



Curriculum title	Python Programmer
Curriculum code	900221-000-00-00
Module code	900221-000-00-KM-04, Intermediate Programming Principles in Python
NQF level	4
Credit(s)	6
Quality assurance functionary	QCTO - Quality Council for Trades and Occupations
Originator	MICT SETA
Qualification type	Skills Programme

Intermediate Programming Principles in Python

Learner Guide

Name	
Contact Address	
Telephone (H)	
Telephone (W)	
Cellular	

Email

Table of contents

Table of contents	3
Introduction	5
Purpose of the Knowledge Module	5
Knowledge Topic KM-04-KT01:	6
KM-04-KT01 Python Object Oriented Programming (OOP) (IAC0101)	7
1.1 Concept, definition and functions (KT0101) (IAC0101)	7
1.2 Syntax (KT0102) (IAC0101)	9
1.3 Class (KT0103) (IAC0101)	16
1.4 Object (KT0104) (IAC0101)	20
1.5 Constructors (KT0105) (IAC0101)	24
1.6 Minimal class (KT0106) (IAC0101)	31
1.7 Class attributes (KT0107) (IAC0101)	36
1.8 Methods in classes (KT0108) (IAC0101)	43
1.9 Inheritance (KT0109) (IAC0101)	50
1.10 Multiple inheritance (KT0110) (IAC0101)	60
1.11 Method overriding (KT0111) (IAC0101)	65
1.12 Operator overloading (KT0112) (IAC0101)	72
Formative Assessment Activity [1]	80
Knowledge Topic KM-04-KT02:	81
KM-04-KT02 Fundamental Concepts of Object-Oriented Programming (IAC0201)	82
2.1 Concept, definition and functions (KT0201) (IAC0201)	82
2.2 Class (KT0202) (IAC0201)	85
2.3 Object (KT0203) (IAC0201)	87
2.4 Method (KT0204) (IAC0201)	91
2.5 Inheritance (KT0205) (IAC0201)	96
2.6 Encapsulation (KT0206) (IAC0201)	102
2.7 Polymorphism (KT0207) (IAC0201)	107
2.8 Data abstraction (KT0208) (IAC0201)	113
Formative Assessment Activity [2]	117
Knowledge Topic KM-04-KT03:	117
KM-04-KT03 Basics of Algorithms (IAC0301)	119
3.1 Concept, definition and functions (KT0301) (IAC0301)	119
3.2 Dynamic programming algorithms (KT0302) (IAC0301)	123
3.3 Searching and sorting algorithms (KT0303) (IAC0301)	130
3.4 Mathematical algorithms (KT0304) (IAC0301)	141
3.5 Data structures algorithms (KT0305) (IAC0301)	147
3.6 Flow charts (on paper) (KT0306) (IAC0301)	154
Formative Assessment Activity [number]	158
Knowledge Topic KM-04-KT04:	159
KM-04-KT04 File Handling (IAC0401)	160
4.1 Concept, definition and functions (KT0401) (IAC0401)	160

4.2 Syntax (KT0402) (IAC0401)	168
4.3 Inbuilt function (KT0403) (IAC0401)	174
4.4 File operation (KT0404) (IAC0401)	180
4.5 Directory (KT0405) (IAC0401)	188
4.6 File modes (KT0406) (IAC0401)	195
4.7 Zip files (KT0407) (IAC0401)	204
4.8 Readline() (KT0408) (IAC0401)	212
4.9 Binary files (KT0409) (IAC0401)	217
Formative Assessment Activity [4]	226
Knowledge Topic KM-04-KT05:	227
KM-04-KT05 Exception Handling (IAC0501)	228
5.1 Concept, definition and functions (KT0501) (IAC0501)	228
5.2 Syntax (KT0502) (IAC0501)	239
5.3 Exceptions (KT0503) (IAC0501)	243
5.4 Common examples of exception (KT0504) (IAC0501)	250
5.5 Rules of exceptions (KT0505) (IAC0501)	260
5.6 Exception handling (KT0506) (IAC0501)	269
5.7 Keywords (try, catch, finally) (KT0507) (IAC0501)	274
5.8 Important Python errors and exceptions (KT0508) (IAC0501)	280
5.9 Error vs exception (KT0509) (IAC0501)	292
5.10 User-defined exception (KT0510) (IAC0501)	298
Formative Assessment Activity [5]	305
References	306

Introduction

Welcome to this skills programme!

This guide will help you understand the content we will cover. You will also complete several class activities as part of the formative assessment process. These activities give you a safe space to practise and explore new skills.

Make the most of this opportunity to gather information for your self-study and practical learning. In some cases, you may need to research and complete tasks in your own time.

Take notes as you go, and share your insights with your classmates. Sharing knowledge helps you and others deepen your understanding and apply what you've learned.

Purpose of the Knowledge Module

The main focus of the learning in this knowledge module is to build an understanding of the principles of intermediate programming with Python programming language

This learner guide will enable you to gain an understanding of the following topics. (The relative weightings within the module are also shown in the following table.)

Code	Topic	Weight
KM-04-KT01	Python Object Oriented Programming (OOP)	25%
KM-04-KT02	Fundamental concepts of Object-oriented programming	20%
KM-04-KT03	Basics of Algorithms	10%
KM-04-KT04	File handling	20%
KM-04-KT05	Exception handling	25%

Knowledge Topic KM-04-KT01:

Topic Code	KM-04-KT01
Topic	Intermediate Programming Principles in Python
Weight	25%

This knowledge topic will cover the following topic elements:

- KT0101 Concept, definition, and functions
- KT0102 Syntax
- KT0103 Class
- KT0104 Object
- KT0105 Constructors
- KT0106 Minimal class
- KT0107 Class attributes
- KT0108 Methods in classes
- KT0109 Inheritance
- KT0110 Multiple inheritance
- KT0111 Method overriding
- KT0112 Operator overloading

After working through this knowledge topic, your competence will be assessed according to the following criterion/criteria:

- IAC0101 Definitions, functions and features of Python OOP are understood and explained

Getting Started

To begin this module, please click on the link to the [**Learner Workbook**](#). This will prompt you to make a copy of the document, where you'll complete all formative and summative assessments throughout this module. Make sure to save your copy in a

secure location, as you'll be returning to it frequently. Once you've made a copy, you're ready to start working through the module materials. You will be required to upload this document to your **GitHub folder** once all formative and summative assessments are complete for your facilitator to review and mark.



Take note

This module has a total of 414 marks available. To meet the passing requirements, you will need to achieve a minimum of 331 marks, which represents 60% of the total marks. Achieving this threshold will ensure that you have met the necessary standards for this module.

KM-04-KT01 Python Object Oriented Programming (OOP) (IAC0101)

1.1 Concept, definition and functions (KT0101) (IAC0101)

Python Object-Oriented Programming (OOP) is a programming paradigm that organises software design around data, or objects, rather than functions and logic. In Python, OOP allows you to create reusable code by defining classes, which are blueprints for objects, encapsulating data (attributes) and behaviours (methods) together.

Key Concepts of Python OOP:

1. Classes and Objects:

- o Class: A blueprint or template for creating objects. It defines a set of attributes and methods that the created objects will have.
- o Object: An instance of a class. It's a concrete entity based on the class blueprint.

2. Encapsulation:

- o The bundling of data (attributes) and methods that operate on that data within one unit (class), restricting direct access to some of the object's components. This protects the integrity of the data.

3. Inheritance:

- o A mechanism where a new class (child class) derives properties and behaviour (methods) from an existing class (parent class). This promotes code reusability and hierarchical classifications.

4. Polymorphism:

- o The ability of different classes to be treated as instances of the same class through a common interface. Methods can perform differently based on the object they are acting upon, even if they share the same name.

5. Abstraction:

- o Hiding the complex reality while exposing only the necessary parts. It focuses on what an object does instead of how it does it.

Functions in Python OOP (Methods):

- Methods are functions defined within a class that describe the behaviours of an object.
 - Instance Methods: Operate on instances of the class (objects). They can access and modify object attributes.
 - Class Methods: Use the `@classmethod` decorator and operate on the class itself, not instances. They can modify class state that applies across all instances.
 - Static Methods: Use the `@staticmethod` decorator. They do not access or modify class or instance state and are utility functions within the class namespace.
 - Special Methods: Also known as magic methods (e.g., `__init__`, `__str__`, `__repr__`), these have double underscores and allow custom behaviour for built-in operations.

