lap Tests.....	79
Unit Three: Debug code.....	80
3.1 Integrated Development Environment (IDEs)	81
3.2 Program Debugging Techniques	83
Self-Check : 3.....	85
Unite Four: Document activities	86
4.1 Crafting Maintainable Object-Oriented Code	87
4.2 Documentation of Object-oriented Programming	94
Self-check :4.....	95
Unite Five: Test code	96
5.1. Simple Tests in Object-Oriented Programming.....	97
5.2. Embracing Corrections in Code and Documentation.....	99
Self-Check:5.....	100
References	101
Developer’s Profile	102

Acknowledgment

Ministry of Labor and Skills wish to extend thanks and appreciation to the many representatives of TVET instructors and respective industry experts who donated their time and expertise to the development of this Teaching, Training and Learning Materials (TTLM).

Page 4 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

Acronym

ADD	-----	Addition / Additive
API	-----	Application Programming Interface
CSV	-----	Comma-Separated Values
IDE	-----	Integrated Development Environment
OO	-----	Object-Oriented
OOP	-----	Object-Oriented Programming
VS	-----	Visual Studio

Introduction to the Module

Object-oriented programming (OOP) is a programming paradigm that structures software design around objects, enabling the creation of modular and reusable code. At its core, OOP focuses on modeling real-world entities as objects that encapsulate data (attributes or properties) and behavior (methods or functions) within a single unit.

This module is designed to meet the industry requirement under the Web development and database administration occupational standard, particularly for the unit of competency: Apply Object-Oriented Programming Language Skills

Module covers the units:

- Basic language syntax and layout
- Basic OO principles
- Debug code
- Document activities
- Test code

Learning Objective of the Module

- Apply basic language syntax and layout
- Apply basic OO principles in the target language
- Perform Debugging code
- Document activities
- Identify and perform Test code

Module Instruction

For effective use this modules trainees are expected to follow the following module instruction:

1. Read the information written in each unit
2. Accomplish the Self-checks at the end of each unit and perform operation sheet and
3. Read the identified reference book for Examples and exercise

Unit One: Identification of content needs

This unit is developed to provide you the necessary information regarding the following content coverage and topics:

- Strategic intent of website
- Development of information requirement
- Identification and Categorization of Information
- Content Requirements

This unit will also assist you to attain the learning outcomes stated in the cover page.

Specifically, upon completion of this learning guide, you will be able to:

- Identify strategic intent of website from business requirements and client expectations
- Develop information requirement
- Identify required information and grouping into business schemes
- Determine content requirements for each processes

1.1 Introduction to Object Oriented Programming language

Object-oriented programming (OOP) is a programming approach developed to enhance productivity by addressing the limitations of the procedural programming approach. It has given rise to popular languages like C++ and Java, designed to manage the growing size and complexity of programs. OOP is centered on objects, which can represent real-world entities or concepts, and are defined by classes. These objects have attributes and behaviors, and interact with each other through messages. For instance, in a program, a ‘customer’ object might request a bank balance from an ‘account’ object.

The data and code within an object can be defined as a user-defined data type using a class, essentially making objects variables of the class type. This encapsulation of data and functions within objects allows for interactions without the need for detailed knowledge of each other’s data or code. Key principles of OOP such as Object, Classes, Data Abstraction, Encapsulation, Inheritance, and Polymorphism are fundamental in managing complexity and promoting code reusability. Thus, OOP provides a structured approach to software development, modeling data as interactive objects

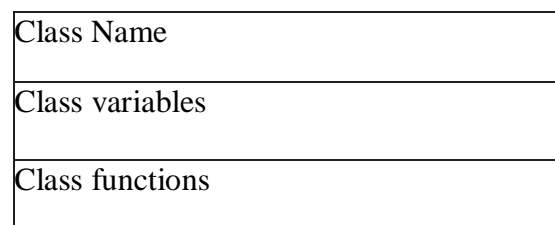


Fig: 1.1: Class Structure

For example, there is a class Employee which has Sue, Bill, Al, Hal, David different employees. Each employee will have unique identity; so they will form the objects of the class Employee.

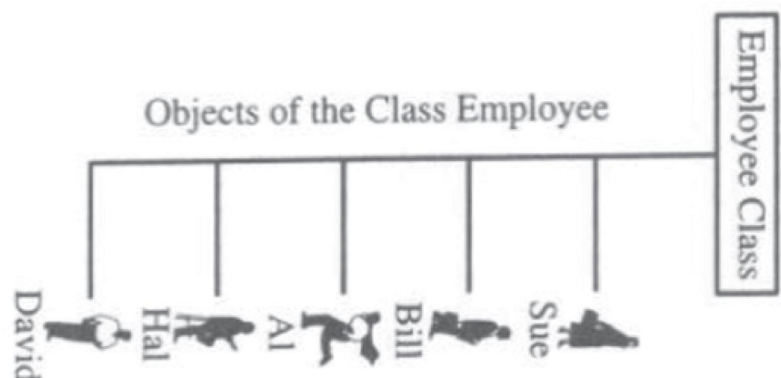


Fig: 1.2: Employee class example

Data Abstraction in Object-Oriented Programming (OOP) is the representation of essential features without including background details, much like how a switchboard operates; you press switches without needing to understand their internal workings. Encapsulation, another OOP concept, refers to the bundling of data and methods within a single unit or object, allowing the object to control its own state and behavior. This encapsulation hides internal implementation details, enabling objects to interact via well-defined interfaces, thereby enhancing security, modifiability, and reducing dependencies in the codebase. In the Figure 1.3 item is a class which has `keep_data` as member variable which cannot be accessed from outside directly. It can be accessed only via the member functions `set()` and `get_value()`.

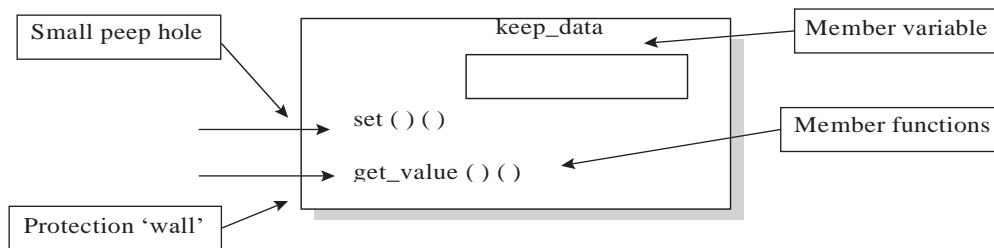


Fig: 1.3: Encapsulated Data and Functions in class item

Inheritance in Object-Oriented Programming (OOP) allows new classes to be created based on existing ones, inheriting their attributes and behaviors. This promotes code reuse, as subclasses can extend or modify their parent classes' functionality. It helps establish a class hierarchy reflecting real-world relationships, aiding in code organization and maintenance. For instance, a motorcycle is a class but also part of the two-wheelers class. Two wheelers class in turn is a member of automotive class as shown in Fig. 1.4. The automotive is an example of base class and two wheelers is its derived class. In simple words, we can say a motor cycle is a two wheeler automotive.

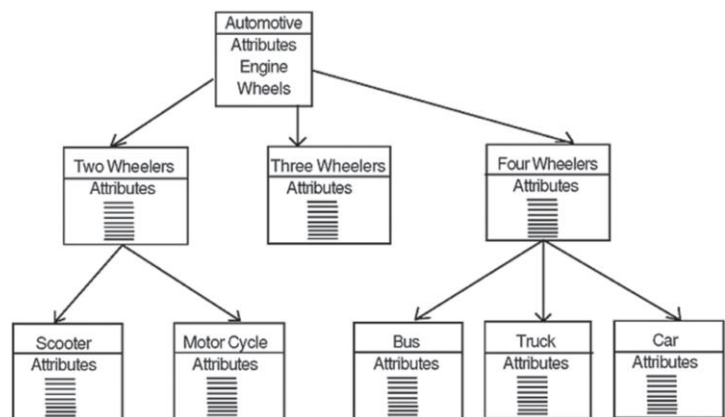


Fig: 1.4: Automotive class

Polymorphism, meaning "many forms," allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used for entities of various types, providing flexibility and allowing different objects to be manipulated through a unified interface.

Polymorphism is typically achieved through method overriding (having methods in subclasses with the same name as in the superclass) and method overloading (multiple methods with the same name but different parameters). For example let us consider Fig 1.5 the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings then the operation would produce a third string by concatenation.

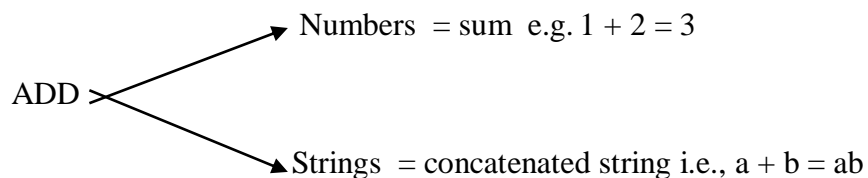


Fig: 1.5: Addition operation

OOP languages like Java, Python, C++, and others provide robust frameworks for implementing these principles, offering developers the tools to create modular, maintainable, and extensible software systems. By leveraging OOP concepts, programmers can design solutions that mimic real-world scenarios, promote code organization, enhance code reusability, and facilitate easier maintenance and scalability of applications.

1.2 Basic language syntax rules and Best practices

Basic language syntax rules and best practices play a crucial role in ensuring the effectiveness, readability, and maintainability of Object-Oriented Programming (OOP) code. These guidelines govern how developers write and structure their code within OOP paradigms, contributing to the overall quality of the software.

A. Naming convention

One fundamental syntax rule in OOP languages is adhering to naming conventions. Meaningful and descriptive names for classes, methods, variables, and other entities enhance code comprehension. Employing camelCase or snake_case for naming, depending on the language's conventions, helps improve code readability.

Additionally, using clear and expressive names that accurately represent the purpose or functionality of an object or method assists other developers in understanding the codebase efficiently.

For instance, in Java, classes typically use CamelCase while methods and variables use camelCase.

```
class ShoppingCart:
    def __init__(self):
        self.customerName = ""

    def addItem(self, itemName):
        pass
```

Consider the following example:

In the above example in the **ShoppingCart** class, the class name adheres to CamelCase, while **addItem** utilizes camelCase for the method name. Consistently applying these conventions improves code readability and helps other developers easily understand the codebase.

B. Encapsulation

Encapsulation, a key principle in OOP, emphasizes restricting access to certain components of an object. To implement encapsulation effectively, the use of access modifiers (such as private, protected, and public) is essential. Encapsulation ensures that data remains hidden and can only be accessed or modified through well-defined interfaces, thereby preventing unintended modifications and promoting better code maintainability. Consider the following Java code

Example:

```
class BankAccount:
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(amount, "deposited successfully.")
        else:
            print("Invalid deposit amount.")
    def withdraw(self, amount):
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            print(amount, "withdrawn successfully.")
        else:
            print("Invalid withdrawal amount or insufficient funds.")
    def getBalance(self):
        return self.balance
```

In this example:

Attributes: The `self.balance` attribute represents the current balance of the bank account. It's initialized to 0 within the `__init__` method.

Methods:

- **`deposit(self, amount)`:** Accepts an amount to be deposited. It checks if the amount is positive and, if so, increases the balance by that amount.
- **`withdraw(self, amount)`:** Accepts an amount to be withdrawn. It verifies if the withdrawal amount is positive and less than or equal to the current balance before deducting it from the balance.
- **`getBalance(self)`:** Returns the current balance.

Encapsulation in this context involves restricting direct access to the **balance** attribute from outside the class. This means that external code cannot directly modify or view the **balance** attribute without using the provided methods (**`deposit`**, **`withdraw`**, **`getBalance`**).

C. Inheritance

Another critical syntax rule involves the proper use of inheritance. While inheritance enables code reuse and promotes a hierarchical structure, it should be employed judiciously. It's essential to favor composition over inheritance when possible, as excessive deep inheritance hierarchies can lead to complex and tightly coupled code. Using inheritance selectively and considering alternative design patterns helps prevent issues related to tight coupling and enhances code flexibility. Look the Python example below.

```
class Animal:
    def make_sound(self):
        pass
class Dog(Animal):
    def make_sound(self):
        print("Woof!")
class Cat(Animal):
    def make_sound(self):
        print("Meow!")
```

D. Polymorphism

Polymorphism, a core OOP concept, should be used appropriately to improve code readability and maintainability. Method overriding and method overloading should follow logical and consistent naming conventions, making it easier for developers to understand the intended behavior of overridden or overloaded methods.

By using polymorphism effectively, developers can write code that is more adaptable to changes and promotes code reuse. get in your consideration the following java example.

```
class Animal {
    public void makeSound() {
        System.out.println("Some sound");
    }
}
class Dog extends Animal {
    public void makeSound() {
        System.out.println("Woof!");
    }
}
class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

Polymorphism: It's exemplified when you call the **makeSound** method on objects of type **Dog** or **Cat**. Despite all objects being treated as instances of the **Animal** class, the specific implementations of **makeSound** from their respective subclasses (**Dog** or **Cat**) are invoked.

Design Pattern

Best practices in OOP also emphasize the importance of modularization and design patterns. Breaking down complex systems into smaller, manageable modules or classes enhances code maintainability and reusability. Design patterns like Factory, Singleton, Observer, and others provide standardized solutions to common design problems, aiding in creating robust and scalable applications. You can see the example in Python.

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

The Singleton pattern in Python ensures that only one instance of the class **Singleton** is created throughout the program's execution. The **_instance** variable stores the single instance, and the **__new__** method checks if an instance exists before creating a new one, thereby ensuring a unique instance.

E. Single Responsibility Principle (SRP)

Moreover, following the Single Responsibility Principle (SRP) ensures that each class or module has a single, well-defined purpose, reducing code complexity and making it easier to maintain and extend. Additionally, adhering to the Don't Repeat Yourself (DRY) principle by avoiding duplication of code promotes code reusability and minimizes errors in the system. The following is an example in Python.