By using OOP in Python, you can create programs that are more modular, scalable, and easier to maintain. It mirrors real-world entities, making complex software development more intuitive by modelling classes and objects that interact in defined ways.

1.2 Syntax (KT0102) (IAC0101)

Python's OOP syntax allows you to define classes, create objects, and implement the principles of encapsulation, inheritance, and polymorphism.

1. Defining a Class

Use the class keyword followed by the class name and a colon. Class names are usually written in PascalCase.

```
class MyClass:  
    pass # An empty class
```

2. The __init__ Method (Constructor)

The `__init__` method initializes new objects of the class. The `self`-parameter refers to the instance of the object itself.

```
class Person:  
    def __init__(self, name, age):  
        self.name = name # Instance attribute  
        self.age = age   # Instance attribute
```

3. Creating an Object (Instance)

Instantiate an object by calling the class name with required arguments.

```
person1 = Person("Alice", 30)
```

4. Instance Attributes and Methods

- Instance Attributes: Variables unique to each object.
- Instance Methods: Functions that operate on instance attributes.

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

    def greet(self):
        print(f"Hello, I'm {self.name}.")

person1 = Person("Alice")
person1.greet() # Output: Hello, I'm Alice.
```

5. Class Attributes

Attributes shared across all instances of the class.

```
class Person:
    species = "Homo sapiens" # Class attribute

print(Person.species) # Output: Homo sapiens
```

6. Inheritance

Create a new class that inherits attributes and methods from another class.

```
class Employee(Person):
    def __init__(self, name, employee_id):
        super().__init__(name) # Calls the __init__ of the parent class
        self.employee_id = employee_id
```

7. Overriding Methods

A child class can override methods from the parent class.

```
class Employee(Person):
    def greet(self):
        print(f"Hello, I'm {self.name}, and my ID is {self.employee_id}.")

employee1 = Employee("Bob", "E123")
employee1.greet() # Output: Hello, I'm Bob, and my ID is E123.
```

8. Access Modifiers (Naming Conventions)

- Public Attributes/Methods: Standard naming (e.g., self.name).
- Protected Attributes/Methods: Prefix with a single underscore (e.g., self._name).
- Private Attributes/Methods: Prefix with double underscores (e.g., self.__name).

```
class Person:  
    def __init__(self, name):  
        self.__name = name # Private attribute  
  
    def get_name(self):  
        return self.__name
```

9. Static Methods and Class Methods

- Static Methods: Defined with @staticmethod. Do not access instance or class data.

```
class MathOperations:  
    @staticmethod  
    def add(a, b):  
        return a + b  
  
print(MathOperations.add(5, 3)) # Output: 8
```

- Class Methods: Defined with @classmethod. Access the class itself via cls.

```
class Person:
    count = 0

    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod
    def get_count(cls):
        return cls.count

print(Person.get_count()) # Output: 0
person1 = Person("Alice")
print(Person.get_count()) # Output: 1
```

10. Magic Methods (Special Methods)

Special methods with double underscores that enable custom behaviour.

- `__str__(self)`: Defines the string representation of the object.

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

    def __str__(self):
        return f"Person named {self.name}"

person1 = Person("Alice")
print(person1) # Output: Person named Alice
```

- `__repr__(self)`: Defines the official string representation of the object, used in debugging.

11. Polymorphism

Different classes can have methods with the same name, and Python can use them interchangeably.

```

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

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

def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

animal_sound(dog) # Output: Woof!
animal_sound(cat) # Output: Meow!

```

Example: Combining Concepts

```

class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def info(self):
        return f"{self.make} {self.model}"


class Car(Vehicle):
    def info(self):
        return f"Car: {super().info()}"


class Motorcycle(Vehicle):
    def info(self):
        return f"Motorcycle: {super().info()}"


vehicles = [Car("Toyota", "Corolla"), Motorcycle("Honda", "CBR")]

```

```
for vehicle in vehicles:  
    print(vehicle.info())  
  
# Output:  
# Car: Toyota Corolla  
# Motorcycle: Honda CBR
```

Summary of Syntax:

- Defining a Class:

```
class ClassName:  
    # Class body
```

- Constructor Method:

```
def __init__(self, parameters):  
    # Initialization code
```

- Instance Method:

```
def method_name(self, parameters):  
    # Method body
```

- Creating an Object:

```
obj = ClassName(arguments)
```

- Inheritance:

```
class SubClassName(ParentClassName):  
    # Subclass body
```

- Accessing Parent Class:

```
super().method_name(parameters)
```

- Static Method:

```
@staticmethod  
def static_method_name(parameters):  
    # Method body
```

- Class Method:

```
@classmethod  
def class_method_name(cls, parameters):  
    # Method body
```

By understanding and utilizing this syntax, you can effectively apply object-oriented principles in Python to create organised, reusable, and efficient code.

1.3 Class (KT0103) (IAC0101)

A class is a fundamental concept that serves as a blueprint or template for creating objects. It defines a set of attributes (data) and methods (functions) that characterize any object instantiated from the class.

Key Aspects of a Class:

1. Blueprint for Objects:

- A class outlines the structure and behaviour that its objects (instances) will have.
- It doesn't represent any particular object but provides the guidelines for creating them.

2. Attributes and Methods:

- Attributes: Variables that hold the data related to the class and its objects.
 - Class Attributes: Shared by all instances of the class.
 - Instance Attributes: Unique to each object.
- Methods: Functions defined within a class that describe the behaviours of the objects.
 - They can manipulate object attributes and perform actions.

3. Instantiation:

- Creating an object from a class is called instantiation.
- Each object is an instance of a class and can have its own unique data.

4. Encapsulation:

- Classes encapsulate data and methods, bundling them into a single unit.
- This promotes modularity and code reusability.

Basic Syntax of a Class in Python:

```
class ClassName:
    # Class attribute (shared by all instances)
    class_attribute = "This is a class attribute"

    def __init__(self, instance_attribute):
        # Instance attribute (unique to each instance)
        self.instance_attribute = instance_attribute

    # Instance method
    def instance_method(self):
        print(f"This is an instance method accessing {self.instance_attribute}")

    # Class method
    @classmethod
    def class_method(cls):
        print(f"This is a class method accessing {cls.class_attribute}")

    # Static method
    @staticmethod
    def static_method():
        print("This is a static method that doesn't access class or instance attributes")
```

- class ClassName:: Defines a new class named ClassName.
- __init__ Method: The constructor method that initializes new objects.
 - self: Refers to the instance of the class.
- Attributes: Variables that hold data.
 - self.instance_attribute: An attribute unique to the instance.
 - class_attribute: An attribute shared across all instances.

Example: Defining and Using a Class

```
class Person:
    # Class attribute
    species = "Homo sapiens"

    # Constructor method
    def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age    # Instance attribute

    # Instance method
    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Instantiating objects from the class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Accessing instance attributes and methods
person1.greet() # Output: Hello, my name is Alice and I am 30 years old.
person2.greet() # Output: Hello, my name is Bob and I am 25 years old.

# Accessing class attribute
print(person1.species) # Output: Homo sapiens
print(person2.species) # Output: Homo sapiens
```

- Creating Objects: person1 and person2 are instances of the Person class.
- Accessing Methods: Calling person1.greet() invokes the greet method for person1.
- Accessing Attributes: person1.name and person1.age access the instance attributes.

Why Use Classes in Python OOP?

- Organisation: Classes help in organising code into logical sections, making it more readable and maintainable.
- Reusability: Once a class is defined, you can create multiple instances without rewriting code.
- Encapsulation: Bundling data and methods together restricts direct access to some of the object's components, which can prevent accidental modification.
- Inheritance and Polymorphism: Classes can inherit from other classes, allowing for hierarchical relationships and code reuse. Polymorphism allows methods to function differently based on the object invoking them.

Summary

- A class is like a blueprint for creating objects with specific attributes and methods.
- It encapsulates data for the object and functions to manipulate that data.
- Classes promote code reusability, organisation, and modularity in programming.