```
class Logger:
    def log(self, message):
        # Code to log messages to a file or console
        print(f"Log: {message}")
class PaymentProcessor:
    def process_payment(self, amount):
        # Code to process payments
        # For demonstration purposes, let's just print
        # a message
        print(f"Processing payment of {amount}")
```

In this example:

- The **Logger** class is responsible solely for logging messages. It contains a **log** method that handles logging messages to a file, database, console, etc. For brevity, it simply prints the log message to the console using **print**.
- The **PaymentProcessor** class focuses entirely on processing payments. It contains a **process_payment** method that simulates the payment processing functionality. Here, for demonstration purposes, it only prints a message indicating the amount being processed.

By separating concerns into distinct classes (**Logger** for logging and **PaymentProcessor** for payment processing), we adhere to the Single Responsibility Principle. Each class encapsulates a single responsibility—logging or payment processing—making the code easier to maintain, understand, and modify.

1.3 Data-types, Operators and Expression

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which can contain data in the form of attributes (variables) and behavior in the form of methods (functions). These objects are instances of classes, which act as blueprints or templates defining the structure and behavior of the objects.

There are different OOP languages but among them our focus is Python. Understanding Object-Oriented Programming across different languages can offer valuable insights into the common principles and syntax variations. Here are brief insights into how some other languages handle OOP concepts and why Python will be our main focus:

A. Java

- **Class-Based:** Like Python, Java is a class-based OOP language where everything is encapsulated within classes.
- **Strongly Typed:** Java requires explicit type declarations for variables, making it statically typed.
- **Interface:** Java uses interfaces for achieving abstraction and defining contracts for classes to implement.

B. C++

- **Hybrid Approach:** C++ supports both procedural and object-oriented programming paradigms.
- **Manual Memory Management:** Unlike Python, C++ requires explicit memory management, which can be more error-prone but allows for more fine-tuning and efficiency.
- **Multiple Inheritances:** C++ supports multiple inheritances, allowing a class to inherit from multiple base classes.

C. JavaScript

- **Prototype-Based:** JavaScript uses a prototype-based model instead of classes directly. Objects inherit directly from other objects.
- **Dynamic Typing:** Similar to Python, JavaScript is dynamically typed, allowing for flexibility but may lead to unexpected behaviors.
- **Functional Programming:** Besides OOP, JavaScript also supports functional programming paradigms.

D. Python (Focus)

- **Clear Syntax:** Python emphasizes readability and simplicity, making it an excellent language for beginners and experts alike.
- **Dynamically Typed:** Python is dynamically typed, meaning you don't need to declare variable types explicitly.
- **Extensive Libraries:** Python boasts a vast standard library and third-party packages, making it versatile and suitable for various applications.
- **Interpreted Language:** Python is an interpreted language, allowing for easier debugging and quicker development cycles.

1.3.1 Start coding with python

Before diving into coding with Python, it's beneficial to undertake a few considerations to set a strong foundation for your programming journey. Begin by grasping fundamental programming concepts like variables, data types, conditionals, loops, and functions. Resources such as online tutorials, books, or interactive platforms can help solidify these basics before delving into Python-specific syntax. Then Install Python on your computer by downloading the latest version from the official Python website. Select one among different options for an Integrated Development Environment (IDE) like PyCharm, Visual Studio Code, or simply use a text editor for writing and executing Python code. Ensure Python is properly configured and ready to run on your system.

- **Visual Studio Code**

Visual Studio Code (VS Code) is a popular and versatile source-code editor by Microsoft that supports various programming languages, including Python. It offers a user-friendly interface, a powerful IntelliSense, a built-in debugger, and Git integration, among other features that facilitate Python development.

Page 16 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

VS Code also allows users to customize the editor with a wide range of extensions that provide additional tools for linting, formatting, testing, virtual environments, and more. Furthermore, VS Code is cross-platform, meaning it works on Windows, macOS, and Linux, ensuring a consistent experience across different operating systems.

1.3.2 Python Data-types, Operators and Expression

Understanding Python's data types, operators, and expressions forms the fundamental groundwork for effective programming. Python offers several built-in data types, including integers, floating-point numbers, strings, lists, tuples, dictionaries, and more, each serving specific purposes in storing and manipulating data.

Numeric data types such as integers and floating-point numbers allow for mathematical operations like addition, subtraction, multiplication, and division, using arithmetic operators (+, -, *, /). Python's flexibility with arithmetic operations extends to support complex numbers and floor division (//) for obtaining integer results of divisions.

Strings, represented within single or double quotes, are sequences of characters and support various manipulations via string operators like concatenation (+) or repetition (*). Python's string manipulation capabilities include slicing, indexing, and a wide range of methods for formatting, searching, and modifying strings.

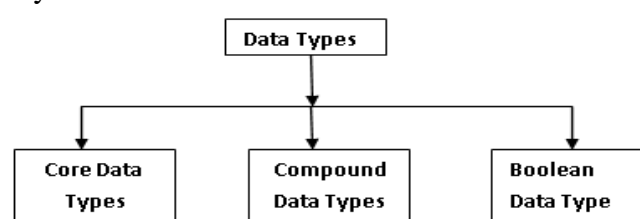
Python incorporates a variety of operators beyond arithmetic and string operations. Logical operators (and, or, not) facilitate boolean operations and comparisons, while relational operators (>, <, ==, !=, >=, <=) help evaluate conditions and comparisons between variables or data.

A. Data Types

Python Data Types are classified as “*Core Data Types*”, “*Compound Data Types*” and “*Boolean Data Type*”. Under the Core data Types we have Numbers and Strings. Under the Compound Data Types, we have Lists, Tuples, Dictionary, and set etc.

The Numbers and Strings represent the Numeric and Textual values, respectively. Under the Boolean Data, a variable can contain any one of the two values: True or False.

Fig: 1.6: Python Data Type



- **Core Data Types**

Numbers: Integers, floating point numbers and complex numbers are fall under the Number category. These are defined as “int”, “float”, and “complex” class in Python. We can use the type function to know type or class of a variable. Integers are whole numbers without any decimal point. It can be of any length, it is only limited by the memory variable. A floating point number is including decimal points and accurate up to 15 decimal points. For example, 1 is an Integer. 1.0 is floating point number. The complex numbers are written in form, x+yj, where x is the Real part and y is the imaginary part.

```
# Integers
num1 = 10
num2 = -25
num3 = 0

# Floating-point numbers
float_num1 = 3.14
float_num2 = -0.5
float_num3 = 2.0

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

# Arithmetic operations
result_sum = num1 + num2
result_product = float_num1 * float_num2
result_division = num1 / num2
result_power = num1 ** 2

# Displaying results
print("Result of integers:", result_sum)
print("Result of floats:", result_product)
print("Result of integers:", result_division)
print("Result of an integer:", result_power)

# Displaying complex numbers
print("Complex Number 1:", complex_num1)
print("Complex Number 2:", complex_num2)
```

Strings: Strings are sequences of characters enclosed within single (' ') or double (" ") quotes in Python. Multiline strings are denoted with Triple Quotes, ‘ ‘ ‘ or “ “ “.

Example :

```
# Single line strings
str1 = 'Hello, World!'
str2 = "Python is awesome."

# Multiline strings using triple quotes
multiline_str = '''This is a multiline string
It can span across multiple lines
And contain various characters such as !@#$%'''
```

We can get the character at the specified position by its index. We can also get the substring from the index by specifying the range of the substring. The index always starts from zero. See the following example:

```
s="Hello Python Programmer"
print(s)
print(s[2]) # obtaining the character at position 2
print(s[3:7]) # retrieving substring using the from index 3 to
7
```

```
OutPut
Hello Python Programmer
l
lo P
```

• Boolean Data Type

A Boolean variable can reference one of two values: True or False. Boolean variables are commonly used as flags, which indicate whether specific conditions exist.

Example:

```
b = False
print("The value of b is:", b)
    string formatting
print("class of b is:", type(b))
    using string formatting
b = True
print("The value of b is:", b)
    b using string formatting
print("class of b is:", type(b))
```

```
OutPut
The value of b is: False
class of b is: <class 'bool'>
The value of b is: True
class of b is: <class 'bool'>
```

• Compound Data Types

Compound data types in Python are data structures that can hold multiple elements of different or similar data types within a single variable.

These data types are essential for organizing and managing complex collections of data. The primary compound data types in Python are Lists, Tuples, Dictionaries, and Sets.

List: Is the Order collection of different types of items. All the items need not be of same type. These are mutable, that means the list can be modified using the index of the item or using the predefined methods such as “*append()*”, “*sort()*”, “*pop()*”, and “*reverse()*”. The list is created with square brackets [].

For example

```
# Creating a list
my_list = [1, 'hello', 3.14, True]
# Accessing elements in the list
print(my_list[0]) # Output: 1
print(my_list[1]) # Output: 'hello'
# Modifying elements in the list
my_list[2] = 'world'
print(my_list) # Output: [1, 'hello', 'world', True]
```

Tuples: It is roughly like a list, but is immutable, that means, the tuple cannot be modified once it is created. We can get the index of an item. We can also get the frequency of the item using the function like “*index()*” and “*count()*”.. This is created with parentheses ().

Example:

```
# Creating a list
my_list = [1, 'hello', 3.14, True]
# Accessing elements in the list
print(my_list[0]) # Output: 1
print(my_list[1]) # Output: 'hello'
# Modifying elements in the list
my_list[2] = 'world'
print(my_list) # Output: [1, 'hello', 'world', True]
```

Dictionaries: are unordered collections of data stored as key-value pairs within curly braces {}. They are mutable and allow for fast retrieval of values based on their associated keys.

```
# Creating a dictionary
my_dict = {'name': 'Alice', 'age': 30, 'is_student': False}
# Accessing values using keys
print(my_dict['name']) # Output: 'Alice'
# Modifying values in the dictionary
my_dict['age'] = 25
print(my_dict) # Output: {'name': 'Alice', 'age': 25, 'is_student': False}
```

Sets: are unordered collections of unique elements enclosed within curly braces {}. They do not allow duplicate values and support set operations like union, intersection, and difference.

```
# Creating a set
my_set = {1, 2, 3, 4, 4, 5}

# Displaying the set (automatically removes duplicates)
print(my_set) # Output: {1, 2, 3, 4, 5}
```

- **Type conversion**

Type conversion, also known as type casting, refers to the process of converting one data type into another in Python. This conversion is essential when you need to perform operations or functions that involve different data types. Python provides built-in functions to facilitate type conversion.

```
# Integer to float conversion
num_int = 10
num_float = float(num_int)
print(num_float) # Output: 10.0

# Float to integer conversion
num_float = 3.14
num_int = int(num_float)
print(num_int) # Output: 3

# String to integer conversion
str_num = "25"
int_num = int(str_num)
print(int_num) # Output: 25

# String to float conversion
str_float = "3.14"
float_num = float(str_float)
print(float_num) # Output: 3.14

# Integer to string conversion
num = 100
str_num = str(num)
print(str_num) # Output: '100'

# Float to string conversion
float_num = 3.14
str_float = str(float_num)
print(str_float) # Output: '3.14'

# Type conversion in operations
num1 = 10
num2 = "20"
result = num1 + int(num2) # Convert string to integer
print(result) # Output: 30
```


B. Operators

Operators in programming are symbols or constructs that perform operations on operands or variables. They allow for various computations, comparisons, and logical operations within a program. Python supports a wide range of operators classified into different categories:

- Arithmetic Operators
- Comparison (Relational) Operators
- Bitwise Operators
- Logical Operator
- Assignment Operators
- Membership Operators
- Identity Operators

Understanding and utilizing operators effectively is crucial for performing various operations, comparisons, and logical evaluations within Python programs.

• Arithmetic Operators

Arithmetic operators perform mathematical operations such as addition, subtraction, multiplication, division, modulus, and exponentiation.

```
# Arithmetic operators
a = 10
b = 3
addition = a + b # Addition
subtraction = a - b # Subtraction
multiplication = a * b # Multiplication
division = a / b # Division
modulus = a % b # Modulus (remainder of division)
exponentiation = a ** b # Exponentiation
print("Addition:", addition) # Output: 13
print("Subtraction:", subtraction) # Output: 7
print("Multiplication:", multiplication) # Output: 30
print("Division:", division) # Output: 3.3333333333333335
print("Modulus:", modulus) # Output: 1
print("Exponentiation:", exponentiation) # Output: 1000
```

- **Comparison (Relational) Operators**

Comparison operators compare values and return True or False based on the comparison

```
# Comparison operators
x = 5
y = 8
print("Greater than:", x > y) # Output: False
print("Less than:", x < y) # Output: True
print("Equal to:", x == y) # Output: False
print("Not equal to:", x != y) # Output: True
print("Greater than or equal to:", x >= y) # Output: False
print("Less than or equal to:", x <= y) # Output: True
```

- **Bitwise Operators**

Bitwise operators perform operations at the binary level on operands. The operators operate bit by bit. For example, $x=2$ (binary value is 10) and $y=7$ (Binary value is 111). The binary equivalent of the decimal values of x and y will be 10 and 111 respectively.

```
# Bitwise operators
num1 = 10
num2 = 4
bitwise_and = num1 & num2 # Bitwise AND
bitwise_or = num1 | num2 # Bitwise OR
bitwise_xor = num1 ^ num2 # Bitwise XOR
bitwise_not = ~num1 # Bitwise NOT
print("Bitwise AND:", bitwise_and) # Output: 0
print("Bitwise OR:", bitwise_or) # Output: 14
print("Bitwise XOR:", bitwise_xor) # Output: 14
print("Bitwise NOT:", bitwise_not) # Output: -11
```

- **Logical Operator**

Logical operators perform logical operations on Boolean values and return a Boolean outcome. There are three logical operators: **and**, **or**, and **not**. The semantics (meaning) of these operators is similar to their meaning in English. For example, $x > 0$ and $x < 10$ is true only if x is greater than 0 and less than 10. $n \% 2 == 0$ or $n \% 3 == 0$ is true if either (or both) of the conditions is true, that is, if the number is divisible by 2 or 3. Finally, the not operator negates a boolean expression, so $\text{not}(x > y)$ is true if $(x > y)$ is false, that is, if x is less than or equal to y .

```
# Logical operators
p = True
q = False
print("Logical AND:", p and q) # Output: False
print("Logical OR:", p or q) # Output: True
print("Logical NOT:", not p) # Output: False
```

- **Assignment Operators**

Assignment operator is used to assign values to the variable. For example, $x=5$ is simple assignment operator, that assigns value 5 to the variable x. There are various compound operators in python like $a+=5$, which adds value 5 to the variable and later assigns it to variable a. This expressions is equivalent to $a=a+5$. The same assignment operator can be applied to all expressions that contain arithmetic operators such as, $*=$, $/=$, $-=$, $**=$, $\%=$ etc.

```
# Assignment operators
x = 5 # Assigns the value 5 to x
y = 10
y += 5 # Equivalent to y = y + 5 (Addition assignment)
print("x after assignment:", x) # Output: 5
print("y after addition assignment:", y) # Output: 15
```

- **Membership Operators**

These operators are used to test whether a value or operand is there in the sequence such as list, string, set, or dictionary. There are two membership operators in python: in and not in. In the dictionary we can use to find the presence of the key, not the value.

```
# Membership operators
my_list = [1, 2, 3, 4, 5]
print("Check if 3 is in the list:", 3 in my_list) # Output: True
print("Check if 6 is not in the list:", 6 not in my_list) # Output:
True
```

- **Identity Operators**

These are used to check if two values (variable) are located on the same part of the memory. If the x is a variable contain some value, it is assigned to variable y. Now both variables are pointing (referring) to the same location on the memory as shown in the example program.


```
# Identity operators
x = 5
y = 5
z = [1, 2, 3]
w = [1, 2, 3]
print("Check if x is y:", x is y) # Output: True (both x and y point
    to the same memory location for integer 5)
print("Check if z is w:", z is w) # Output: False (z and w are different objects, even with
    the same values)
print("Check if z is not w:", z is not w) # Output: True (z and w are
    not the same object)
```

C. Expression

In programming, an expression is a combination of values, variables, operators, and function calls that are evaluated to produce a single value. Python supports various types of expressions, including arithmetic expressions, string expressions, logical expressions, and more. Here are examples of different types of expressions along with their outputs:

- **Arithmetic Expressions:**

Arithmetic expressions involve mathematical operations on numbers. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, **2*(3-1)** is 4, and **(1+1)**(5-2)** is 8.

You can also use parentheses to make an expression easier to read, as in **(minute*100)/60**, even though it doesn't change the result. **E**xponentiation has the next highest precedence, so **2**1+1** is 3 and not 4, and **3*1**3** is 3 and not 27.

Multiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So **2*3-1** yields 5 rather than 4, and **2/3-1** is -1, not 1 (re- member that in integer division, **2/3=0**).

Operators with the same *precedence are evaluated from left to right*. If the **minute=59**, then in the expression **minute*100/60**, the multiplication happens first, yielding **5900/60**, which in turn yields 98. If the operations had been evaluated from *right to left*, the result would have been **59*1** which is wrong. If in doubt, use parentheses.

```
# Arithmetic expression
result = 5 + 3 * 2 # Addition and multiplication
print("Arithmetic Expression Result:", result) # Output: 11
```

- **String Expressions:**

String expressions involve concatenation and manipulation of strings.

```
# String expression
str1 = "Hello, "
str2 = "World!"
combined_str = str1 + str2 # String concatenation
print("String Expression Result:", combined_str) # Output: 'Hello,World!'
```

- **Logical Expressions:**

Logical expressions involve boolean operations that result in a Boolean value (True or False).

```
# Logical expression
x = 5
y = 10
is_greater = x > y # Greater than comparison
print("Logical Expression Result:", is_greater) # Output: False
```

- **Function Call Expressions:**

Function call expressions involve calling a function and using its return value in an expression.

```
# Function call expression
def square(num):
    return num ** 2
result = square(4) + square(3) # Using function return values in an
expression
print("Function Call Expression Result:", result) # Output: 25 (16 + 9)
```

- **Compound Expressions:**

Compound expressions involve combining multiple expressions within a larger expression.

```
# Compound expression
x = 10
y = 5
compound_expr = (x * 2) + (y / 2) # Combination of arithmetic
operations
print("Compound Expression Result:", compound_expr) # Output: 25.0
```

1.4 Sequence, Selection and Iteration constructs

In Python, sequence, selection, and iteration constructs are fundamental programming concepts that control the flow of a program, allowing developers to organize code, make decisions, and execute repetitive tasks efficiently.

A. Sequence:

Sequences are ordered collections of elements. In Python, lists, tuples, and strings are common sequence types. Lists are mutable and allow modifications; tuples are immutable and cannot be changed after creation, while strings are sequences of characters.

```
# List sequence example
my_list = [1, 2, 3, 4, 5]
print("List:", my_list) # Output: [1, 2, 3, 4, 5]
# Tuple sequence example
my_tuple = (10, 20, 30)
print("Tuple:", my_tuple) # Output: (10, 20, 30)
# String sequence example
my_string = "Hello"
print("String:", my_string) # Output: Hello
```

B. Selection (Condition Statement)

Selection constructs, represented by conditional statements like `if`, `elif` (else if), and `else`, allow branching based on conditions. These conditions evaluate as `True` or `False`, and code blocks are executed based on the outcome.

The "if statement" in Python is a fundamental control structure that allows conditional execution of code blocks based on certain conditions. It enables developers to create decision-making processes within their programs. The syntax involves using the `if` keyword followed by a condition, and if that condition evaluates to `True`, the code block associated with the `if` statement gets executed.

Syntax for If Statement

```
if boolean_expression:
    statement(s) # block of statements inside if else:
statement(s)    # block of statements inside else
```

- **Basic If Statement:**

A basic if statement checks a condition and executes a block of code if the condition is True.

```
# Basic if statement example
x = 10
if x > 5:
    print("x is greater than 5")# Output: x is greater than 5
```

- **If-Else Statement:**

The if-else statement allows for branching, where if the condition is True, one block of code executes, and if the condition is False, another block executes.

```
# If-else statement example
y = 3
if y % 2 == 0:
    print("y is even")
else:
    print("y is odd")# Output: y is odd
```

- **If-Elif-Else Statement:**

The if-elif-else statement allows multiple conditions to be checked sequentially. When the first condition is True, the associated block of code executes, and the rest of the conditions are skipped.

```
# If-elif-else statement example
number = 7
if number < 0:
    print("Number is negative")
elif number == 0:
    print("Number is zero")
else:
    print("Number is positive")# Output: Number is positive
```

- **Nested If Statements:**

If statements can be nested inside other if statements, allowing for more complex decision-making.

```
# Nested if statement example
a = 10
b = 5
if a > 0:
    if b > 0:
        print("Both a and b are positive")
    else:
        print("a is positive but b is not")# Output: Both a and b are positive
```

C. Iteration (Loops):

Iteration constructs, such as for loops and while loops, enable the execution of a block of code repeatedly. **for** loops iterate over a sequence, executing the block of code for each element, while **while** loops execute as long as a certain condition remains True.

- **for statement**

For loops iterate over a given sequence or list. It is helpful in running a loop on each item in the list. The general form of “for” loop in Python will be as follow:

```
for variable in [value1, value2, etc.]: # list statement1
statement2
..... Statement N
```

Here variable is the name of the variable. And **in** is the keyword. Inside the square brackets a sequence of values are separated by comma. In Python, a comma-separated sequence of data items that are enclosed in a set of square brackets is called a **list**. The list is created with help of [] square brackets. The list also can be created with help of tuple. We can also use **range()** function to create the list.

The general form of the range() function will be as follow: **range(number)** –ex: range(10) –It takes all the values from 0 to 9. **range(start,stop, interval_size)** –ex: range(2,10,2)–It lists all the numbers such as 2,4,6,8.**Range(start,stop)**–ex: range(1,6), lists all the numbers from 1to 5, but not 6. Here, by default the interval size is 1.

```
# List of numbers
numbers = [5, 10, 15, 20, 25]
# Initialize sum
total = 0
# Iterate through the list using a for loop and calculate the sum
for num in numbers:
    total += num
# Display the sum
print("The sum of all items in the list is:", total)# output '75'
```

Example:

```
print("The Number Square")
print("-----")
for x in range(1, 6): # list
    print(f"The square of {x} is {x**2}")
print("-----")
```

```
The Number Square
-----
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
-----
```

- **while statement**

While loops repeat as long as a certain boolean condition is met. The block of statements is repeatedly executed as long as the condition is evaluated to True. The general form of while will be as follow:

```
while condition:
    statement1 statement2
    ..... statementN
```

```
s = 0
n = int(input("Enter any number: "))
while n > 0:
    s = s + n
    n = n - 1
print("The sum is:", s)
```

Output

```
Enter any number: 5
The sum is: 15
```


- **ELSE for a loop**

Loop statements may have an else clause. It is executed when the loop terminates through exhaustion of the list (with for loop) and executed when the condition becomes false (with while loop), But not when the loop is terminated by a break statement.

Example: printing all primes numbers up to 1000

```
for n in range(2, 1000):
    for x in range(2, n):
        if (n % x) == 0:
            break
    else:
        print(n)
```

<pre># List of numbers numbers = [1, 2, 3, 4, 5] # For loop to print even and odd numbers print("Even and Odd numbers using for loop:") for num in numbers: if num % 2 == 0: print(f"{num} is even.") else: print(f"{num} is odd.") # While loop to display a countdown countdown = 5 print("\nCountdown using while loop:") while countdown > 0: print(countdown) countdown -= 1 print("Blast off!")</pre>	<pre>Even and Odd numbers using for loop: 1 is odd. 2 is even. 3 is odd. 4 is even. 5 is odd. Countdown using while loop:5 4 3 2 1 Blast off!</pre>
--	--

- **Jump statement**

In Python, jump statements alter the normal flow of control in a program. Two essential jump statements are break and continue.

➤ **Break Statement:**

The break statement is used to terminate the execution of a loop prematurely based on a certain condition. It is commonly used within loops to immediately exit the loop when a specific condition is met, regardless of whether the loop has finished iterating through all elements.

➤ Continue Statement:

The continue statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration. It allows you to skip specific iterations based on a condition without completely exiting the loop.

```
# Example illustrating break and continue statements
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Break example:")
for num in numbers:
    if num == 6:
        break # Terminate the loop when num equals 6
    print(num)
# Output: 1 2 3 4 5
print("\nContinue example:")
for num in numbers:
    if num % 3 == 0:
        continue # Skip current iteration when num is divisible by 3
    print(num)
# Output: 1 2 4 5 7 8 10
```

1.5 Modular Programming Approach

Modular programming is a software design technique, which is based on the general principal of modular design. Modular design is an approach which has been proven as indispensable in engineering even long before the first computers. Modular design means that a complex system is broken down into smaller parts or components, i.e. modules. These components can be independently created and tested. In many cases, they can be even used in other systems as well.

There is hardly any product nowadays, which doesn't heavily rely on modularization, like cars and mobile phones. Computers belong to those products which are modularized to the utmost. So, what's a must for the hardware is an unavoidable necessity for the software running on the computers.

If you want to develop programs which are readable, reliable and maintainable without too much effort, you have to use some kind of modular software design. Especially if your application has a certain size. There exists a variety of concepts to design software in modular form. Modular programming is a software design technique to split your code into separate parts. These parts

are called modules. The focus for this separation should be to have modules with no or just few dependencies upon other modules. In other words: Minimization of dependencies is the goal. When creating a modular system, several modules are built separately and more or less independently. The executable application will be created by putting them together.

A. Importing Modules

So far we haven't explained what a Python module is. In a nutshell: every file, which has the file extension .py and consists of proper Python code, can be seen or is a module! There is no special syntax required to make such a file a module. A module can contain arbitrary objects, for example files, classes or attributes. All those objects can be accessed after an import. There are different ways to import modules. We demonstrate this with the math module:

import math

The module math provides mathematical constants and functions, e.g. π (math.pi), the sine function (math.sin()) and the cosine function (math.cos()). Every attribute or function can only be accessed by putting "math." in front of the name:

```
import math
print(math.pi) # Output: Value of Pi (approximately 3.14159)
print(math.cos(math.pi)) # Output: -1.0 (cosine of Pi radians)
print(math.cos(math.pi / 2)) # Output: 6.123233995736766e-17 (cosine of Pi/2 radians)
```

It's possible to import more than one module in one import statement. In this case the module names are separated by commas:

import math, random

Import statements can be positioned anywhere in the program, but it's good style to place them directly at the beginning of a program. If only certain objects of a module are needed, we can import only those:

from math import sin, pi

The other objects, e.g. cos, are not available after this import. We are capable of accessing sin and pi directly, i.e. without prefixing them with "math." Instead of explicitly importing certain objects from a module, it's also possible to import everything in the namespace of the importing module. This can be achieved by using an asterisk in the import:

Page 33 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

```
from math import *
result = sin(3.01) + tan(cos(2.1)) + e
print(result) # Output: The sum of sin(3.01), tan(cos(2.1)), and the
               value of 'e'
print(e) # Output: The value of the mathematical constant 'e'
```

B. Designing and Writing Modules

But how do we create modules in Python? A module in Python is just a file containing Python definitions and statements. The module name is moulded out of the file name by removing the suffix .py. For example, if the file name is fibonacci.py, the module name is fibonacci.

Let's turn our Fibonacci functions into a module. There is hardly anything to be done; we just save the following code in the file fibonacci.py:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
def ifib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

The newly created module "fibonacci" is ready for use now. We can import this module like any other module in a program or script. We will demonstrate this in the following interactive Python shell:

```
import fibonacci
fibonacci.fib(7) #Output : 13
```

C. Importing Names from a Module Directly

Names from a module can directly be imported into the importing module's symbol table:

```
from fibonacci import fib, ifib
ifib(500)
```

1.6 Arrays and Arrays of objects

In Python, arrays and arrays of objects are fundamental data structures used to store collections of items. The primary difference between arrays and lists in Python is that arrays can only store elements of the same data type, whereas lists can contain elements of different data types. Arrays are part of the array module and require a specific data type declaration upon creation, such as integers, floats, or characters.

To work with arrays of objects in Python, one can utilize arrays from the array module or create arrays of custom objects using classes. For instance, a class definition can be created to represent objects with various attributes, and an array can be instantiated to store instances of these objects.

Arrays are handled by a Python object-type module **array**. Arrays behave like lists except for the fact that the objects they contain are constrained by their types and most importantly, they are faster and use lesser memory space.

- **Array Syntax**

Elements: Are items stored in the array.

Index: Represents the location where an element is stored in an array.

Length: Is the size of the array or the number of indexes the array possesses.

Indices: Is the index map of the array value stored in the object.

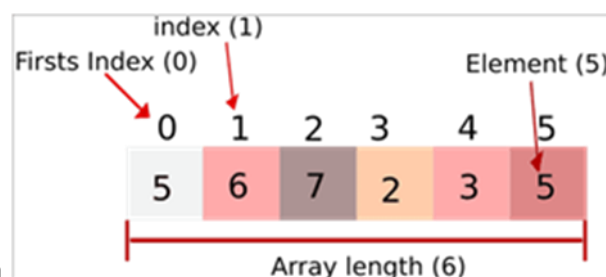


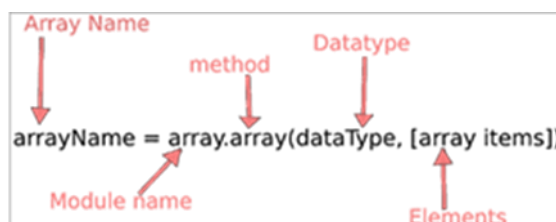
Fig: 1.6: Array Representation

The above figure displays an array with a length of **6**, and the elements of the array are **[5, 6, 7, 2, 3, 5]**. The index of the array always begins with **0**(zero-based) for the first element, then **1** for the next element, and so on. They are used to access the elements in an array.

- **Python Built-in Array Module**

There are many other built-in modules in Python which you can read more about from [here](#). A module is a Python file containing Python definitions and statements or functions. These statements are used by calling them from the module when the module is imported into another Python file. The module used for the array is called an **array**.

The array module in Python defines an object that is represented in an array. This object contains basic data types such as integers, floating points, and characters. Using the array module, an array can be initialized using the following syntax.



Syntax

```
arrayName = array.array(dataType, [array items])
```

Example: Printing an array of values with type code, **int**

```
import array # import array module
myarray = array.array('i', [5, 6, 7, 2, 3, 5])
print(myarray) # Output: array('i', [5, 6, 7, 2, 3, 5])
```

The above example is explained below;

1. The name **arrayName** is just like naming any other variable. It can be anything that abides by Python naming conventions, in this case, **myarray**.
2. The first **array** in **array.array** is the module name that defines the **array()** class. It must be imported before used. The first line of code does just that.
3. The second **array** in **array.array** is the class called from the **array** module which initializes the array. This method takes two parameters.
4. The first parameter is **dataType** which specifies the data type used by the array. In **example 1**, we used the data type '**i**' which stands for **signed int**.
5. The second parameter used by the array method specifies the elements of the array provided as an iterable like **list**, **tuple**. In **example 1** a list of integers was provided.

- **Array Basic Operation**

Operations that can be performed on its object are **Traverse, Insertion, Deletion, Search, and Update**.

1. Traversing an Array

Just like lists, we can access elements of an array by **indexing, slicing** and **looping**.

- **Indexing Array**

An array element can be accessed by indexing, similar to a list i.e. by using the location where that element is stored in the array. The index is enclosed within square brackets [], the first element is at index **0**, next at index **1** and so on.

```
from array import array # import array class from array module
```

Example 3: Access elements of an array by indexing.

```
from array import array
a = array('i', [4, 5, 6, 7]) # create an array of signed integers
print(a[0]) # access at index 0, first element 4
print(a[3]) # access at index 3, 4th element 7
print(a[-1]) # access at index -1, last element, same as a[len(a) - 1]
               which is 7
print(a[9])
```

Just like lists, we can access elements of an array using the slicing operator [start : stop : stride] . To know more about slicing and how it applies to strings, check out the tutorial **Python String Operators and Methods**.

Example: Access elements of an array by slicing.

```
from array import array # Import the array class from the array module
a = array('f', [4, 3, 6, 33, 2, 8, 0]) # Create an array of floats
print(a) # Output: array('f', [4.0, 3.0, 6.0, 33.0, 2.0, 8.0, 0.0])
print(a[0:4]) # Slice from index 0 to index 3 (4 is exclusive)
               # Output: array('f', [4.0, 3.0, 6.0, 33.0])
print(a[2:4]) # Slice from index 2 to index 3 (4 is exclusive)
               # Output: array('f', [6.0, 33.0])
print(a[:2]) # Slice from start to end while skipping every second
               element
               # Output: array('f', [4.0, 6.0, 2.0, 0.0])
print(a[::-1]) # Slice from start to end in reverse order
               # Output: array('f', [0.0, 8.0, 2.0, 33.0, 6.0, 3.0, 4.0])
```

- **Looping Array**

Looping an array is done using the **for loop**. This can be combined with slicing as we saw earlier or with built-in methods like **enumerate()**.

Example: Access elements of array by looping.