Additional Example: Class with Class Method and Static Method

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

    @classmethod
    def info(cls):
        print("This is a calculator class.")

# Using static method without creating an instance
result = Calculator.add(5, 7)
print(result) # Output: 12

# Using class method
Calculator.info() # Output: This is a calculator class.
```

- Static Methods: Defined using `@staticmethod`, they do not access or modify class or instance data.
- Class Methods: Defined using `@classmethod`, they have access to the class itself via `cls`.

With knowledge on classes, you gain the ability to model real-world entities and complex systems within your code, leading to more efficient and effective programming practices.

1.4 Object (KT0104) (IAC0101)

An object is a fundamental entity that represents an instance of a class. When you create an object, you are bringing to life the blueprint defined by a class, assigning specific values to the attributes, and enabling the use of its methods.

Key Characteristics of an Object:

1. Instance of a Class:
 - An object is created from a class, inheriting its structure and behaviour.
 - Each object can have its own unique values for the attributes defined in the class.
2. Attributes (Data):
 - Objects store data in variables called attributes or properties.
 - These attributes represent the state or characteristics of the object.
3. Methods (Behaviour):
 - Objects have methods, which are functions defined within the class.
 - Methods define the behaviours or actions that the object can perform.
4. Encapsulation of Data and Behaviour:
 - Objects encapsulate both data and the methods that operate on that data.
 - This encapsulation promotes modularity and code reusability.

Creating an Object (Instantiation):

To create an object, you instantiate a class by calling it like a function. The `__init__` method (constructor) initializes the object's attributes.

```
class Person:
    def __init__(self, name, age):
        self.name = name # Instance attribute
        self.age = age   # Instance attribute

    def introduce(self):
        print(f"Hello, my name is {self.name} and I'm {self.age} years old.")

# Creating objects (instances of the Person class)
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
```

- `person1` and `person2` are objects (instances) of the `Person` class.
- Each object has its own name and age attributes.

Accessing Attributes and Methods:

You can interact with an object's attributes and methods using dot notation.

```
# Accessing attributes
print(person1.name) # Output: Alice
print(person2.age)  # Output: 25

# Calling methods
person1.introduce() # Output: Hello, my name is Alice and I'm 30 years old.
person2.introduce() # Output: Hello, my name is Bob and I'm 25 years old.
```

Objects are Independent:

Each object maintains its own state. Modifying one object's attributes doesn't affect other objects.

```
person1.age += 1
print(person1.age) # Output: 31
print(person2.age) # Output: 25 # Unchanged
```

Objects Can Interact:

Objects can interact with one another through their methods and attributes, enabling complex behaviours.

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

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

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"{self.owner} withdrew ${amount}.")
        else:
            print(f"{self.owner} has insufficient funds.")

# Creating objects
account1 = BankAccount("Alice", 100)
account2 = BankAccount("Bob", 50)

# Objects interacting
account1.deposit(50)      # Output: Alice deposited $50.
account2.withdraw(20)      # Output: Bob withdrew $20.
print(account1.balance)   # Output: 150
print(account2.balance)   # Output: 30
```

Summary:

- An object is an instance of a class, embodying the attributes and methods defined by the class.

- Objects encapsulate data (attributes) and behaviours (methods), allowing for organised and modular code.
- Instantiation is the process of creating an object from a class.
- Objects are central to Python OOP, enabling you to model real-world entities and interactions.

Analogy:

Think of a class as a cookie cutter (blueprint) and objects as the cookies made from it. While the cookie cutter defines the shape and size, each cookie (object) can have different decorations (attribute values), but all share the same basic form.

1.5 Constructors (KT0105) (IAC0101)

A constructor is a special method used to initialize newly created objects. The constructor's primary role is to set up the initial state of an object by assigning values to its attributes. This ensures that the object starts its life in a valid and usable state.

Key Characteristics of Constructors:

1. Special Method `__init__`:

- The constructor method in Python is named `__init__`, short for "initialize."
- It is automatically called when you create a new instance of a class.
- The double underscores before and after init indicate that it's a special or "magic" method in Python.

2. Initialization of Attributes:

- Constructors set the initial values of object attributes.
- They can accept parameters to customize the object's initial state.
- By initializing attributes, constructors help maintain data integrity and encapsulation.

3. The `self`-Parameter:

- The first parameter of the constructor is always `self`, which refers to the instance being created.
- `self` allows you to access and assign values to the object's attributes within the class.

Basic Syntax of a Constructor:

```
class ClassName:  
    def __init__(self, parameters):  
        # Initialization code  
        self.attribute1 = value1  
        self.attribute2 = value2
```

- class ClassName:: Defines a new class named ClassName.
- def __init__(self, parameters):: Defines the constructor method.
- self.attribute = value: Assigns a value to an instance attribute.

Example: Using a Constructor to Initialize Objects

```
class Person:  
    def __init__(self, name, age):  
        self.name = name # Assigning the 'name' attribute  
        self.age = age   # Assigning the 'age' attribute  
  
    def introduce(self):  
        print(f"Hello, my name is {self.name} and I'm {self.age} years old.")  
  
# Creating instances (objects) of the Person class  
person1 = Person("Alice", 30)  
person2 = Person("Bob", 25)  
  
# Calling methods on the objects  
person1.introduce() # Output: Hello, my name is Alice and I'm 30 years old.  
person2.introduce() # Output: Hello, my name is Bob and I'm 25 years old.
```

Explanation:

- Instantiation: When `person1 = Person("Alice", 30)` is executed, Python:
 - Calls the `__init__` method of the Person class.
 - Passes `self` (the new object), "Alice" as name, and 30 as age.
 - Initializes `self.name` and `self.age` with the provided values.
- Method Invocation: The `introduce` method accesses the initialized attributes to display the message.

Why Use Constructors?

- Ensure Proper Initialization:
 - Objects start with all necessary attributes properly set.
 - Reduces the risk of errors from uninitialized attributes.
- Encapsulation and Data Integrity:
 - Encapsulate initialization logic within the class.
 - Enforce constraints or validation during object creation.
- Customizable Object Creation:
 - Allow different objects to have different initial states.
 - Support default values and optional parameters.

Default Parameters in Constructors:

You can provide default values for constructor parameters, making them optional during instantiation.

```
class Rectangle:
    def __init__(self, width=1, height=1):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Creating objects with and without specifying all parameters
rect1 = Rectangle(5, 10)
rect2 = Rectangle(3)
rect3 = Rectangle()

print(rect1.area()) # Output: 50
print(rect2.area()) # Output: 3
print(rect3.area()) # Output: 1
```

Explanation:

- rect1: Width and height provided.
- rect2: Width provided; height defaults to 1.
- rect3: Both width and height default to 1.

Constructors in Inheritance:

When dealing with inheritance, constructors play a crucial role in initializing both parent and child class attributes.

```

class Animal:
    def __init__(self, species):
        self.species = species

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__("Canine") # Calls the constructor of the parent class
        self.name = name
        self.breed = breed

    def info(self):
        print(f"{self.name} is a {self.breed} belonging to the {self.species} species.")

dog = Dog("Buddy", "Golden Retriever")
dog.info() # Output: Buddy is a Golden Retriever belonging to the Canine species.

```

Explanation:

- `super().__init__("Canine")`: Calls the constructor of the `Animal` class to initialize `species`.
- Child Class Initialization: The `Dog` class initializes additional attributes `name` and `breed`.

Multiple Constructors (Overloading):

Python does not support method overloading directly, but you can mimic multiple constructors using default parameters or by checking the number and type of arguments.

```

class Person:
    def __init__(self, name, age=None):
        self.name = name
        if age is not None:
            self.age = age
        else:
            self.age = "Not specified"

person1 = Person("Alice", 30)
person2 = Person("Bob")

print(f"{person1.name}, Age: {person1.age}") # Output: Alice, Age: 30
print(f"{person2.name}, Age: {person2.age}") # Output: Bob, Age: Not specified

```

Explanation:

- Flexible Initialization: The constructor can handle cases when age is provided or omitted.
- Default Handling: Sets age to "Not specified" if no value is given.