```
from array import array # import array class from array module
# define array of floats
a = array('f', [4,3,6,33,2,8,0])
print("Normal looping")
- for i in a:
    print(i)
    print("Loop with slicing")
- for i in a[3:]:
    print(i)
    print("loop with method enumerate() and slicing")
- for i in enumerate(a[1::2]):
    print(i)
```

2. Inserting into an Array

In Python, inserting elements into an array can be performed using various methods provided by the array module. The `insert()` method, while available for Python lists, isn't natively available for arrays. Instead, arrays offer certain functionalities for **appending and extending**, facilitating the addition of elements at the end of the array. However, for inserting elements at specific positions within an array, one common approach involves using the `insert()` method available for Python lists and then converting the modified list back to an array, if necessary.

```
from array import array
# Create an array of integers
int_array = array('i', [1, 2, 3, 4, 5])
# Convert the array to a list for inserting elements
int_list = list(int_array)
# Insert an element at a specific index using list's insert() method
index_to_insert = 2
element_to_insert = 10
int_list.insert(index_to_insert, element_to_insert)
# Convert the modified list back to an array
int_array = array('i', int_list)
# Display the updated array
print("Updated Array after Insertion:", int_array)
```


In the above example:

- Initially, an array of integers (**int_array**) is created using the **array()** function from the **array** module.
- To perform an insertion, the array is converted to a list (**int_list**) using Python's **list()** function.
- Using the **insert()** method available for lists, an element (here, **10**) is inserted at a specific index (**2**) within the list.
- Finally, the modified list is converted back to an array using the **array()** function, resulting in the updated **int_array**.
- The output showcases the updated array after the insertion operation.

A. Appending Elements:

The **append()** method in Python is used to add a single element at the end of an array or list.

```
from array import array
# Create an array of integers
int_array = array('i', [1, 2, 3])
# Append a single element to the array
int_array.append(4)
int_array.append(5)
# Display the updated array
print("Updated Array after Appending:", int_array)
```

In this example:

- An array of integers (**int_array**) is initially created using the **array()** function from the **array** module.
- The **append()** method is used to add elements **4** and **5** one by one at the end of the array.
- The output showcases the updated array after the appending operations.

B. Extending Elements:

The `extend()` method in Python is utilized to append multiple elements (another array, list, or iterable) at the end of an array or list.

```
from array import array
# Create two arrays of integers
int_array_1 = array('i', [1, 2, 3])
int_array_2 = array('i', [4, 5, 6])
# Extend int_array_1 with elements from int_array_2
int_array_1.extend(int_array_2)
# Display the updated array
print("Updated Array after Extending:", int_array_1)
```

In this example:

- Two arrays of integers (**int_array_1** and **int_array_2**) are created using the **array()** function from the **array** module.
- The **extend()** method is used on **int_array_1** to append all elements from **int_array_2** at the end of **int_array_1**.
- The output showcases the updated **int_array_1** after the extension operation

Self-Check -1

I. State whether the following statement is TRUE or FALSE

1. Objects in OOP can interact with each other by directly accessing and modifying each other's internal data and code.
2. Encapsulation in OOP refers to the bundling of data and methods within a single unit or object.
3. Inheritance allows new classes to inherit properties and behaviours only from one specific parent class.
4. Polymorphism in OOP is solely achieved through method overriding.

II. Choose the best answer

1. Which OOP principle refers to the act of representing essential features without including background details?
 - A) Encapsulation
 - B) Polymorphism
 - C) Inheritance
 - D) Data Abstraction
2. Which concept allows creating new classes based on existing classes, inheriting their attributes and behaviours?
 - A) Encapsulation
 - B) Data Abstraction
 - C) Inheritance
 - D) Polymorphism
3. Which of the following best describes an object in OOP?
 - A) A variable of any data type
 - B) A variable of type 'class'
 - C) A collection of methods only
 - D) A collection of attributes only
4. What is the primary goal of encapsulation in OOP?
 - A) Reducing code complexity
 - B) Hiding internal implementation details
 - C) Promoting code reuse
 - D) All of the above
5. Which OOP principle allows objects of different classes to be treated as objects of a common superclass?
 - A) Inheritance
 - B) Encapsulation
 - C) Polymorphism
 - D) Data Abstraction

Operation sheet 1.1: Installation of Python Program for windows computer

Operation Title: Installing Python on windows computer

Purpose: To Install and run Python on windows computers

Equipment Tools and Materials:

- Computer
- Internet connection or Latest version of Python Installer

Steps in doing the task

1. Go to the official Python website: python.org. (if you have Python installer skip to step 5)
2. Click on the "Downloads" tab.
3. Choose the latest release within Python 3.
4. Click on the "Download Python 3.x.x" button.
5. Double-click on installer file to start the installation process.
6. Check the box that says "Add Python 3.x to PATH." This option will allow you to use Python from the command line easily.
7. Click on "Install Now" to start the installation. You can customize the installation by clicking on "Customize installation" if needed.
8. The installer will begin installing Python on your system. This process might take a few moments.



9. Once the installation is complete, you will see a screen showing that Python has been installed successfully.

Quality Criteria: Open a command prompt by searching for "Command Prompt" or "cmd" in the Start menu. Type **python --version** or **python -V** and press Enter. This command will display the installed Python version.

Page 42 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

Operation sheet 1.2: Installation of VS Code IDE

Operation Title: Installing VS code on windows computer

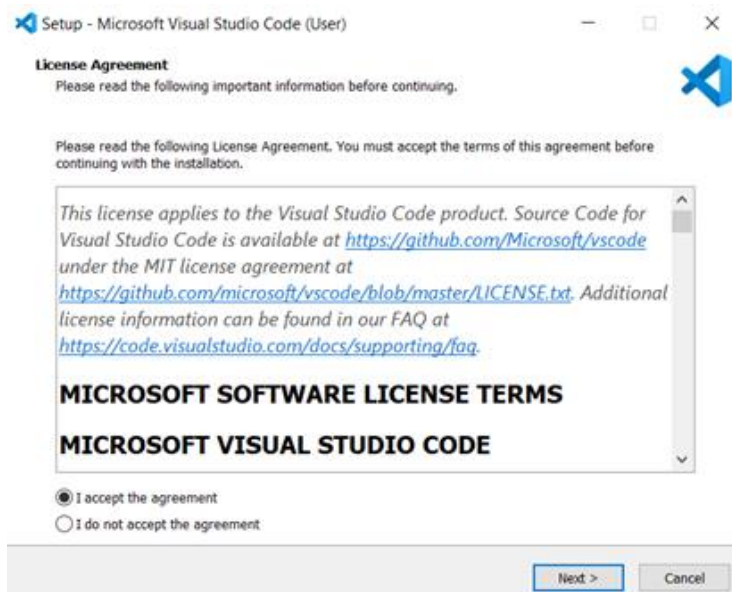
Purpose: To install VS code

Equipment Tools and Materials:

- Computer
- Internet connection or Latest version of VS code Installer

Steps in doing the task

1. Visit the official Visual Studio Code website: code.visualstudio.com.
2. Click on the "Download" button
3. Once the installer file is downloaded, double-click on it to start the installation process.(if you have .exe file on your computer start from this step)
4. Accept the license agreement, choose the destination folder, and select additional components if needed.



5. Click "Next" or "Install" to proceed with the default installation settings.

Quality Criteria: Once the installation is complete, launch Visual Studio Code by clicking on the desktop shortcut or searching for "Visual Studio Code" in your applications

Page 43 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

Operation sheet 1.3: Set Up VS Code for Python Development

Operation Title: Setting up VS Code for Python Development

Purpose: To run Python program using VS code

Equipment Tools and Materials:

- Computer
- Internet connection

Steps in doing the task

1. Open Visual Studio Code.
2. Go to the Extensions view by clicking on the square icon on the left sidebar or using the shortcut **Ctrl+Shift+X**
3. Search for "Python" in the Extensions Marketplace.
4. Click "Install" for the official Python extension offered by Microsoft.
5. Open a new or existing Python file (.py) in VS Code.
6. At the lower-left corner, click on the Python interpreter version (e.g., "Select Python Interpreter").
7. Choose the Python interpreter you installed earlier. If not detected automatically, you may need to locate the Python interpreter path (e.g., **python.exe** or **python3**) manually.
8. Create a new Python file or open an existing one within VS Code.



9. Write your Python code in the editor.
10. To run the code, right-click in the editor and select "Run Python File in Terminal" or use the shortcut **Ctrl+Alt+N**.

Quality Criteria: Open VS code and start writing python code.

Page 44 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

Operation sheet 1.4: Python Selection

Operation Title: Selection

Purpose: To demonstrate how simple selection code is written

Steps in doing the task

1. Define a variable number and assigning it a value of 10. You can do this by typing
number = 10

```
# Define a variable
number = 10
```

2. Next, use an if statement to check if the value of number is greater than 5. You can do this by typing if number > 5:
3. Write a code If the value of number is greater than 5, print the message “The number is greater than 5.

```
# Check if the number is greater than 5
if number > 5:
    print("The number is greater than 5.")
else:
```

4. If the value of number is not greater than 5, print the message “The number is not greater than 5.”

```
else:
    print("The number is not greater than 5.")
```

5. Finally run the code as follow

```
# Define a variable
number = 10

# Check if the number is greater than 5
if number > 5:
    print("The number is greater than 5.")
else:
    print("The number is not greater than 5.")
```

Quality Criteria: The Output must say “The number is greater than 5.”

Operation sheet 1.5: Python Sequence

Operation Title: Sequence

Purpose: To demonstrate how simple sequence code is written

Steps in doing the task

1. Define a list of numbers by enclosing them in square brackets and separating them with commas.

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

2. Use a for loop to iterate through the list and print each number.

```
# Iterate through the list and print each number
for number in numbers:
    print(number)
```

3. To perform a calculation using the numbers in the list, use the sum() function.

```
# Perform a calculation using the numbers in the list
total = sum(numbers)
print("The sum of the numbers is:", total)
```

4. Finally write the whole code as follow and run it.

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
total = sum(numbers)
print("The sum of the numbers is:", total)
```

Quality Criteria: The output is look like the following.

```
1
2
3
4
5
The sum of the numbers is: 15
```


Operation sheet 1.6: Python Iteration

Operation Title: Iteration

Purpose: To demonstrate how simple iteration code is written

Steps in doing the task

1. Write a code that define a list of fruits: fruits = ["apple", "banana", "orange", "grape"]

```
# Define a list of fruits
fruits = ["apple", "banana", "orange", "grape"]
```

2. Iterate through the list and print each fruit

```
# Iterate through the list and print each fruit
for fruit in fruits:
    print("I like", fruit)
```

3. Use range() for iteration

```
# Using range() for iteration
print("Counting from 1 to 5:")
for number in range(1, 6):
    print(number)
```

4. Finally write the whole code as follows

```
fruits = ["apple", "banana", "orange", "grape"]
for fruit in fruits:
    print("I like", fruit)
print("Counting from 1 to 5:")
for number in range(1, 6):
    print(number)
```

Quality criteria: The output should look like the following

```
I like apple
I like banana
I like orange
I like grape
Counting from 1 to 5:
1
2
3
4
5
```

Operation sheet 1.7: Python Data types, Operators and Expression

Operation Title: Data types, Operators and Expression

Purpose: To demonstrate arithmetic operations using different data types and operators

Steps in doing the task

1. Write a code that define variables with different data types

```
# Variables with different datatypes
integer_number = 10
float_number = 3.5
string_text = "Hello"
boolean_value = True
```

2. Use arithmetic operators to perform calculations

```
# Arithmetic operators
addition_result = integer_number + float_number
subtraction_result = float_number - integer_number
multiplication_result = integer_number * 5
division_result = float_number / 2
exponential_result = integer_number ** 2
modulo_result = 17 % 3 # Modulo operator (%) returns the
                        remainder
```

3. Use variables and operators to create expressions

```
# Expressions using variables and operators
expression_1 = integer_number + 20 * 5
expression_2 = (integer_number + float_number) * 2
```

4. Write a code that can display the results

```
# Displaying results
print("Addition:", addition_result)
print("Subtraction:", subtraction_result)
print("Multiplication:", multiplication_result)
print("Division:", division_result)
print("Exponential:", exponential_result)
print("Modulo:", modulo_result)
print("Expression 1:", expression_1)
print("Expression 2:", expression_2)
```

5. Write the whole code as follow

```
integer_number = 10
float_number = 3.5
string_text = "Hello"
boolean_value = True
addition_result = integer_number + float_number
subtraction_result = float_number - integer_number
multiplication_result = integer_number * 5
division_result = float_number / 2
exponential_result = integer_number ** 2
modulo_result = 17 % 3
expression_1 = integer_number + 20 * 5
expression_2 = (integer_number + float_number) * 2
print("Addition:", addition_result)
print("Subtraction:", subtraction_result)
print("Multiplication:", multiplication_result)
print("Division:", division_result)
print("Exponential:", exponential_result)
print("Modulo:", modulo_result)
print("Expression 1:", expression_1)
print("Expression 2:", expression_2)
```

Quality criteria: the output of the written code we be look like the following

```
Addition: 13.5
Subtraction: -6.5
Multiplication: 50
Division: 1.75
Exponential: 100
Modulo: 2
Expression 1: 110
Expression 2: 27.0
```

Operation sheet 1.8: Operators and If-Elif-Else Statement

Operation Title: Selection and operators

Purpose: To write a code that can calculate user inputs

Steps in doing the task

1. Write a code that can print a welcome message and prompt the user to enter two numbers and an operation

```
print("Welcome to the Python Calculator!")
print("Enter two numbers and the operation you want to perform (+, -, *, /)")
```

2. Get user input for the two numbers and the operation

```
# Getting user input for numbers and operation
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
operation = input("Enter the operation (+, -, *, /): ")
```

3. Perform the calculation based on the operation using conditional statements

```
# Perform the calculation based on the operation
if operation == '+':
    result = num1 + num2
    print(f"The sum of {num1} and {num2} is: {result}")
elif operation == '-':
    result = num1 - num2
    print(f"The difference of {num1} and {num2} is: {result}")
elif operation == '*':
    result = num1 * num2
    print(f"The product of {num1} and {num2} is: {result}")
elif operation == '/':
    if num2 == 0:
        print("Error: Division by zero is not allowed!")
    else:
        result = num1 / num2
        print(f"The division of {num1} by {num2} is: {result}")
else:
    print("Invalid operation entered. Please enter a valid operation.")
```

4. Write the whole code as follow

```
print("Welcome to the Python Calculator!")
print("Enter two numbers and the operation you want to perform (+, -, *, /)")
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
operation = input("Enter the operation (+, -, *, /): ")
if operation == '+':
    result = num1 + num2
    print(f"The sum of {num1} and {num2} is: {result}")
elif operation == '-':
    result = num1 - num2
    print(f"The difference of {num1} and {num2} is: {result}")
elif operation == '*':
    result = num1 * num2
    print(f"The product of {num1} and {num2} is: {result}")
elif operation == '/':
    if num2 == 0:
        print("Error: Division by zero is not allowed!")
    else:
        result = num1 / num2
        print(f"The division of {num1} by {num2} is: {result}")
else:
    print("Invalid operation entered. Please enter a valid operation.")
```

Quality criteria: The output should look like the following

```
Welcome to the Python Calculator!
Enter two numbers and the operation you want to perform (+, -, *, /)
Enter the first number: |
```

And will continue by asking the user second number and the operator.

Operation sheet 1.9: sequence, selection and iteration

Operation Title: coding of sequence, selection and iteration

Purpose: to demonstrate the basic programming concepts of sequence, selection, and iteration in Python

Steps in doing the task

1. Open your Code editor
2. Declare a list of numbers and named it **number** `numbers = [1, 2, 3, 4, 5]`
3. Enters a for loop, inside this loop, an if-else statement checks whether the current number is even or odd by using the modulo operator (%)

```
for num in numbers:
    if num % 2 == 0:
        print(f"{num} is even")
    else:
        print(f"{num} is odd")
```

4. Declares a variable count with a value of 5. It then enters a while loop that continues as long as count is greater than 0. Inside the loop, it prints the current value of count and then decreases count by 1. Once the while loop finishes (i.e., count is no longer greater than 0), it prints "Blast off!"

```
count = 5
while count > 0:
    print(f"Countdown: {count}")
    count -= 1
print("Blast off!")
```

5. Finally Run the whole code

```
# Sequence: Declaring a list of numbers
numbers = [1, 2, 3, 4, 5]
# Selection: Using a conditional statement (if-else) to check a
condition
for num in numbers:
    if num % 2 == 0:
        print(f"{num} is even")
    else:
        print(f"{num} is odd")
# Iteration: Using a loop (while loop) to perform an action repeatedly
count = 5
while count > 0:
    print(f"Countdown: {count}")
    count -= 1
print("Blast off!")
```


. **Quality Criteria:** The result must be

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Blast off!
```

Operation sheet 1.10: Arrays and Arays Object

Operation Title: coding of different array type

Purpose: To perform basic operations on arrays, including accepting user input, traversing an array, and inserting elements into an array.

Steps in doing the task

1. Create an empty list named user_array. It then asks the user to input the size and the elements of the array.

```
user_array = []
size = int(input("Enter the size of the array: "))
print("Enter the elements of the array:")
for i in range(size):
    element = int(input(f"Enter element {i + 1}: "))
    user_array.append(element)
```

2. Write a code that checks if the array is not empty. If it's not, it prints the first and last elements of the array.

```
print("\nIndexing Array:")
if len(user_array) > 0:
    print("First element:", user_array[0])
    print("Last element:", user_array[-1])
else:
    print("Array is empty.")
```

3. Write a code that uses a for loop to traverse through the array and print each element.

```
print("\nLooping Array:")
for num in user_array:
    print(num)
```

4. Write a code to ask the user to input an element. It then appends this element to user_array and prints the updated array.
5. Write a code that asks the user to input additional elements to extend the array. It then extends user_array with these elements and prints the updated array.

```
additional_numbers = input("Enter additional elements to extend the  
array (separated by spaces): ").split()
additional_numbers = [int(num) for num in additional_numbers]
user_array.extend(additional_numbers)
print("Array after extending:", user_array)
```

6. Finally write the whole code like the following:

```
user_array = []
size = int(input("Enter the size of the array: "))
print("Enter the elements of the array:")
- for i in range(size):
    element = int(input(f"Enter element {i + 1}: "))
    user_array.append(element)
print("\nIndexing Array:")
- if len(user_array) > 0:
    print("First element:", user_array[0])
    print("Last element:", user_array[-1])
- else:
    print("Array is empty.")
print("\nLooping Array:")
- for num in user_array:
    print(num)
append_element = int(input("\nEnter an element to append to the array:
    "))
user_array.append(append_element)
print("Array after appending:", user_array)
additional_numbers = input("Enter additional elements to extend the
    array (separated by spaces): ").split()
additional_numbers = [int(num) for num in additional_numbers]
user_array.extend(additional_numbers)
print("Array after extending:", user_array)
```

Quality Criteria: The output must ask for the array size and based on the must proceed asking the elements

Lap Tests

Instructions: Given necessary templates, tools and materials you are required to perform the following tasks accordingly.

Task 1: Install python on your windows computer

Task 2: Install VS code on your computer

Task 3: Set Up VS code for Python Development

Task 4: Write Python code that accomplishes the following output. Your code should prompt the user to input the size of the array, populate the array with elements, display the first and last elements, traverse the array, check and print whether each element is even or odd, and finally perform a countdown using a while loop until reaching zero, printing "Blast off!" at the end.

```
Enter the size of the array: 4
Enter the elements of the array:
Enter element 1: 7
Enter element 2: 14
Enter element 3: 5
Enter element 4: 9

Traversing and Indexing the Array:
First element: 7
Last element: 9
Array elements:
7
14
5
9

Conditional Statements based on Array Elements:
7 is odd
14 is even
5 is odd
9 is odd

Iteration using while loop:
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Blast off!
```

Unit Two: Basic OO principles

This unit is developed to provide you the necessary information regarding the following content coverage and topics:

- Primitive Member Variables in Class Implementation
- Flexible Object Construction
- User-Defined Aggregation in Class Design
- Navigating Hierarchical Inheritance
- Code Extension through Versatile Polymorphism

This unit will also assist you to attain the learning outcomes stated in the cover page.

Specifically, upon completion of this learning guide, you will be able to:

- Understand the concept of primitive member variables within class structures.
- Explore various methods for constructing objects in a flexible manner.
- Comprehend the concept of user-defined aggregation within class designs.
- Navigate and create class hierarchies through inheritance to establish relationships between classes.
- Grasp the fundamentals of polymorphism and its role in code extension.

2.1 Primitive Member Variables in Class Implementation

In Python, implementing classes involves the utilization of primitive member variables, which are fundamental data types directly assigned within a class. These variables are integral to defining the characteristics and states of objects created from the class blueprint. For instance, consider a simple 'Car' class:

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

Here, the **make**, **model**, **year**, and **mileage** are primitive member variables of the 'Car' class. They store basic data such as strings for make and model, integers for year, and mileage, forming the essential attributes of a car object. These variables encapsulate crucial information about each car instance created from this class.

Primitive member variables within a class facilitate encapsulation, enabling the class methods to access and manipulate these attributes, controlling their behaviour and providing a way to interact with them. For instance, one can create methods to update the mileage, retrieve specific information, or perform calculations based on these variables:

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

    def update_mileage(self, new_mileage):
        self.mileage = new_mileage

    def get_age(self, current_year):
        return current_year - self.year
```

In this updated 'Car' class, the **update_mileage** method allows modification of the **mileage** attribute, while **get_age** calculates the age of the car based on the current year. Overall, utilizing primitive member variables in Python classes serves as the foundation for building objects with distinct attributes and behaviors. These variables encapsulate the object's state, enabling the implementation of methods to manipulate and retrieve data, thus facilitating the organization and functionality of the class.

Generally, delve into the concept of classes in Python in greater detail, emphasizing the significance of primitive member variables and their broader implications within class-based programming. Classes in Python serve as blueprints for creating objects, defining their attributes (represented by member variables) and behaviors (implemented through methods). They encapsulate data and functionalities within a single unit, facilitating code organization, reuse, and maintainability.

Classes in Python promote the concept of encapsulation, where data (member variables) and methods (functions defined within the class) are bound together. This encapsulation ensures that the internal state of an object is protected, and access to its data is regulated through well-defined interfaces (methods). By utilizing primitive member variables and encapsulation, classes facilitate the creation of multiple instances (objects) that share the same structure but can hold different data.

Classes and their associated primitive member variables form the backbone of object-oriented programming in Python, enabling the creation of organized, reusable, and modular code that models real-world entities effectively.

A. Constructors

Constructors, represented by the `--init--` method in Python, initialize the object's state by assigning initial values to its member variables when an instance of the class is created. In the context of primitive member variables, the constructor is responsible for setting up these variables with the provided or default values.

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

Here, the `--init--` method sets up the primitive member variables (**make**, **model**, **year**, and **mileage**) for each **Car** object created. Parameters passed during object instantiation are used to initialize these variables, and **mileage** has a default value of 0 if not provided during object creation

B. Methods

Methods in a class are functions defined within the class structure that operate on the class's data (member variables). They encapsulate behaviours associated with the objects created from the class and enable interaction with the member variables

```
class Car:
    def __init__(self, make, model, year, mileage=0):
        self.make = make
        self.model = model
        self.year = year
        self.mileage = mileage
    def update_mileage(self, new_mileage):
        self.mileage = new_mileage
    def get_age(self, current_year):
        return current_year - self.year
```

In this example, **update_mileage** and **get_age** are methods defined within the **Car** class. The **update_mileage** method allows modification of the **mileage** attribute of a car object, while **get_age** computes the age of the car based on the **year** attribute and the current year passed as an argument.

- **Interaction between Constructors and Methods**

Constructors initialize the primitive member variables when an object is instantiated, providing the starting state for the object. Methods then operate on these member variables, allowing manipulation, retrieval, or modification of their values during the object's lifecycle

```
car1 = Car("Toyota", "Corolla", 2018, 0)
car1.update_mileage(50000)
current_year = 2023
age = car1.get_age(current_year)
print(f"The car's age is {age} years")
```

In this usage scenario, **car1** is an instance of the **Car** class. The constructor sets initial values for **make**, **model**, **year**, and **mileage**. The **update_mileage** method modifies the **mileage** attribute, and **get_age** calculates the car's age based on the current year.

Constructors initialize primitive member variables, providing the initial state of objects, while methods define the behaviours and operations that can be performed on these variables, facilitating interaction and manipulation throughout the object's lifespan.

This cohesive relationship between constructors and methods ensures effective management of primitive member variables within class implementation in Python.

In Python, the concepts of public, private, and protected member variables are associated with controlling access to class attributes (member variables) and methods. However, it's important to note that Python doesn't enforce access restrictions in the same way as some other programming languages (like Java or C++), but it provides conventions and some mechanisms to achieve similar functionality.

- **Public Member Variables:**

By default, all member variables in a Python class are considered public and can be accessed and modified from outside the class.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make      # Public member variable
        self.model = model    # Public member variable
        self.year = year      # Public member variable
car = Car("Toyota", "Corolla", 2020)
print(car.make) # Accessing a public member variable
car.model = "Camry" # Modifying a public member variable
```

In the above example, **make**, **model**, and **year** are public member variables of the **Car** class. They can be accessed and modified directly from outside the class.

- **Private Member Variables:**

In Python, there is a convention for indicating that a member variable should be treated as private by prefixing it with a double underscore (__). However, note that Python doesn't strictly enforce this as private, but it does implement name mangling to make it less accessible from outside the class.

```
class Car:
    def __init__(self, make, model, year):
        self.__make = make    # Private member variable
        self.__model = model  # Private member variable
        self.__year = year    # Private member variable
car = Car("Toyota", "Corolla", 2020)
print(car.__make) # Accessing a private member variable (throws an error)
```

In this example, attempting to access **__make** from outside the class directly will result in an error.

- **Protected Member Variables:**

In Python, there's a convention for indicating that a member variable should be treated as protected by prefixing it with a single underscore (_). This indicates that the variable should be treated as protected but it doesn't enforce it.

```
class Car:
    def __init__(self, make, model, year):
        self._make = make      # Protected member variable
        self._model = model    # Protected member variable
        self._year = year      # Protected member variable
car = Car("Toyota", "Corolla", 2020)
print(car._make) # Accessing a protected member variable (allowed but not
                 recommended)
```

Protected member variables are intended to indicate to other developers that these attributes should be considered non-public, and accessing or modifying them from outside the class should be done with caution. However, there's no strict enforcement in Python, and they can still be accessed directly.

In Python, these concepts are more about conventions and signaling intent rather than strict access control. Developers are expected to follow these conventions to respect the intended visibility of class attributes and methods.

2.2 Flexible Object Construction

In Python, classes are fundamental in defining the structure and behaviour of objects. One aspect of effective class design involves providing flexibility in creating objects by offering multiple options for object construction. This enables users to instantiate objects with various configurations based on their needs. Let's explore how to design such a class using different constructors.

- **Understanding Constructors in Python:**

In Python, the **--init--** method acts as a constructor and is called automatically when an object of a class is created. It initializes the object's attributes (member variables) with specific values provided during object instantiation.

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
    @classmethod
    def from_sku(cls, sku_code):
        # Perform logic to extract name and price based on SKU code
        name, price = get_info_from_sku(sku_code)
        return cls(name, price)
    @classmethod
    def from_file(cls, file_path):
        # Read data from a file to set name and price
        name, price = read_info_from_file(file_path)
        return cls(name, price)
```

Explanation:

- **Standard Constructor (--init--):** This constructor takes in **name** and **price** parameters to initialize the object directly with specified values.
- **Alternative Constructors (from_sku and from_file):** These are class methods (marked by **@classmethod**) that act as alternative constructors allowing object creation with different parameters.
- **from_sku:** Creates an instance by extracting information like **name** and **price** from a product's SKU code.

from_file: Reads information from a file (e.g., a CSV file) to set the **name** and **price**

```
# Creating objects using different constructors
product1 = Product("Laptop", 1200) # Using standard constructor
product2 = Product.from_sku("SKU123") # Using 'from_sku' constructor
product3 = Product.from_file("data.csv") # Using 'from_file' constructor
```

Designing a class with multiple constructors in Python provides flexibility in creating objects by offering various ways to initialize attributes. This approach allows users to create objects using different parameters or external data sources, enhancing the class's usability and versatility.

2.3 User-Defined Aggregation in Class Design

In Python, user-defined aggregation involves the creation of custom aggregation functions to process or combine data elements in a meaningful way. This concept is particularly useful in scenarios where built-in aggregation functions (such as `sum()`, `max()`, `min()`, etc.) might not suffice, or when there's a need to perform complex aggregations on custom data structures.

- **Understanding User-Defined Aggregation:**

Python provides powerful tools like ‘reduce()’ from the ‘functools’ module or list comprehensions for basic aggregation tasks. However, user-defined aggregation empowers developers to create custom aggregation logic tailored to specific requirements.

Example 1: Aggregating Custom Objects

Consider a scenario where you have a list of Product objects and want to find the average price of these products.

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
# List of Product objects
products = [
    Product("Laptop", 1200),
    Product("Phone", 800),
    Product("Headphones", 150) ]
# User-defined aggregation function to find average price
def calculate_average_price(product_list):
    total_price = sum(product.price for product in product_list)
    return total_price / len(product_list)
average_price = calculate_average_price(products)
print(f"The average price of products is: ${average_price}")
```

Here, ‘calculate_average_price()’ is a user-defined aggregation function that computes the average price of a list of **Product** objects by summing up their prices and dividing by the number of products.

Example 2: Aggregating Custom Data Structures

Suppose you have a dictionary containing sales data where each key represents a month, and the value is a list of sales amounts for that month.