Destructor Method (`__del__`):

- While not a constructor, the `__del__` method is a special method called when an object is about to be destroyed.
- In Python, explicit destructors are less common due to automatic garbage collection.
- Use with caution, as reliance on `__del__` can lead to unexpected behaviours.

Summary:

- Constructors (`__init__`):
 - Special methods that initialize new objects.
 - Automatically invoked during object creation.
 - Set up initial attribute values and enforce any setup logic.
- Usage:
 - Define necessary parameters for object creation.
 - Assign values to instance attributes using `self`.

- Can include default parameters for flexibility.
- Best Practices:
 - Keep constructors focused on initialization.
 - Avoid performing complex operations or side effects in constructors.
 - Use other methods for additional setup if needed.

Analogy:

Think of a constructor like the setup instructions for assembling furniture. When you buy a piece of furniture (class), the instructions (constructor) guide you to assemble it correctly, ensuring all parts (attributes) are in place and the item (object) is ready for use.

1.6 Minimal class (KT0106) (IAC0101)

A minimal class refers to the simplest form of a class that can be defined and used to create objects. It includes the essentials needed to define a class without any additional attributes or methods. Minimal classes are often used for illustrative purposes, simple data storage, or as placeholders during the development process.

Key Characteristics of a Minimal Class:

1. Definition Using the class Keyword:

- A minimal class is defined using the class keyword followed by the class name and a colon.

2. Empty Body:

- The class body contains no attributes or methods. To create an empty class, the pass statement is used as a placeholder to indicate that no action is taken.

3. Instantiation:

- Even without attributes or methods, you can create instances (objects) of a minimal class.

Basic Syntax of a Minimal Class:

```
class MyClass:  
    pass # Indicates an empty class body
```

- class MyClass:: Defines a new class named MyClass.
- pass: A no-operation statement used here to create an empty class body.

Example: Creating and Using a Minimal Class

```
# Defining a minimal class
class EmptyClass:
    pass

# Creating an instance (object) of the minimal class
empty_object = EmptyClass()

# Checking the type of the object
print(type(empty_object)) # Output: <class '__main__.EmptyClass'>
```

Explanation:

- Class Definition: EmptyClass is defined with no attributes or methods.
- Instantiation: empty_object is an instance of EmptyClass.
- Type Checking: Using type(empty_object) confirms that empty_object is indeed an instance of EmptyClass.

Use Cases for Minimal Classes:

1. Placeholders During Development:
 - When designing a system, you might define minimal classes first and later add attributes and methods as needed.
2. Simple Data Containers:
 - Minimal classes can act as simple containers to group related data together without behaviour.
3. Inheritance and Extension:
 - They can serve as base classes that other classes inherit from, allowing you to build more complex structures incrementally.

Enhancing a Minimal Class: Adding Attributes and Methods

While a minimal class starts empty, you can easily extend it by adding attributes and methods to give it functionality.

Example: Extending the Minimal Class

```
class EmptyClass:  
    pass  
  
# Adding attributes dynamically  
empty_object = EmptyClass()  
empty_object.name = "Alice"  
empty_object.age = 30  
  
# Adding a method dynamically  
def greet(self):  
    print(f"Hello, my name is {self.name} and I'm {self.age} years  
old.")  
  
# Binding the method to the instance  
import types  
empty_object.greet = types.MethodType(greet, empty_object)  
  
# Using the added method  
empty_object.greet() # Output: Hello, my name is Alice and I'm 30  
years old.
```

Explanation:

- Dynamic Attribute Addition: Attributes name and age are added to empty_object after its creation.
- Dynamic Method Addition: A greet method is defined outside the class and then bound to empty_object using types.MethodType.
- Method Invocation: Calling empty_object.greet() executes the dynamically added method.

Note: While Python allows adding attributes and methods to instances dynamically, it's generally better practice to define them within the class for clarity and maintainability.

Benefits of Starting with a Minimal Class:

1. Simplicity: Helps beginners understand the basic structure of a class without being overwhelmed by additional details.
2. Flexibility: Allows developers to incrementally build up class complexity as needed.
3. Focus on Fundamentals: Emphasizes the concept that classes are blueprints for creating objects, even in their simplest form.

Summary:

- A minimal class in Python OOP is the simplest form of a class with no attributes or methods.
- Defined using the `class` keyword followed by a `pass` statement.
- Useful as placeholders, simple data containers, or starting points for more complex classes.
- Even minimal classes can be instantiated and extended dynamically, though it's better to define attributes and methods within the class for clarity.

Analogy:

Think of a minimal class as an empty blueprint for a building. While the blueprint doesn't show any rooms or features initially, it provides the foundational structure upon which you can later add specific details like rooms, doors, and windows.

1.7 Class attributes (KT0107) (IAC0101)

Class attributes are variables that are shared across all instances of a class. They belong to the class itself rather than any individual object created from the class. Class attributes are useful for storing data that is common to all instances of a class, promoting memory efficiency and consistency.

Key Characteristics of Class Attributes:

1. Shared Among All Instances:

- Class attributes are the same for every object instantiated from the class unless explicitly overridden.
- Modifying a class attribute affects all instances that haven't overridden the attribute.

2. Defined Outside Methods:

- They are defined within the class body but outside of any instance methods like `__init__`.

3. Accessed via Class Name or Instance:

- You can access class attributes using the class name or an instance of the class.
- It's common practice to reference them via the class name for clarity when the shared nature is important.

Defining Class Attributes:

```
class ClassName:  
    class_attribute = value # Class attribute definition  
  
    def __init__(self, parameters):  
        self.instance_attribute = value # Instance attribute
```

- `class_attribute = value`: Defines a class attribute.
- `self.instance_attribute = value`: Defines an instance attribute within the constructor.

Example: Using Class Attributes

```
class Car:  
    wheels = 4 # Class attribute shared by all instances  
  
    def __init__(self, make, model):  
        self.make = make      # Instance attribute  
        self.model = model    # Instance attribute  
  
  
# Creating instances of the Car class  
car1 = Car("Toyota", "Corolla")  
car2 = Car("Honda", "Civic")  
  
# Accessing class attribute via instance  
print(car1.wheels) # Output: 4  
print(car2.wheels) # Output: 4  
  
# Accessing class attribute via class name  
print(Car.wheels) # Output: 4
```

Explanation:

- Class Attribute (wheels): Defined once and shared among all Car instances.
- Instance Attributes (make, model): Unique to each instance (car1, car2).

Modifying Class Attributes:

- Via Class Name:
 - Changes affect all instances that haven't overridden the attribute.

```
Car.wheels = 6
print(car1.wheels) # Output: 6
print(car2.wheels) # Output: 6
```

- Via Instance:
 - Assigning a value to a class attribute via an instance creates or modifies an instance attribute with the same name, shadowing the class attribute.

```
car1.wheels = 5 # This creates an instance attribute 'wheels' for car1
print(car1.wheels) # Output: 5 (Instance attribute)
print(car2.wheels) # Output: 6 (Still refers to class attribute)
print(Car.wheels) # Output: 6 (Class attribute remains unchanged)
```

Note: Modifying class attributes via instances can lead to confusion. It's better to modify them using the class name when intending to change the shared value.

Use Cases for Class Attributes:

1. Constants and Default Values:
 - Store values common to all instances, like default settings or constants.
2. Tracking Data Across Instances:
 - Keep track of data that pertains to the class, such as counting the number of instances.

```
class User:
    total_users = 0 # Class attribute to count instances

    def __init__(self, name):
        self.name = name
        User.total_users += 1 # Increment class attribute

user1 = User("Alice")
user2 = User("Bob")
print(User.total_users) # Output: 2
```

3. Shared Configuration:

- o Define settings or configurations that apply to all instances unless specifically overridden.

Class Attributes vs. Instance Attributes:

- Class Attributes:

- o Scope: Shared across all instances.
- o Definition Location: Inside the class body, outside any methods.
- o Access: Can be accessed via the class name or an instance.
- o Modification: Changing via the class affects all instances; changing via an instance creates a new instance attribute.

- Instance Attributes:

- o Scope: Unique to each instance.
- o Definition Location: Typically defined within methods (like `__init__`) using `self`.
- o Access: Accessed via the instance.
- o Modification: Changes affect only that instance.

Illustrative Example:

```
class Employee:  
    company_name = "TechCorp"  # Class attribute  
  
    def __init__(self, name):  
        self.name = name  # Instance attribute  
  
  
# Creating instances  
emp1 = Employee("Alice")  
emp2 = Employee("Bob")  
  
# Accessing attributes  
print(emp1.company_name)  # Output: TechCorp  
print(emp2.company_name)  # Output: TechCorp  
  
# Modifying class attribute via class name  
Employee.company_name = "Innovatech"  
  
print(emp1.company_name)  # Output: Innovatech  
print(emp2.company_name)  # Output: Innovatech  
  
# Modifying class attribute via instance (creates instance attribute)  
emp1.company_name = "AlphaTech"  
  
print(emp1.company_name)      # Output: AlphaTech (Instance attribute)  
print(emp2.company_name)      # Output: Innovatech (Class attribute remains)  
print(Employee.company_name)  # Output: Innovatech
```

Best Practices:

- Use Class Attributes for Shared Data:
 - When you need a value to be consistent across all instances unless explicitly overridden.

- Avoid Unintended Shadowing:
 - Be cautious when assigning values to class attributes via instances to prevent accidental creation of instance attributes with the same name.
- Reference via Class Name When Appropriate:
 - Accessing or modifying class attributes using the class name enhances readability and clarity.

Summary:

- Class Attributes are variables defined within a class but outside any methods, shared by all instances of the class.
- They are ideal for storing data common to all objects, like constants, configurations, or class-wide statistics.
- Modifying a class attribute affects all instances that refer to it, unless an instance has its own attribute with the same name.
- Understanding the distinction between class attributes and instance attributes is crucial for effective class design in Python OOP.

Analogy:

Imagine a blueprint for building houses:

- Class Attribute (Blueprint Specification): The number of floors planned for all houses is set to 2 by default in the blueprint (class attribute).
- Instance Attribute (House Customization): Each house can have different paint colors (instance attributes) based on the owner's preference.

If the blueprint changes the default number of floors to 3, all future houses built from it will have 3 floors. Existing houses remain unaffected unless they undergo renovation (modification of instance attributes).

1.8 Methods in classes (KT0108) (IAC0101)

methods are functions defined within a class that describe the behaviours and actions that an object (instance of the class) can perform. Methods allow objects to interact with their own data (attributes) and with other objects, encapsulating both data and functionality within a single unit.

Key Characteristics of Methods in Classes

1. Defined Inside a Class:
 - o Methods are functions declared within the body of a class.
 - o They are associated with the class and its instances.
2. Access to Object Data:
 - o Methods can access and modify the object's attributes.
 - o They use self (or cls for class methods) to refer to the instance or class.
3. Encapsulation of Behaviour:
 - o Methods encapsulate behaviours relevant to the object.
 - o They enable interaction with the object's state in a controlled manner.

Types of Methods in Python Classes

1. Instance Methods:
 - o The most common type of method.
 - o Defined without any decorators.
 - o The first parameter is always self, referring to the instance.
 - o Can access and modify instance attributes.

```

class Person:
    def __init__(self, name):
        self.name = name # Instance attribute

    def greet(self):
        print(f"Hello, my name is {self.name}.") # Accessing instance attribute

person = Person("Alice")
person.greet() # Output: Hello, my name is Alice.

```

2. Class Methods:

- o Defined using the @classmethod decorator.
- o The first parameter is cls, referring to the class itself.
- o Can access and modify class attributes.
- o Useful for factory methods that create instances in a specific way.

```

class Person:
    species = "Homo sapiens" # Class attribute

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

    @classmethod
    def change_species(cls, new_species):
        cls.species = new_species

Person.change_species("Homo neanderthalensis")
print(Person.species) # Output: Homo neanderthalensis

```

3. Static Methods:

- o Defined using the @staticmethod decorator.
- o Do not receive self or cls as the first parameter.
- o Cannot access or modify instance or class attributes.
- o Used for utility functions related to the class's context.

```
class MathOperations:  
    @staticmethod  
    def add(a, b):  
        return a + b  
  
result = MathOperations.add(5, 3)  
print(result) # Output: 8
```

4. Special Methods (Magic Methods):

- o Also known as dunder methods (double underscores).
- o Enable custom behaviour for built-in operations.
- o Examples include `__init__`, `__str__`, `__repr__`, `__len__`, `__eq__`.

```
class Book:  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author  
  
    def __str__(self):  
        return f'{self.title} by {self.author}'  
  
book = Book("1984", "George Orwell")  
print(book) # Output: '1984' by George Orwell
```

Defining Methods in Classes

Instance Methods

- Syntax:

```
class ClassName:  
    def method_name(self, parameters):  
        # Method body  
        pass
```

- Example:

```
class Circle:  
    def __init__(self, radius):  
        self.radius = radius  
  
    def area(self):  
        return 3.1416 * self.radius ** 2  
  
circle = Circle(5)  
print(circle.area()) # Output: 78.54
```

Class Methods

- Syntax:

```
class ClassName:  
    @classmethod  
    def method_name(cls, parameters):  
        # Method body  
        pass
```

- Example:

```
class Date:  
    def __init__(self, year, month, day):  
        self.year = year  
        self.month = month  
        self.day = day  
  
    @classmethod  
    def from_string(cls, date_string):  
        year, month, day = map(int, date_string.split('-'))  
        return cls(year, month, day)  
  
date = Date.from_string("2023-10-05")  
print(date.year, date.month, date.day) # Output: 2023 10 5
```

Static Methods

- Syntax:

```
class ClassName:  
    @staticmethod  
    def method_name(parameters):  
        # Method body  
        pass
```

- Example:

```
class Validator:  
    @staticmethod  
    def is_valid_age(age):  
        return 0 <= age <= 120  
  
print(Validator.is_valid_age(25)) # Output: True
```

Special Methods

- Common Special Methods:
 - `__init__(self, ...)`: Constructor, initializes new instances.
 - `__str__(self)`: Defines behaviour for `str()` and `print()`.
 - `__repr__(self)`: Defines the official string representation.
 - `__len__(self)`: Defines behaviour for `len()`.

- `__eq__(self, other)`: Defines behaviour for equality ==.
- Example:

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

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

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2
print(v3) # Output: Vector(7, 10)
```

Accessing Methods

- Instance Methods:

```
instance = ClassName(arguments)
instance.method_name(arguments)
```

- Class Methods:

```
ClassName.method_name(arguments)
# or
instance.method_name(arguments)
```

- Static Methods:

```
ClassName.method_name(arguments)
# or
instance.method_name(arguments)
```

Why Use Different Types of Methods?

1. Instance Methods:

- o Operate on data unique to each instance.
- o Modify the object's state or query information from it.

2. Class Methods:

- o Operate on the class itself rather than instances.
- o Useful for factory methods and altering class state.

3. Static Methods:

- o Utility functions that make sense to be part of the class but don't need access to class or instance data.

4. Special Methods:

- o Integrate with Python's built-in features.
- o Provide custom behaviour for operators and built-in functions.

Best Practices

- Use Descriptive Method Names:
 - o Method names should clearly indicate their purpose.
- Keep Methods Focused:
 - o Each method should perform a specific task.
- Access Control:
 - o Prefix methods with a single underscore _ to indicate that they are intended for internal use (a convention).

- Avoid Side Effects:
 - Methods should avoid altering global state or performing unexpected actions.

Summary

- Methods are integral to defining the behaviour of classes and their instances in Python OOP.
- They provide a way to encapsulate functionality, making code more modular and reusable.
- Understanding the different types of methods allows you to structure your classes effectively, leveraging Python's features for clean and efficient code design.

1.9 Inheritance (KT0109) (IAC0101)

Inheritance is a fundamental principle that allows a class (called a child class or subclass) to derive attributes and methods from another class (called a parent class or superclass). Inheritance promotes code reusability, hierarchical classification, and the creation of complex systems by building upon existing code.

Key Concepts of Inheritance

1. Parent Class (Superclass):
 - The class whose attributes and methods are inherited.
 - Defines common properties and behaviours.
2. Child Class (Subclass):
 - The class that inherits from the parent class.
 - Can use, modify, or extend the attributes and methods of the parent class.
 - Can have additional attributes and methods specific to itself.

3. Single Inheritance:

- o A subclass inherits from one parent class.

4. Multiple Inheritance:

- o A subclass inherits from more than one parent class.
- o Python supports multiple inheritance, allowing a class to derive from multiple base classes.

Benefits of Inheritance

- Code Reusability:
 - o Common functionality is defined once in the parent class and reused in child classes.
- Hierarchical Classification:
 - o Allows the creation of a natural hierarchy, organising classes in a structured manner.
- Ease of Maintenance:
 - o Changes in the parent class can propagate to child classes, simplifying updates.
- Extensibility:
 - o Child classes can extend or customize the behaviour of parent classes without modifying them.

Syntax of Inheritance in Python

Defining a Parent Class:

```
class ParentClass:  
    # Parent class attributes and methods  
    pass
```

Defining a Child Class (Single Inheritance):

```
class ChildClass(ParentClass):  
    # Child class attributes and methods  
    pass
```

Defining a Child Class (Multiple Inheritance):

```
class ChildClass(ParentClass1, ParentClass2):  
    # Child class attributes and methods  
    pass
```

Example: Single Inheritance

Parent Class:

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

    def eat(self):
        print(f"{self.name} is eating.")
```

Child Class:

```
class Dog(Animal):
    def bark(self):
        print(f"{self.name} says woof!")
```

Usage:

```
dog = Dog("Buddy")

dog.eat()    # Inherited from Animal class
dog.bark()   # Defined in Dog class

# Output:
# Buddy is eating.
# Buddy says woof!
```

Explanation:

- Inheritance: The Dog class inherits from the Animal class.
- Inherited Method: Dog instances can use the eat method defined in Animal.

- Child Class Method: Dog defines its own method bark.

Overriding Methods

A child class can override methods from the parent class to provide a different implementation.

Example:

```
class Animal:  
    def sound(self):  
        print("The animal makes a sound.")  
  
class Cat(Animal):  
    def sound(self):  
        print("The cat meows.")
```

Usage:

```
animal = Animal()  
animal.sound() # Output: The animal makes a sound.  
  
cat = Cat()  
cat.sound() # Output: The cat meows.
```

Explanation:

- Method Overriding: The Cat class overrides the sound method of the Animal class.
- When sound is called on a Cat instance, the overridden method in Cat is executed.

Using super() Function

The super() function allows you to call methods from the parent class within a child class.

Example:

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

class Car(Vehicle):
    def __init__(self, make, model, doors):
        super().__init__(make, model) # Call parent constructor
        self.doors = doors

    def info(self):
        print(f"{self.make} {self.model} with {self.doors} doors.")
```

Usage:

```
car = Car("Toyota", "Corolla", 4)
car.info() # Output: Toyota Corolla with 4 doors.
```

Explanation:

- Constructor Inheritance: The Car class calls the __init__ method of Vehicle using super().__init__(make, model).
- Additional Attributes: Car adds the doors attribute.

Multiple Inheritance

Python allows a class to inherit from multiple parent classes.

Example:

```
class Flyable:
```

```
def fly(self):
    print("This object can fly.")

class Swimmable:
    def swim(self):
        print("This object can swim.")

class Duck(Flyable, Swimmable):
    pass
```

Usage:

```
duck = Duck()
duck.fly()  # Output: This object can fly.
duck.swim() # Output: This object can swim.
```

Explanation:

- Multiple Inheritance: Duck inherits from both Flyable and Swimmable.
- Combined Behaviours: Duck instances have both fly and swim methods.

The Method Resolution Order (MRO)

In multiple inheritance, Python follows the Method Resolution Order (MRO) to determine the order in which base classes are searched when looking for a method.

- The MRO can be viewed using the `__mro__` attribute or the `mro()` method.

Example:

```
class A:
    def process(self):
        print("Process method from class A.")

class B(A):
    def process(self):
```

```
    print("Process method from class B.")

class C(A):
    pass

class D(B, C):
    pass

print(D.__mro__)
```

Output (Arduino):

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

Explanation:

- MRO Order: When D calls process, Python looks for the method in D, then B, then C, then A, following the MRO.
- Conflict Resolution: If multiple base classes have a method with the same name, the method in the first base class in the MRO is used.

Hierarchical Inheritance

A class can serve as a parent to multiple child classes.

Example:

```
class Animal:
    def move(self):
        print("The animal moves.")
```

```
class Fish(Animal):
    def move(self):
        print("The fish swims.")

class Bird(Animal):
    def move(self):
        print("The bird flies.")
```

Usage:

```
a = Animal()
f = Fish()
b = Bird()

a.move()  # Output: The animal moves.
f.move()  # Output: The fish swims.
b.move()  # Output: The bird flies.
```

Explanation:

- Hierarchical Structure: Both Fish and Bird inherit from Animal.
- Method Overriding: Each subclass provides its own implementation of move.

Advantages of Inheritance

1. Code Reusability:
 - Reduce duplication by reusing existing code.
2. Extensibility:
 - Extend and customize existing classes without modifying them.
3. Logical Representation:

- o Model real-world relationships using hierarchical structures.

4. Maintainability:

- o Easier to maintain and update code, as changes in the parent class can affect child classes.

Considerations and Best Practices

- Avoid Deep Inheritance Hierarchies:

- o Complex inheritance chains can make code hard to understand and maintain.

- Use Composition Over Inheritance:

- o Sometimes, using composition (objects containing other objects) is more appropriate than inheritance.

- Understand the MRO:

- o In multiple inheritance scenarios, be aware of the MRO to avoid unexpected behaviour.

- Override Methods Carefully:

- o When overriding methods, consider using `super()` to maintain the behaviour of the parent class.

Summary

- Inheritance allows classes to inherit attributes and methods from other classes, promoting code reuse and logical structuring.
- Single Inheritance involves one parent class; Multiple Inheritance involves multiple parent classes.
- Overriding enables child classes to provide specific implementations of methods defined in parent classes.
- The `super()` Function is used to call methods from the parent class.
- Method Resolution Order (MRO) determines the order in which base classes are searched in multiple inheritance.

1.10 Multiple inheritance (KT0110) (IAC0101)

Multiple inheritance is a feature that allows a class (known as a child class or subclass) to inherit attributes and methods from more than one parent class (also called base classes or superclasses). This enables the subclass to combine and extend the functionalities of all its parent classes.

Key Concepts of Multiple Inheritance

1. Inheritance from Multiple Parents:

- o A subclass derives from more than one parent class.
- o The subclass gains all the attributes and methods of its parent classes.

2. Method Resolution Order (MRO):

- o Python uses a specific order to determine which method or attribute to use when multiple parent classes have methods or attributes with the same name.
- o The MRO defines the order in which base classes are searched when looking for a method or attribute.

3. super() Function:

- o The super() function can be used to call methods from the parent classes.
- o In multiple inheritance, super() follows the MRO to determine which parent method to call.

Syntax of Multiple Inheritance in Python

```
class ChildClass(ParentClass1, ParentClass2, ...):  
    # Child class body  
    pass
```

Example:

```

class ParentClass1:
    def method1(self):
        print("Method from ParentClass1")

class ParentClass2:
    def method2(self):
        print("Method from ParentClass2")

class ChildClass(ParentClass1, ParentClass2):
    pass

# Creating an instance of ChildClass
child = ChildClass()
child.method1() # Output: Method from ParentClass1
child.method2() # Output: Method from ParentClass2

```

Explanation:

- ChildClass inherits from both ParentClass1 and ParentClass2.
- An instance of ChildClass can access methods from both parent classes.

Benefits of Multiple Inheritance

1. Reusability:
 - o Combine functionalities of multiple classes without rewriting code.
2. Flexibility:
 - o Create classes that reflect complex relationships and behaviours.
3. Modularity:
 - o Separate concerns by defining distinct parent classes and combining them as needed.

Potential Issues with Multiple Inheritance

1. Name Conflicts:

- o If multiple parent classes have methods or attributes with the same name, it can lead to ambiguity.
- o Python resolves this using the MRO.

2. Diamond Problem:

- o Occurs when a subclass inherits from two classes that inherit from the same superclass.
- o Can lead to multiple copies of the superclass's attributes and methods.
- o Python's MRO and the C3 linearization algorithm help to resolve this issue.

Method Resolution Order (MRO)

- Definition:
 - o The MRO determines the order in which Python looks for a method or attribute in a hierarchy of classes.
- Viewing the MRO:
 - o You can view a class's MRO using the `__mro__` attribute or the `mro()` method.

Example:

```
class A:  
    def method(self):  
        print("Method from A")  
  
class B(A):  
    def method(self):  
        print("Method from B")  
  
class C(A):  
    def method(self):  
        print("Method from C")  
  
class D(B, C):
```

```
pass

d = D()
d.method() # Output: Method from B
print(D.__mro__)
```

Output:

```
Method from B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

Explanation:

- The MRO for class D is [D, B, C, A, object].
- When d.method() is called, Python looks for method in D, then B, finds it in B, and stops searching.