```
sales_data = {
    'January': [1500, 2200, 1800],
    'February': [2000, 2500],
    'March': [1800, 1900, 2100, 2300] }
# User-defined aggregation function to find total sales for all months
def calculate_total_sales(data):
    total = sum(sum(month_sales) for month_sales in data.values())
    return total
total_sales = calculate_total_sales(sales_data)
print(f"The total sales across all months is: ${total_sales}")
```


In this example, `calculate_total_sales()` is a user-defined aggregation function that aggregates sales data across all months by summing up the sales amounts in the dictionary.

User-defined aggregation in Python offers the flexibility to create custom aggregation functions suited to specific data structures and aggregation requirements.

This approach enables developers to perform complex aggregations, process custom objects, or aggregate data in unique ways that might not be achievable using standard aggregation functions.

2.4 Navigating Hierarchical Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a class to inherit properties and behaviours from another class. Inheritance can be implemented to multiple levels of depth, creating a hierarchy of classes.

In Python, hierarchical inheritance refers to the arrangement of classes in a hierarchy where a child class inherits properties and behaviors from a single parent class, and multiple levels of inheritance exist. Navigating hierarchical inheritance involves understanding how attributes and methods are inherited across different levels of the inheritance tree, facilitating the utilization and extension of functionalities.

- **Understanding Hierarchical Inheritance:**

Hierarchical inheritance forms a tree-like structure, where each child class inherits from a common parent class. This allows the child classes to inherit attributes and methods from their immediate parent and, indirectly, from all the ancestors up the hierarchy.

Example 1: Basic Hierarchical Inheritance

```
class Animal:
    def __init__(self, species):
        self.species = species
    def make_sound(self):
        pass # Placeholder method

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"
```

Explanation:

- **Parent Class (Animal):** This class defines the common attributes and methods shared among its child classes. It has an `__init__` method initializing the **species** attribute and a placeholder **make_sound()** method.
- **Child Classes (Dog and Cat):** These classes inherit from the **Animal** class, inheriting the **species** attribute and the **make_sound()** method. However, they override the **make_sound()** method with their specific implementations

Example 2: Multiple Levels of Inheritance

Here, Mammal is a child class of Animal, and Dog and Cat inherit from Mammal. This demonstrates multiple levels of hierarchical inheritance, where Mammal inherits from Animal, and Dog and Cat inherit from Mammal

```
class Animal:
    def __init__(self, species):
        self.species = species
    def make_sound(self):
        pass # Placeholder method

class Mammal(Animal):
    def give_birth(self):
        return "Live birth"

class Dog(Mammal):
    def make_sound(self):
        return "Woof!"

class Cat(Mammal):
    def make_sound(self):
        return "Meow!"
```

- **Method Overriding In Inheritance**

Method overriding is a fundamental concept in object-oriented programming and inheritance, allowing a subclass to provide a specific implementation for a method that is already defined in its superclass. When a method in the subclass has the same name, same parameters, and same return type as a method in its superclass, the method in the subclass overrides the method in the superclass.

```
class Animal:
    def make_sound(self):
        return "Generic sound..."
```

Basic Understanding of Method Overriding:

Consider a scenario where a parent class (**Animal**) has a method `make_sound()`. Now, a subclass (**Dog**) inheriting from **Animal** can override the `make_sound()` method with its specific implementation

```
class Animal:
    def make_sound(self):
        return "Generic sound..."
```

Explanation:

- **Parent Class (Animal):** Defines a generic `make_sound()` method that provides a default behavior for all animals.
- **Child Class (Dog):** Overrides the `make_sound()` method with a specific implementation for a dog's sound

Example Demonstrating Method Overriding

```
class Animal:
    def make_sound(self):
        return "Generic sound..."
class Dog(Animal):
    def make_sound(self):
        return "Woof!"
class Cat(Animal):
    def make_sound(self):
        return "Meow!"
# Using overridden methods
dog = Dog()
print(dog.make_sound()) # Calls the overridden method in the Dog class
cat = Cat()
print(cat.make_sound()) # Calls the overridden method in the Cat class
```

In this example, both the **Dog** and **Cat** classes override the `make_sound()` method inherited from the **Animal** class with their specific sound implementations. When instances of **Dog** and **Cat** are created and the `make_sound()` method is called on them, the overridden methods in the respective child classes are executed.

- **Purpose and Benefits of Method Overriding:**

Customization: Allows subclasses to provide their specific implementation for methods inherited from the superclass.

Polymorphism: Facilitates polymorphic behavior, enabling different objects to be treated uniformly through a common interface

Flexibility: Provides flexibility in modifying the behavior of inherited methods without altering the superclass's implementation.

2.5 Code Extension through Versatile Polymorphism

Code Extension through Versatile Polymorphism refers to the ability to extend and enhance the functionality of code by leveraging polymorphism in various ways, allowing for adaptable and flexible behaviour within an object-oriented programming paradigm.

A. Understanding Polymorphism:

Polymorphism, a key principle in object-oriented programming, allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used for entities of different types, promoting flexibility and code reusability.

B. Versatile Nature of Polymorphism:

Polymorphism manifests in various forms:

1. **Method Overriding:** As discussed earlier, where a subclass can provide its implementation of a method inherited from its superclass, adapting its behavior as needed.
2. **Parameterized Polymorphism:** Achieved through methods or functions that can accept parameters of different types, enabling them to behave differently based on the type of object passed.
3. **Operator Overloading:** Allowing operators like +, -, *, etc., to behave differently based on the operands' types.

Self-check: 3

I. Say TRUE or FALSE

1. Polymorphism in Python allows objects of different classes to be treated uniformly through a common interface.
2. Method overriding in inheritance allows a subclass to provide a specific implementation for a method that is already defined in its superclass.
3. Hierarchical inheritance in Python creates a tree-like structure where child classes inherit properties only from their immediate parent class.
4. Python provides strict access control through keywords like 'public', 'private', and 'protected' to manage member variables in classes.
5. User-defined aggregation in Python refers to the creation of custom aggregation functions to process or combine data elements, catering to specific requirements.

II. Choose the best answer from the choice provided

1. **Which of the following describes method overriding in Python?**
 - A) Creating methods with different names in parent and child classes.
 - B) Providing a specific implementation for a method in the child class that is already defined in its superclass.
 - C) Allowing methods to accept different parameters based on the object type.
 - D) Using built-in aggregation functions to process data elements.
2. **What does polymorphism enable in object-oriented programming?**
 - A) Treating objects of different classes as identical.
 - B) Restricting the use of methods from the superclass in child classes.
 - C) Enforcing strict access control for member variables.
 - D) Limiting the use of inheritance among classes.
3. **Which statement best defines hierarchical inheritance in Python?**
 - A) A child class inherits from multiple parent classes.
 - B) All classes in the hierarchy inherit directly from a single superclass.
 - C) A class inherits properties and behaviors from multiple ancestor classes.
 - D) Child classes inherit from a common parent class and form a tree-like structure.

4. **What is the purpose of user-defined aggregation in Python?**

- A) Restricting access to methods and attributes in classes.
- B) Providing a common interface for objects of different types.
- C) Creating custom aggregation functions to process or combine data elements.
- D) Defining multiple constructors for initializing objects.

III. Elaborate the following questions

1. Explain the concept of method overriding in Python inheritance with an example.
2. Discuss the versatility of polymorphism in Python.
3. Illustrate the hierarchical inheritance structure in Python with a comprehensive explanation.
4. Explain the significance of user-defined aggregation in Python.
5. Discuss the various types of access control for member variables in Python classes

Operation sheet 2.1 : Object Construction

Operation Title: Extracting Text File Names

Purpose: To extract text file names from a list of file names using list comprehension

Steps in doing the task

1. Write a code that can define a list of file names

```
# Sample list of file names
file_names = [
    "document1.txt",
    "image.jpg",
    "script.py",
    "notes.txt",
    "data.csv",
    "report.docx"
]
```

2. Extract only the names of text files using list comprehension

```
# Extracting only the names of text files using list comprehension
text_files = [file for file in file_names if file.endswith(".txt")]
```

3. Display the extracted text file names

```
# Displaying the extracted text file names
print("Text Files:")
print(text_files)
```

4. Write the whole code as follow

```
file_names = [
    "document1.txt",
    "image.jpg",
    "script.py",
    "notes.txt",
    "data.csv",
    "report.docx"
]
text_files = [file for file in file_names if file.endswith(".txt")]
print("Text Files:")
print(text_files)
```

Quality criteria: the output should look like the following

```
Text Files:
['document1.txt', 'notes.txt']
```


Operation sheet 2.2: Primitive member of variables

Operation Title: Primitive member variables within a class

Purpose: To demonstrate the use of primitive member variables within a class.

Steps in doing the task:

1. Define student class. This is a blueprint for creating Student objects.
2. Define `__init__` method. This is a special method that gets called when a new Student object is created. It takes `self`, `name`, `age`, and `grade` as parameters.

```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade
```

3. The `display_info` method is defined. This method prints the name, age, and grade of a Student object when called.

```
def display_info(self):
    print(f"Student Name: {self.name}")
    print(f"Age: {self.age}")
    print(f"Grade: {self.grade}")
```

4. Creating a `student1` Instance created by calling the Student class with the name "Alice", age 15, and grade "9th".

```
# Creating an instance of the Student class
student1 = Student("Alice", 15, "9th")
# Accessing and displaying student information using the method
student1.display_info()
```

5. Finally type the whole code as follow

```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display_info(self):
        print(f"Student Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Grade: {self.grade}")

# Creating an instance of the Student class
student1 = Student("Alice", 15, "9th")

# Accessing and displaying student information using the method
student1.display_info()
```

Quality criteria: the output must look like these.

```
Student Name: Alice
Age: 15
Grade: 9th
```

Operation sheet 2.3: Object Construction and User-Defined Aggregation

Operation Title: Object and Aggregation in Class

Purpose: Demonstrating flexible object construction and user-defined aggregation within a class.

Steps in doing the task:

1. Define two classes, **Student** and **Course**. Student represents an individual student, while Course represents a course that can have multiple students.
2. Define the `__init__` method in both classes. This method gets called when a new object is created. It takes `self` and other parameters to initialize the object's attributes.
3. Create `self.name`, `self.student_id` in Student and `self.course_name`, `self.course_code`, `self.students` in Course.
4. Create **display_info** method in both classes' prints the object's information. The **add_student** method in Course adds a Student object to the course.
5. Create Three Student objects and one Course object.
6. Create `display_course_info` method to call the Course object to display the course information and the details of the enrolled students.
7. Create the whole code like the following:

```
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

    def display_info(self):
        print(f"Student Name: {self.name}")
        print(f"Student ID: {self.student_id}")

class Course:
    def __init__(self, course_name, course_code):
        self.course_name = course_name
        self.course_code = course_code
        self.students = [] # List to store Student obj

    def add_student(self, student):
        self.students.append(student)

    def display_course_info(self):
        print(f"Course Name: {self.course_name}")
        print(f"Course Code: {self.course_code}")
        print("Students enrolled:")
        for student in self.students:
            student.display_info()
        print("-----")

# Creating Student objects
student1 = Student("Alice", 101)
student2 = Student("Bob", 102)
student3 = Student("Charlie", 103)
# Creating a Course object
course_python = Course("Python Programming", "PY101")
# Adding students to the course
course_python.add_student(student1)
course_python.add_student(student2)
course_python.add_student(student3)
# Displaying course information along with enrolled stu
course_python.display_course_info()
```

```
Course Name: Python Programming
Course Code: PY101
Students enrolled:
Student Name: Alice
Student ID: 101
-----
Student Name: Bob
Student ID: 102
-----
Student Name: Charlie
Student ID: 103
-----
```

Quality criteria: The output should be look like the following

Page 74 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

Operation sheet 2.4: Inheritance

Operation Title: writing inheritance code

Purpose: To demonstrate the concept of inheritance in object-oriented programming.

Steps in doing the task:

1. Create the Employee class. It should have the following attributes: name and emp_id. These attributes should be initialized in the `__init__` method. Also, create a method `display_info` that prints the employee's name and ID

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

    def display_info(self):
        print(f"Employee Name: {self.name}")
        print(f"Employee ID: {self.emp_id}")
```

2. Next, create the FullTimeEmployee and PartTimeEmployee classes. These are subclasses that inherit from the Employee base class.
3. The FullTimeEmployee class should have an additional attribute salary, and the PartTimeEmployee class should have additional attributes hourly_rate and hours_worked.
4. The attributes should be initialized in their respective `__init__` methods, along with the base class attributes. Also, override the `display_info` method in each subclass to print the additional attributes along with the base attributes.

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

    def display_info(self):
        super().display_info()
        print(f"Employee Type: Full-Time")
        print(f"Salary: ${self.salary}")

class PartTimeEmployee(Employee):
    def __init__(self, name, emp_id, hourly_rate, hours_worked):
        super().__init__(name, emp_id)
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def display_info(self):
        super().display_info()
        print(f"Employee Type: Part-Time")
        print(f"Hourly Rate: ${self.hourly_rate}")
        print(f"Hours Worked: {self.hours_worked}")
```

5. Finally, create instances of FullTimeEmployee and PartTimeEmployee and call their display_info methods to test the program.

```
full_time_employee = FullTimeEmployee("Alice", 101, 60000)
part_time_employee = PartTimeEmployee("Bob", 102, 20, 25)

print("Full-Time Employee Details:")
full_time_employee.display_info()

print("\nPart-Time Employee Details:")
part_time_employee.display_info()
```

6. Run the program and verify that it correctly prints the details of the Full-Time and Part-Time employees.

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

    def display_info(self):
        print(f"Employee Name: {self.name}")
        print(f"Employee ID: {self.emp_id}")

class FullTimeEmployee(Employee):
    def __init__(self, name, emp_id, salary):
        super().__init__(name, emp_id)
        self.salary = salary

    def display_info(self):
        super().display_info()
        print(f"Employee Type: Full-Time")
        print(f"Salary: ETB{self.salary}")

class PartTimeEmployee(Employee):
    def __init__(self, name, emp_id, hourly_rate, hours_worked):
        super().__init__(name, emp_id)
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def display_info(self):
        super().display_info()
        print(f"Employee Type: Part-Time")
        print(f"Hourly Rate: ETB{self.hourly_rate}")
        print(f"Hours Worked: {self.hours_worked}")

full_time_employee = FullTimeEmployee("Alice", 101, 60000)
part_time_employee = PartTimeEmployee("Bob", 102, 20, 25)
print("Full-Time Employee Details:")
full_time_employee.display_info()
print("\nPart-Time Employee Details:")
part_time_employee.display_info()
```

Quality criteria: The output should look like

```
Full-Time Employee Details:
Employee Name: Alice
Employee ID: 101
Employee Type: Full-Time
Salary: ETB60000

Part-Time Employee Details:
Employee Name: Bob
Employee ID: 102
Employee Type: Part-Time
Hourly Rate: ETB20
Hours Worked: 25
```

Operation sheet 2.5: Polymorphism

Operation Title: Versatile Polymorphism

Purpose: To demonstrate the concept of polymorphism in object-oriented programming.

Steps in doing the task:

1. Create Shape class, It should have a placeholder method area() that will be overridden in each subclass.

```
class Shape:
    def area(self):
        pass # Placeholder method
```

2. Create the Circle and Rectangle classes. And it should have an attribute radius, and the Rectangle class should have attributes length and width. These attributes should be initialized in their respective __init__ methods.

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

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

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

3. Create a function calculate_area(shape). It takes an object of the Shape class (or any subclass) and calls its area() method.

```
def calculate_area(shape):
    return shape.area()
```

4. Create instances of Circle and Rectangle and pass them to the calculate_area() function.

```
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(calculate_area(circle)) # Calculates area of the circle
print(calculate_area(rectangle)) # Calculates area of the rectangle
```

5. Run the program as like as follow :

```
class Shape:
    def area(self):
        pass # Placeholder method

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self): # Correct indentation for the area method
        return 3.14 * self.radius ** 2

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self): # Correct indentation for the area method
        return self.length * self.width

# Function demonstrating polymorphic behavior
def calculate_area(shape):
    return shape.area()

# Using versatile polymorphism
circle = Circle(5)
rectangle = Rectangle(4, 6)
print(calculate_area(circle)) # Calculates area of the circle
print(calculate_area(rectangle)) # Calculates area of the rectangle
```

Quality criteria: The output of the above program must be like the following:

```
78.5
24
```


Lap Tests

Instructions: Given necessary templates, tools and materials you are required to perform the following tasks accordingly.

Task 1: write a python code that can display the following

```
Enter the number of dogs to add to the zoo: 2
Enter dog's name: Bella
Enter dog's age: 3
Enter dog's breed: Labrador
Enter dog's name: Max
Enter dog's age: 2
Enter dog's breed: Bulldog
Enter the number of cats to add to the zoo: 1
Enter cat's name: Luna
Enter cat's age: 4
Enter cat's color: Black

- Zoo Animals:
Dog Name: Bella
Age: 3
Breed: Labrador
Sound: Woof!
-----
Dog Name: Max
Age: 2
Breed: Bulldog
Sound: Woof!
-----
Cat Name: Luna
Age: 4
Color: Black
Sound: Meow!
```

Based on the following instruction

1. Prompt the user to input the number of dogs they want to add to the zoo.
2. For each dog:
 - Ask for the dog's name, age, and breed.
 - Create a **Dog** object using the provided information and add it to the zoo.
3. Prompt the user to input the number of cats they want to add to the zoo.
4. For each cat:
 - Ask for the cat's name, age, and color.
 - Create a **Cat** object using the provided information and add it to the zoo.
5. Finally, display information about all animals in the zoo, including their names, ages, breeds/colors, and the sounds they make.

Unit Three: Debug code

This unit is developed to provide you the necessary information regarding the following content coverage and topics:

- Integrated Development Environment (IDEs)
- Program Debugging Techniques

This unit will also assist you to attain the learning outcomes stated in the cover page.

Specifically, upon completion of this learning guide, you will be able to:

- Employ time-saving techniques and shortcuts within the IDE to enhance coding efficiency and productivity.
- Master various debugging techniques.
- Develop the skill to analyze runtime errors, logical issues, and exceptions within code, and apply appropriate debugging methods to rectify them.

3.1 Integrated Development Environment (IDEs)

Integrated Development Environments (IDEs) serve as comprehensive software applications designed to streamline the process of software development. These environments provide programmers and developers with a unified platform encompassing various tools and functionalities essential for creating, testing, and deploying software. An IDE typically integrates a code editor, debugger, compiler, and often version control systems, offering a cohesive workspace to write, edit, debug, and manage code efficiently. The primary purpose of an IDE is to enhance developers' productivity by providing features such as syntax highlighting, auto-completion, debugging tools, and project management capabilities within a single user interface. By consolidating these tools into one environment, IDEs aim to simplify and optimize the software development process, allowing developers to focus on coding and problem-solving rather than managing disparate tools separately. Popular examples of IDEs include Visual Studio, IntelliJ IDEA, Eclipse, and PyCharm, each tailored to different programming languages and providing a range of features catering to developers' needs.

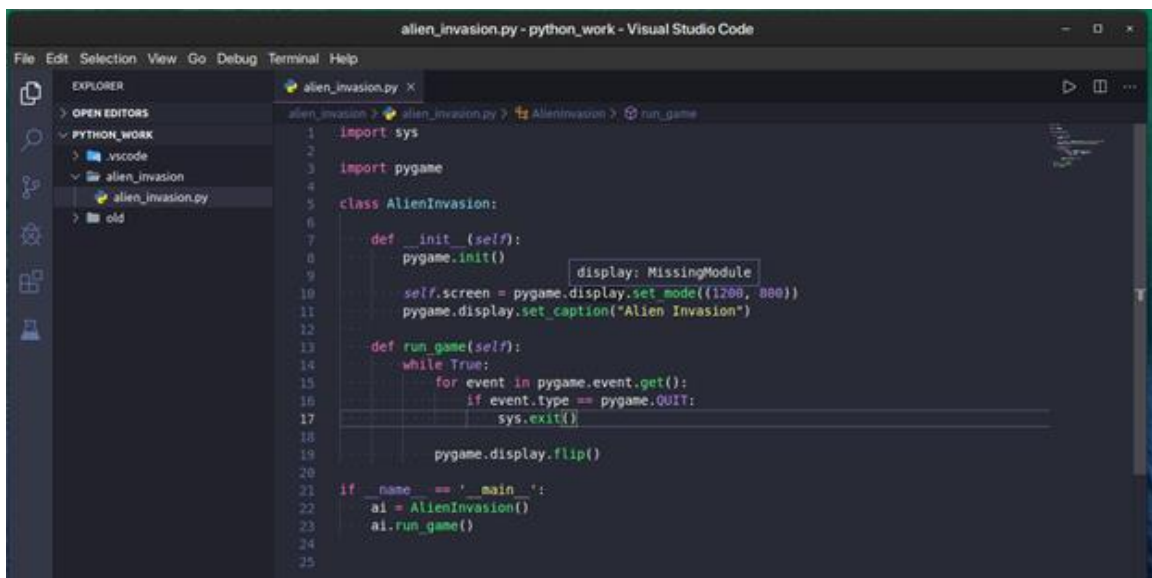


Fig: 3.1: debugging Using VS code

There are several Integrated Development Environments (IDEs) that support debugging Python code effectively. Here are a few notable ones:

1. **PyCharm by JetBrains:** PyCharm is a widely used IDE that offers robust debugging capabilities for Python. It provides an interactive debugger with features like breakpoints, variable inspection, stepping through code, watch expressions, and stack trace analysis. PyCharm's debugging tools allow developers to efficiently identify and resolve issues in their Python code.
2. **Visual Studio Code (VS Code) with Python Extension:** VS Code, a lightweight and versatile code editor, has a Python extension that provides excellent support for debugging Python applications. It offers features like breakpoints, variable tracking, stepping through code, and integrated terminal access, enabling developers to debug Python code seamlessly.
3. **Spyder:** Spyder is an open-source IDE designed specifically for scientific computing and data analysis in Python. It includes an interactive debugger that allows users to set breakpoints, inspect variables, and step through code, making it suitable for debugging Python code involved in data analysis and scientific computations.
4. **PyDev for Eclipse:** PyDev is a Python IDE plugin for the Eclipse platform. It provides a feature-rich environment for Python development, including debugging capabilities like breakpoints, variable inspection, and interactive debugging, leveraging Eclipse's extensibility.

These IDEs offer comprehensive debugging tools and features that aid developers in efficiently identifying and resolving issues within their Python codebases, catering to different preferences and project requirements. Developers can choose an IDE based on their familiarity, workflow preferences, and specific needs for Python debugging.

3.2 Program Debugging Techniques

Program debugging techniques encompass a range of systematic approaches and methodologies aimed at identifying and resolving errors or bugs within software code. These techniques are fundamental in ensuring code functionality, reliability, and overall software quality.

- **Logging and Output Analysis:** Incorporating logging statements strategically within the code allows developers to output specific messages, variable values, or program flow details. Analyzing log outputs aids in understanding the code's behavior during runtime without halting the program's execution, offering valuable insights into potential issues.
- **Print Statement Debugging:** Placing print statements at crucial points in the code to display variable values, intermediate outputs, or the program's execution flow. This technique, while simplistic, is effective for quick debugging, enabling developers to track the program's execution and identify potential issues or unexpected behavior.
- **Utilizing Debugger Tools:** Leveraging specialized debugging tools provided by Integrated Development Environments (IDEs) or standalone debugging software. These tools, such as breakpoints, watchpoints, step-by-step execution, and variable inspection, allow developers to closely analyze code behavior during runtime, facilitating precise identification and resolution of errors.
- **Exception Handling:** Implementing try-except blocks to catch and handle exceptions or errors gracefully. By incorporating exception handling, developers prevent program termination due to errors, gaining insights into the nature and location of potential issues.
- **Code Review and Pair Programming:** Collaborating with peers for code review or engaging in pair programming sessions to identify potential issues. These practices offer fresh perspectives, aiding in the identification of logical errors, inefficiencies, or bugs that might have been overlooked by the original developer.
- **Using Assertions:** Inserting assertions to validate assumptions about code behavior. Assertions act as checks during code execution to ensure certain conditions hold true, indicating deviations from expected behavior and potential issues.

- **Automated Testing:** Developing comprehensive test suites and performing automated tests to detect bugs systematically. Automated testing verifies code functionality, ensuring that changes to the codebase do not introduce new bugs, regressions, or unexpected behavior.
- **Profiling and Performance Analysis:** Utilizing profiling tools to analyze code performance and identify bottlenecks. Profiling aids in optimizing code by revealing areas consuming excessive resources or causing slowdowns, indirectly aiding in debugging by highlighting potential sources of errors or inefficiencies.

Employing multiple debugging techniques tailored to the specific context of the codebase and encountered errors enables developers to systematically identify, isolate, and resolve issues. These methods collectively contribute to the creation of robust, error-free software by ensuring comprehensive error detection and resolution throughout the development process.

Self-Check : 3

I. Say TRUE or FALSE

1. Logging and Output Analysis provide insights into the code's behaviour during runtime by halting the program's execution.
2. Assertion statements in debugging are used to validate assumptions about code behaviour during runtime.
3. Spyder is an IDE specifically designed for general-purpose Python development, not focused on scientific computing or data analysis.
4. Integrated Development Environments (IDEs) aim to complicate and diversify the software development process by managing disparate tools separately.

II. Choose the best answer from the provided choices

1. Which of the following debugging techniques involves strategically placing statements to display variable values or execution flow?
 - A) Exception Handling
 - B) Assertions
 - C) Print Statement Debugging
 - D) Automated Testing
2. What is the primary purpose of an Integrated Development Environment (IDE)?
 - A) To confuse developers with complex interfaces
 - B) To consolidate various tools for streamlined software development
 - C) To eliminate the need for version control systems
 - D) To limit developers' productivity by providing limited features
3. Which of the following is NOT a feature typically integrated into an IDE?
 - A) Code editor
 - B) Debugger
 - C) Version Control Systems
 - D) Manual code compilation
4. Which Python IDE offers robust debugging capabilities such as breakpoints, variable inspection, and stack trace analysis?
 - A) PyCharm by JetBrains
 - B) Spyder
 - C) Visual Studio Code with Python Extension
 - D) PyDev for Eclipse

Unit Four: Document activities

This unit is developed to provide you the necessary information regarding the following content coverage and topics:

- Crafting Maintainable Object-Oriented Code
- Documentation of Object-Oriented Code

This unit will also assist you to attain the learning outcomes stated in the cover page.

Specifically, upon completion of this learning guide, you will be able to:

- Maintain well-crafted Object- Oriented code
- Document Object- Oriented code

4.1 Crafting Maintainable Object-Oriented Code

Crafting maintainable object-oriented code entails designing and implementing code that prioritizes ease of maintenance, scalability, and adaptability. This process revolves around employing SOLID principles—Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion—to guide the design of object-oriented systems. It involves breaking down the codebase into cohesive components with distinct responsibilities, reducing coupling, and enabling extension without altering existing code. Leveraging encapsulation, inheritance, polymorphism, and abstraction, developers create modular and reusable structures, hiding complexities, promoting code reuse, and facilitating flexibility. Writing clean, self-documenting code with clear naming conventions, logical structure, and consistent styles is pivotal. Coupled with comprehensive unit testing and refactoring, this approach ensures code reliability, catches errors early, and continuously improves structure without altering external behavior. It necessitates architectural foresight, adherence to OOP principles, continuous code reviews, and commitment to producing understandable, adaptable, and extensible code throughout the software's lifecycle. This approach minimizes technical debt, fosters collaboration, and enhances the software's long-term viability and agility.

- **Guidelines to create maintainable code while following a set coding standard:**

A. Adhering to PEP 8

PEP 8 stands for "Python Enhancement Proposal 8," and it's the official style guide for Python code. It outlines guidelines and recommendations for writing Python code in a consistent and readable manner. Created by Guido van Rossum, Barry Warsaw, and Nick Coghlan in 2001, PEP 8 aims to enhance the readability of Python code and promote a common coding style across the Python community.

Key aspects covered in PEP 8 include:

1. **Code Layout:** Recommendations for indentation, line length, and the use of whitespace to make code more readable.
2. **Naming Conventions:** Guidelines for naming variables, functions, classes, and modules, ensuring consistency and readability.

Page 87 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

3. Comments and Documentation: Suggestions for writing comments, docstrings, and documentation to improve code understanding.
4. Programming Recommendations: Best practices and recommendations for writing Python code that is clear, concise, and easy to understand.
5. Imports: Guidelines for organizing imports and managing module dependencies in Python scripts.

Adhering to PEP 8 helps in creating code that is not only consistent but also more readable and maintainable. However, it's important to note that PEP 8 is a set of guidelines and not strict rules. In some cases, deviation might be acceptable, especially when adhering strictly to PEP 8 reduces code clarity or readability. Nonetheless, following PEP 8's recommendations generally helps improve code quality and promotes collaboration among developers within the Python community.

B. Writing Readable and Self-Explanatory Code

It refers to the practice of creating code that is easily understandable, clear, and coherent without needing extensive comments or explanations. The primary goal is to ensure that the code can be read and comprehended effortlessly by other developers, including your future self.

Key principles for writing readable and self-explanatory code include:

Meaningful Naming: Use descriptive and intuitive names for variables, functions, classes, and methods. Names should reflect the purpose and functionality of the entity they represent, making the code more understandable at first glance.

Clear and Concise Code: Break down complex logic into smaller, well-structured functions or methods. Each function or method should ideally have a single responsibility, performing a specific task without unnecessary complexity. This makes the code easier to read, understand, and maintain.

- **Avoiding Ambiguity:** Write code that is explicit and leaves no room for ambiguity. Avoid cryptic or overly complex expressions that might confuse other developers. Instead, favor simplicity and clarity in the code structure and logic.
- **Documentation and Comments:** While the goal is to write code that is self-explanatory, sometimes it's necessary to include comments or documentation to clarify intricate or

non-obvious sections. However, the emphasis should be on writing code that doesn't rely heavily on comments to be understood.

- **Avoiding Magic Numbers and Strings:** Replace hard-coded values (referred to as "magic numbers" or "magic strings") with named constants or variables. This practice enhances code readability and makes it easier to maintain and modify in the future.
- **Consistency and Formatting:** Follow a consistent coding style throughout the codebase. Adhering to a standard style guide, such as PEP 8 in the case of Python, helps maintain uniformity and readability across the project.
- **Readable Control Structures:** Ensure that control structures like loops and conditionals are well-structured and easy to follow. Use meaningful variable names in loops and avoid deeply nested or convoluted logic.

Writing readable and self-explanatory code is crucial for collaboration among developers, especially in large projects. Code that is clear and understandable not only helps in reducing bugs and errors but also improves maintainability and facilitates future enhancements or modifications to the codebase.

- **Modularization and Encapsulation:**

Fundamental principles in software design, particularly within the context of object-oriented programming (OOP), aim to enhance code organization, maintainability, and reusability.

C. Modularization:

Modularization involves breaking down a software system into smaller, independent modules or components, each responsible for a specific functionality or feature. These modules act as building blocks that can be developed, tested, and maintained separately, contributing to easier code management and reducing complexity.

Benefits of Modularization:

- **Code Reusability:** Modules can be reused across different parts of an application or in other projects, promoting efficiency and reducing redundant code.
- **Simplified Maintenance:** Smaller, focused modules are easier to understand, update, and maintain, leading to quicker bug fixes and enhancements.

Page 89 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

- **Scalability:** Modular design allows for easier scalability as new functionalities or features can be added by creating or extending modules without affecting the entire system.

D. Encapsulation:

Encapsulation is a key concept in OOP that involves bundling the data (attributes or properties) and the methods (functions or behavior) that operate on that data within a single unit, typically a class. This bundling is done in such a way that the internal workings and data of an object are hidden and can only be accessed or modified through well-defined interfaces (methods).