Using super() with Multiple Inheritance

- The super() function follows the MRO.
- It allows you to call the next method in the MRO chain.

Example:

```
class A:
    def method(self):
        print("A method")

class B(A):
    def method(self):
        print("B method")
        super().method()
```

```
class C(A):
    def method(self):
        print("C method")
        super().method()

class D(B, C):
    def method(self):
        print("D method")
        super().method()

d = D()
d.method()
```

Output (sql):

```
D method
B method
C method
A method
```

Explanation:

- `super().method()` in each class calls the next method according to the MRO.
- The methods are called in the order: D, B, C, A.

Best Practices with Multiple Inheritance

1. Be Cautious with Name Conflicts:
 - Ensure methods and attributes are uniquely named when possible.
 - Be aware of which parent class's method is being called.
2. Understand the MRO:
 - Knowing the MRO helps predict which method will be executed.
 - Use `ClassName.__mro__` to inspect the method resolution order.

3. Use Composition Over Inheritance When Appropriate:

- o Sometimes it's better to compose classes (objects containing other objects) rather than inherit from multiple classes.

Summary

- Multiple Inheritance allows a class to inherit from more than one parent class, combining their attributes and methods.
- Method Resolution Order (MRO) defines the order in which Python looks for methods and attributes in parent classes.
- `super()` Function can be used to call parent class methods, following the MRO.
- While multiple inheritance offers flexibility, it can introduce complexity; understanding the MRO and careful design can mitigate potential issues.

1.11 Method overriding (KT0111) (IAC0101)

Method overriding is a feature that allows a subclass (child class) to provide a specific implementation of a method that is already defined in its superclass (parent class). The overridden method in the subclass has the same name and parameters as the method in the parent class but implements different functionality.

Key Concepts of Method Overriding

1. Inheritance Relationship:

- o Method overriding occurs within an inheritance hierarchy where a child class inherits from a parent class.
- o The child class inherits attributes and methods from the parent class.

2. Same Method Signature:

- o The method in the child class must have the same name and parameters as in the parent class.
- o This ensures that the method can be called in the same way on both parent and child class instances.

3. Polymorphism:

- Method overriding enables polymorphism, allowing a single interface to represent different underlying forms (data types).
- It allows you to write code that works with the superclass type but executes the overridden method in the subclass.

Why Use Method Overriding

- Customization:
 - To tailor or extend the behaviour of inherited methods to meet specific requirements in the subclass.
- Consistency:
 - Maintains a consistent method interface across classes, making code more maintainable and understandable.
- Dynamic Method Resolution:
 - At runtime, the method corresponding to the actual object's class is called, allowing for flexible and dynamic behaviour.

Basic Syntax of Method Overriding

```
class ParentClass:
    def method_name(self, parameters):
        # Parent class method implementation
        pass

class ChildClass(ParentClass):
    def method_name(self, parameters):
        # Child class method implementation (overrides ParentClass method)
        pass
```

- ParentClass: Defines the original method.
- ChildClass: Inherits from ParentClass and overrides method_name.

Example of Method Overriding

Parent Class:



66

```
class Animal:  
    def speak(self):  
        print("The animal makes a sound.")
```

Child Classes:

```
class Dog(Animal):  
    def speak(self):  
        print("The dog barks.")  
  
class Cat(Animal):  
    def speak(self):  
        print("The cat meows.")
```

Usage:

```
animal = Animal()  
dog = Dog()  
cat = Cat()  
  
animal.speak() # Output: The animal makes a sound.  
dog.speak() # Output: The dog barks.  
cat.speak() # Output: The cat meows.
```

Explanation:

- Inheritance: Dog and Cat inherit from Animal.
- Overriding: Both Dog and Cat override the speak method to provide specific behaviour.
- Polymorphism: The speak method behaves differently depending on the object's class.

Using the super() Function

You can use the `super()` function to call the parent class's method within the overridden method in the child class. This is useful when you want to extend rather than completely replace the parent class's method.

Example:

```
class Animal:
    def speak(self):
        print("The animal makes a sound.")

class Dog(Animal):
    def speak(self):
        super().speak() # Calls the parent class's speak method
        print("The dog barks.")
```

Usage:

```
dog = Dog()
dog.speak()
```

Output (css):

```
The animal makes a sound.
The dog barks.
```

Explanation:

- `super().speak()`: Calls the `speak` method of the `Animal` class.
- Extended Behaviour: The `Dog` class adds additional behaviour to the `speak` method.

Method Overriding and Polymorphism

Method overriding enables polymorphic behaviour, allowing you to write code that works with superclass references but executes subclass methods.

Example:

```
def make_animal_speak(animal):
    animal.speak()

animals = [Animal(), Dog(), Cat()]

for animal in animals:
    make_animal_speak(animal)
```

Output:

```
The animal makes a sound.
The dog barks.
The cat meows.
```

Explanation:

- Polymorphic Function: `make_animal_speak` accepts any object that has a `speak` method.
- Dynamic Binding: The correct `speak` method is called based on the object's actual class at runtime.

Best Practices

1. Maintain Method Signature Consistency:
 - The overridden method should have the same name and parameters as the method in the parent class.
2. Use `super()` Appropriately:
 - When extending the behaviour of the parent method, use `super()` to call the parent class's method.
3. Follow the Liskov Substitution Principle:
 - Objects of the subclass should be substitutable for objects of the superclass without affecting the correctness of the program.

4. Document Overridden Methods:

- o Provide clear documentation to explain why the method was overridden and how the new behaviour differs.

Summary

- Method Overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.
- It enables polymorphism, allowing methods to behave differently based on the object's actual class.
- Overriding is achieved by defining a method in the child class with the same name and parameters as in the parent class.
- Use the super() function to extend the parent class's method when needed.

1.12 Operator overloading (KT0112) (IAC0101)

Operator overloading is a feature that allows developers to redefine the behaviour of built-in operators (like +, -, *, ==, etc.) for user-defined classes. By overloading operators, you can specify how objects of your class interact with these operators, enabling intuitive and expressive code that mirrors natural language or mathematical expressions.

Key Concepts of Operator Overloading

1. Special Methods (Magic Methods):

- o Python provides special methods, also known as "dunder methods" (double underscore methods), that allow you to define the behaviour of operators for your classes.
- o These methods have names that begin and end with double underscores, such as `__add__`, `__sub__`, `__eq__`, etc.

2. Customization of Built-in Operations:

- o By implementing these special methods in your class, you can customize how instances of your class respond to operators.

3. Enhancing Code Readability:

- o Operator overloading enables writing code that is more intuitive and closer to the problem domain, improving readability and maintainability.

How Operator Overloading Works in Python

- Implementing Special Methods:

- o Define methods in your class that correspond to the operators you want to overload.
- o The method names are predefined and correspond to specific operators.