Benefits of Encapsulation:

- **Data Protection:** Encapsulation allows for the protection of data from unauthorized access or modifications by providing controlled access through methods (getters and setters).
- **Improved Maintainability:** By hiding the implementation details, changes within a class don't affect the external code using that class, promoting easier maintenance and reducing dependencies.
- **Code Flexibility:** It enables better control over how data is accessed or manipulated, allowing modifications to the internal implementation without affecting other parts of the program.

Together, modularization and encapsulation play crucial roles in creating robust, maintainable, and scalable software systems. Modularization promotes a clear and organized structure by breaking down complex systems into manageable parts, while encapsulation ensures that each part of the system is self-contained, reducing dependencies and facilitating easier maintenance and evolution of the codebase.

- **Documentation:**

Documentation refers to the process of creating and maintaining records, descriptions, instructions, or explanations that accompany software code, applications, systems, or processes. It serves as a detailed reference or guide for developers, users, or other stakeholders involved in working with or understanding a particular piece of software or system.

In software development, documentation typically includes:

1. **Code Comments:** Inline comments within the code that explains specific sections or complex logic. These comments are for developers' understanding and help clarify the code's functionality.
2. **Docstrings:** Descriptive strings of text included in code to document functions, classes, methods, and modules. Docstrings are usually accessible through programming interfaces and are meant to provide comprehensive information about the purpose, inputs, outputs, and usage of code elements.
3. **Guides and Manuals:** External documents or resources providing instructions, guidelines, or explanations for using and understanding the software. This might include user manuals, installation guides, troubleshooting guides, and developer documentation.
4. **API Documentation:** For software libraries or frameworks, API documentation outlines how to use the application programming interfaces (APIs) provided by the software, including details about available functions, classes, parameters, and return values.
5. **Design and Architecture Documents:** High-level descriptions and diagrams illustrating the system's design, architecture, components, relationships, and interactions. These documents often serve as a blueprint for developers and stakeholders to understand the system's structure.

Effective documentation is essential for various reasons:

- **Understanding:** It helps developers and users comprehend the code, system, or application, especially when working on or troubleshooting it.
- **Maintenance:** Good documentation facilitates easier maintenance and updates by providing insights into the code's structure, purpose, and dependencies.

- **Onboarding and Training:** Documentation serves as a valuable resource for new team members to understand existing systems and for users to learn how to use software effectively.
- **Collaboration:** It fosters collaboration among team members by providing a common reference point and ensuring everyone understands the software's functionalities and usage.

Maintaining up-to-date and accurate documentation is crucial for the long-term success, usability, and maintainability of software project

➤ **Testing and Quality Assurance:**

Refer to the processes and practices within software development aimed at ensuring that software products meet predefined quality standards, are free from defects, and function as intended.

1. Testing:

Testing involves the systematic evaluation of software to identify and fix errors, bugs, or unexpected behavior. It's performed across different stages of the software development life cycle (SDLC) and includes various types of tests:

- **Unit Testing:** Tests individual components or units of code to verify their functionality in isolation from the rest of the software.
- **Integration Testing:** Verifies the interaction between different units or modules to ensure they work together as expected.
- **System Testing:** Tests the entire system to ensure it meets specified requirements and functions correctly as a whole.
- **Regression Testing:** Repeatedly tests the software after modifications to ensure that existing functionalities aren't affected by the changes.
- **Performance Testing:** Assesses the system's performance under various conditions, such as load, stress, or scalability testing, to ensure it meets performance requirements.

2. Quality Assurance:

Quality Assurance involves activities that focus on establishing and maintaining processes to ensure that the development and delivery of software adhere to predefined quality standards. It includes:

- **Quality Planning:** Defining quality standards, processes, and procedures that need to be followed during the software development life cycle.
- **Process Audits:** Reviewing and assessing adherence to established processes and standards to ensure consistency and compliance.
- **Defect Prevention:** Identifying potential issues early in the development process and implementing measures to prevent defects from occurring.
- **Continuous Improvement:** Striving for continuous enhancement of processes and methodologies based on feedback, best practices, and lessons learned from previous projects.

Testing and Quality Assurance work hand-in-hand to ensure that software is reliable, functional, secure, and meets user expectations. They contribute significantly to delivering high-quality software products by identifying and addressing issues early in the development cycle, reducing risks, and enhancing customer satisfaction. These practices are essential for building robust and maintainable software systems.

➤ Version Control and Collaboration:

- **Version Control Best Practices:** Utilize Git for version control, following branching strategies like GitFlow, and commit frequently with clear, descriptive commit messages.
- **Collaborative Development:** Foster collaboration through code reviews, ensuring that changes align with coding standards and maintainability guidelines.

➤ Refactoring and Continuous Improvement:

Regular Refactoring: Refactor code continuously to enhance readability, eliminate duplication, and improve maintainability without altering external behavior.

Learning and Adapting: Stay updated with Python best practices, new features, and community standards to evolve the codebase accordingly.

By incorporating these guidelines into your Python development process, you can ensure the creation of maintainable code that aligns with established coding standards, facilitating collaboration, readability, and long-term maintenance.

4.2 Documentation of Object-oriented Programming

Documenting object-oriented code in Python involves creating comprehensive, understandable documentation that clarifies classes, methods, interfaces, and modules within the codebase. Leveraging docstrings, which follow specific formats like Google Python Style Guide or numpydoc, serves as self-contained documentation, aiding understanding via the `help()` function or tools like Sphinx. It's crucial to document classes and methods, detailing their purpose, attributes, method signatures, parameters, return values, and exceptions raised. UML diagrams, like class or sequence diagrams, offer visual representations of class relationships and structures, complementing textual documentation and enhancing understanding.

In addition to docstrings and UML diagrams, inline comments are employed to explain complex algorithms or less intuitive code sections, offering detailed context beyond high-level descriptions. Maintaining versioning and change logs alongside documentation aids in tracking code evolution and understanding modifications made in different iterations. This combination of well-crafted docstrings, UML diagrams, and concise comments serves as a vital resource for developers, facilitating comprehension, consistent usage, maintenance, and collaborative development processes.

Ultimately, by ensuring clear, up-to-date documentation, Python developers promote the longevity, reliability, and usability of their object-oriented codebases. This documentation acts as a foundational asset, fostering comprehension, consistency, and support throughout the software development lifecycle.

Self-check :4

I. Say TRUE or FALSE

1. Docstrings in Python are single-line comments used to clarify specific sections of code.
2. UML diagrams, such as class diagrams and sequence diagrams, are used in Python documentation to visually represent the execution flow of a program.
3. Inline comments in Python code are intended to provide high-level descriptions for classes and their methods.
4. Versioning and change logs in Python documentation help track the evolution of the codebase and record modifications made in different versions.

II. Choose the best answer from the provided choices

5. **Which Python tool generates documentation from docstrings?**
 - a) Sphinx
 - b) Black
 - c) Flake8
 - d) Pylint
6. **What purpose do UML diagrams serve in Python documentation?**
 - a) To define syntax rules for Python classes
 - b) To represent user interface designs
 - c) To visually illustrate relationships between classes and their interactions
 - d) To showcase Python's debugging capabilities
7. **What information is typically included in class docstrings in Python?**
 - a) Description of specific code lines
 - b) Expected parameters for methods
 - c) List of global variables
 - d) Usage examples for the class attribute
8. **What role do change logs play in Python documentation?**
 - a) They assist in tracking changes made in different software versions.
 - b) They provide syntax rules for Python classes.
 - c) They generate automated testing reports.
 - d) They outline the execution flow of Python programs.

Unit Five: Test code

This unit is developed to provide you the necessary information regarding the following content coverage and topics:

- Simple Tests in Object-Oriented Programming
- Documenting performed test
- Embracing Corrections in Code and Documentation

This unit will also assist you to attain the learning outcomes stated in the cover page.

Specifically, upon completion of this learning guide, you will be able to:

- Perform Simple Tests in Object-Oriented Programming
- Document performed test
- Embrace Corrections in Code and Documentation

5.1. Simple Tests in Object-Oriented Programming

In Python's object-oriented programming (OOP) paradigm, conducting "Simple Tests" involves employing various techniques to verify the functionality and behavior of classes and their methods. OOP facilitates the creation of testable code by encapsulating behavior within classes, allowing for modular and focused testing.

To conduct simple tests in OOP, developers often utilize Python's built-in testing frameworks like unittest or pytest. These frameworks enable the creation of test cases, where each test case typically consists of methods that validate specific behaviors or functionalities within classes.

The process starts by defining test cases that encompass scenarios and expected outcomes for individual methods or functionalities of classes. In Python, test methods within these test cases are prefixed with "test_" to differentiate them as test functions.

Using assertions within these test methods is crucial. Assertions, such as `assertEqual`, `assertTrue`, or `assertFalse`, compare actual results against expected outcomes. For instance, `assertEqual` verifies if a method returns an expected value, ensuring it aligns with the intended functionality.

Incorporating mock objects or dependency injection techniques further aids in isolating and testing specific functionalities within classes. Mock objects simulate the behavior of dependencies, enabling controlled testing environments and facilitating the testing of complex interactions or edge cases.

Moreover, leveraging concepts like `setUp()` and `tearDown()` methods in test cases helps in preparing test fixtures or cleaning up resources before and after each test, ensuring a consistent and controlled testing environment.

The simplicity of these tests lies in their focus on individual methods or components within classes, ensuring that each unit of code behaves as expected. Simple tests not only verify the correctness of code but also enhance maintainability by providing a safety net for future modifications, encouraging clean and modular designs, and promoting confidence in the reliability of the software.

Here's an example demonstrating simple tests in object-oriented programming using Python's **unittest** framework:

Page 97 of 102	Ministry of Labor and Skills Author/Copyright	Object Oriented programming	Version -1
			November, 2023

Suppose we have a simple **Calculator** class with basic arithmetic operations: addition and subtraction. We'll create tests to verify the functionality of the **add()** and **subtract()** methods

```
import unittest

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

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

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        result = self.calc.add(5, 3)
        self.assertEqual(result, 8) # Asserting that 5 + 3 = 8
        result = self.calc.add(-2, 2)
        self.assertEqual(result, 0) # Asserting that -2 + 2 = 0

    def test_subtract(self):
        result = self.calc.subtract(10, 5)
        self.assertEqual(result, 5) # Asserting that 10 - 5 = 5
        result = self.calc.subtract(8, 3)
        self.assertEqual(result, 5) # Asserting that 8 - 3 = 5

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

Explanation:

- The **Calculator** class contains **add()** and **subtract()** methods performing addition and subtraction operations, respectively.
- The **TestCalculator** class, inheriting from **unittest.TestCase**, contains test methods prefixed with "test_".
- **setUp()** method initializes an instance of the **Calculator** class before each test method execution.
- **test_add()** and **test_subtract()** methods perform tests using **self.assertEqual()** to assert expected outcomes against actual results.

To run the tests, save the code in a Python file and execute it. The **unittest** framework will execute the tests, and if all assertions pass, it will display OK; otherwise, it will indicate which test failed and why. This example demonstrates simple tests verifying basic functionality

5.2. Embracing Corrections in Code and Documentation

It embodies a proactive and adaptive approach within the software development process, emphasizing the importance of acknowledging and addressing errors, inaccuracies, or improvements identified in both the codebase and accompanying documentation.

In the realm of code, embracing corrections involves fostering a culture that encourages developers to recognize and rectify issues promptly. This includes bugs, logical errors, or inefficiencies discovered during various phases of development or even after deployment. Adopting strategies like continuous integration and continuous delivery (CI/CD) facilitates the rapid identification and rectification of bugs, ensuring that corrections are integrated into the codebase seamlessly. Embracing corrections also involves soliciting feedback from peers through code reviews or collaborative discussions to identify potential improvements and implement necessary changes.

Similarly, in documentation, embracing corrections emphasizes the significance of maintaining accurate, up-to-date, and comprehensive documentation. It involves acknowledging that documentation can also contain errors or become outdated as software evolves. Actively encouraging and welcoming contributions or suggestions for improvements from team members or users ensures that documentation stays relevant and beneficial. Moreover, implementing version control for documentation allows for the tracking of changes and corrections, ensuring transparency and traceability of modifications made to the documentation.

Embracing corrections in both code and documentation not only improves the overall quality of the software but also fosters a culture of continuous improvement. It promotes accountability, transparency, and collaboration among team members, enabling a more resilient and adaptable development environment. Furthermore, acknowledging and rectifying mistakes promptly in both code and documentation demonstrate a commitment to delivering high-quality software products and reliable documentation, ultimately enhancing user satisfaction and trust in the software. Overall, embracing corrections is an integral part of the iterative and evolutionary nature of software development, contributing to the continual enhancement and refinement of software products and their accompanying documentation.

Self-Check:5

I. Say TRUE or FALSE

1. Embracing corrections in code and documentation primarily focuses on ignoring identified errors to speed up software deployment.
2. Test reports in software testing summarize the testing activities, outcomes, and potential improvements identified during the testing phase.
3. Simple tests in Python's object-oriented programming involve verifying the functionality of classes and methods using assertions and mock objects.
4. Continuous integration and continuous delivery (CI/CD) strategies aim to delay the identification and rectification of bugs in the codebase.

II. Choose the best answer from the provided choices

1. Which approach emphasizes acknowledging and addressing errors, inaccuracies, or improvements in both code and documentation?
 - A) Avoiding changes for maintaining stability
 - B) Embracing corrections
 - C) Ignoring identified issues
 - D) Speeding up software deployment
2. What purpose do test reports serve in software testing?
 - A) Summarizing testing activities and identifying errors
 - B) Speeding up the development process
 - C) Avoiding documentation of test outcomes
 - D) Delaying software deployment
3. What is a key aspect of simple tests in Python's object-oriented programming?
 - A) Focusing on complex system-level testing
 - B) Utilizing Python's built-in testing frameworks
 - C) Avoiding the use of assertions in test methods
 - D) Testing the entire software system at once

References

Books

Head First Object-Oriented Analysis and Design 2nd edition by Brett D. McLaughlin, Gary Pollice, David West

Python Crash Course 2nd edition by Eric Matthes

Learning Python 5th edition by Mark Lutz

Python Cookbook 3rd edition by David Beazley, Brian K. Jones

URLs

[Learn Python - Free Interactive Python Tutorial](#)

[Python For Beginners | Python.org](#)

[BeginnersGuide/Programmers - Python Wiki](#)

[46 Best resources to learn Python as of 2023 - Slant](#)

[12 Resources to Learn Python for Beginners \(geekflare.com\)](#)

<https://www.udemy.com/course/complete-python-bootcamp/>

[\(104\) Python Tutorials - YouTube](#)

Developer's Profile

No	Name	Qualification	Field of Study	Organization/ Institution	Mobile number	E-mail
1	Frew Atkilt	M-Tech	Network & Information Security	Bishoftu Polytechnic College	0911787374	frew.frikii@gmail.com
2	Gari Lencha	MSc	ICT Managment	Gimbi Polytechnic	0917819599	Garilenchal2@gmail.com
3	Kalkidan Daniel	BSc	Computer Science	Entoto Polytechnic	0978336988	kalkidaniel08@gmail.com
4	Solomon Melese	M-Tech	Computer Engineering	M/G /M/Polytechnic College	0918578631	solomonmelese6@gmail.com
5	Tewodros Girma	MSc	Information system	Sheno Polytechnic College	0912068479	girmatewodiros@gmail.com
6	Yohannes Gebeyehu	BSc	Information system	Entoto Polytechnic College	0923221273	yohannesgebeyehu73@gmail.com