- Common Special Methods for Operators:

Operator	Expression	Special Method
Addition	$a + b$	<code>a.__add__(b)</code>
Subtraction	$a - b$	<code>a.__sub__(b)</code>
Multiplication	$a * b$	<code>a.__mul__(b)</code>
Division	a / b	<code>a.__truediv__(b)</code>

Operator	Expression	Special Method
Equality	a == b	a.__eq__(b)
Less Than	a < b	a.__lt__(b)
Greater Than	a > b	a.__gt__(b)
String Representation	str(a)	a.__str__()
Representation	repr(a)	a.__repr__()

Example: Overloading the Addition Operator

Creating a Vector Class:

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

    # Overloading the + operator
    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Operand must be of type Vector")

    # String representation for printing
    def __str__(self):
        return f"Vector({self.x}, {self.y})"
```

Usage:

```
v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2 # Uses the __add__ method

print(v3) # Output: Vector(7, 10)
```

Explanation:

- __add__ Method:
 - Defines how the + operator works with Vector instances.

- Checks if the other operand is also a Vector.
- Returns a new Vector instance with the sum of the coordinates.
- `__str__` Method:
 - Provides a human-readable string representation for printing.

Example: Overloading Comparison Operators

Creating a Box Class:

```
class Box:
    def __init__(self, volume):
        self.volume = volume

    # Overloading the < operator
    def __lt__(self, other):
        return self.volume < other.volume

    # Overloading the == operator
    def __eq__(self, other):
        return self.volume == other.volume
```

Usage:

```
box1 = Box(100)
box2 = Box(150)

print(box1 < box2)    # Output: True
print(box1 == box2)   # Output: False
```

Explanation:

- `__lt__` Method:
 - Defines behaviour for the `<` operator.
 - Compares the volumes of two Box instances.

- `__eq__` Method:
 - Defines behaviour for the `==` operator.
 - Checks if the volumes are equal.

Example: Overloading the String Representation Operators

Creating a Person Class:

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

    # Overloading str()
    def __str__(self):
        return f"Person named {self.name}"

    # Overloading repr()
    def __repr__(self):
        return f"Person('{self.name}')"
```

Usage:

```
person = Person("Alice")
print(str(person))  # Output: Person named Alice
print(repr(person)) # Output: Person('Alice')
```

Explanation:

- `__str__` Method:
 - Returns a readable, informal string representation.
 - Used by the `str()` function and `print()`.
- `__repr__` Method:
 - Returns an official string representation.
 - Should be unambiguous and, if possible, match the code needed to recreate the object.

- o Used by the `repr()` function and in interactive prompts.

Benefits of Operator Overloading

1. Enhanced Readability:
 - o Code that uses natural operators is often easier to read and understand.
2. Intuitive Interface:
 - o Objects can be manipulated using standard operators, making them behave like built-in types.
3. Polymorphism:
 - o Enables objects of different classes to be used interchangeably with operators if they implement the same special methods.
4. Maintainability:
 - o Reduces the need for verbose method calls, leading to cleaner code.

Best Practices for Operator Overloading

1. Follow Expected Behaviour:
 - o Overloaded operators should behave in a way that is consistent with their typical use to avoid confusion.
 - o For example, `+` should perform an addition-like operation.
2. Type Checking:
 - o Ensure that operands are of compatible types.
 - o Use `isinstance()` to check operand types and handle errors appropriately.
3. Immutable vs. Mutable Objects:
 - o Operators like `+` should return new instances for immutable objects.
 - o Operators like `+=` (in-place addition, `__iadd__`) can modify the object if it is mutable.
4. Provide All Relevant Operators:
 - o When overloading comparison operators, implement all necessary methods (`__eq__`, `__lt__`, `__le__`, `__gt__`, `__ge__`, `__ne__`) for full functionality.

5. Avoid Overcomplicating:

- Only overload operators when it makes logical sense and improves code clarity.

Special Methods for Common Operators

- Arithmetic Operators:

Operator	Method	In-Place Method
+	__add__	__iadd__
-	__sub__	__isub__
*	__mul__	__imul__
/	__truediv__	__itruediv__
//	__floordiv__	__ifloordiv__
%	__mod__	__imod__
**	__pow__	__ipow__

- Comparison Operators:

Operator	Method
==	__eq__
!=	__ne__
<	__lt__
<=	__le__
>	__gt__
>=	__ge__

- Unary Operators:

Operator	Method
-	__neg__
+	__pos__
~	__invert__

Example: Overloading the In-Place Addition Operator

```
Creating a ShoppingCart Class:  
class ShoppingCart:  
    def __init__(self):  
        self.items = []  
  
    # Overloading the += operator  
    def __iadd__(self, item):  
        self.items.append(item)  
        return self  
  
    # String representation  
    def __str__(self):  
        return f"ShoppingCart({self.items})"
```

Usage:

```
cart = ShoppingCart()  
cart += "Apple"  
cart += "Banana"  
  
print(cart) # Output: ShoppingCart(['Apple', 'Banana'])
```

Explanation:

- `__iadd__` Method:
 - Defines behaviour for the `+=` operator.
 - Modifies the object in place by adding an item to the list.

Summary

- Operator Overloading allows custom classes to interact with Python's built-in operators by implementing special methods.
- By overloading operators, you can create classes that behave like built-in types, making your code more intuitive and expressive.

- Special Methods correspond to specific operators and need to be implemented according to the expected behaviour of those operators.
- Best Practices include ensuring that overloaded operators behave logically, performing type checks, and avoiding unnecessary complexity.



Formative Assessment Activity [1]

Python Object-Oriented Programming (OOP)

Complete the formative activity in your **KM4 Learner Workbook**,

Knowledge Topic KM-04-KT02:

Topic Code	KM-04-KT02
Topic	Fundamental concepts of Object-oriented programming
Weight	20%

This knowledge topic will cover the following topic elements:

- KT0201 Concept, definition, and functions
- KT0202 Class
- KT0203 Object
- KT0204 Method
- KT0205 Inheritance
- KT0206 Encapsulation
- KT0207 Polymorphism
- KT0208 Data abstraction

After working through this knowledge topic, your competence will be assessed according to the following criterion/criteria:

- IAC0201 The use of fundamental concepts of OOP is stated.

KM-04-KT02 Fundamental Concepts of Object-Oriented Programming (IAC0201)

2.1 Concept, definition and functions (KT0201) (IAC0201)

Object-Oriented Programming (OOP) is a programming paradigm that structures software design around objects rather than actions, and data rather than logic. It enables programmers to create modules that can be reused in other programs, enhancing code reusability and scalability.

Fundamental Concepts of OOP:

1. Classes and Objects

- o Concept:
 - Class: A blueprint or template for creating objects. It defines a set of attributes (properties) and methods (behaviours) that the objects created from the class will have.
 - Object: An instance of a class. It represents a specific implementation of the class with actual values.
- o Functions:
 - Encapsulation of Data and Behaviour: Classes encapsulate data and methods into a single unit, promoting modularity.
 - Instantiation: Objects are created (instantiated) from classes, allowing multiple instances with different data.

2. Encapsulation

- o Concept:
 - The bundling of data (attributes) and methods that manipulate that data within one unit, typically a class.
 - Restricts direct access to some of an object's components, which is a means of preventing unauthorized or accidental interference.
- o Functions:
 - Data Hiding: Protects the internal state of an object from external modification.

- Controlled Access: Provides public methods (getters and setters) to access or modify private attributes.

3. Inheritance

- Concept:
 - A mechanism where a new class (child or subclass) derives properties and behaviours from an existing class (parent or superclass).
 - Establishes an "is-a" relationship between classes.
- Functions:
 - Code Reusability: Allows child classes to inherit and reuse code from parent classes.
 - Hierarchical Classification: Organises classes into a hierarchy, reflecting real-world relationships.

4. Polymorphism

- Concept:
 - The ability of different classes to be treated as instances of the same class through inheritance.
 - Enables a single interface to represent different underlying data types.
- Functions:
 - Method Overriding: Allows child classes to provide specific implementations of methods that are already defined in their parent class.
 - Dynamic Binding: The method to be invoked is determined at runtime based on the object's actual type.

5. Abstraction

- Concept:
 - The process of hiding the complex reality while exposing only the necessary parts.

- Focuses on what an object does rather than how it does it.
- Functions:
 - Simplification: Reduces programming complexity and effort by providing a clear model of a complex system.
 - Interface Implementation: Defines abstract classes and interfaces to outline methods that must be created within any child classes.

These fundamental concepts work together to form the backbone of object-oriented programming:

- Classes and Objects provide the basic structure.
- Encapsulation ensures data integrity.
- Inheritance promotes code reusability.
- Polymorphism offers flexibility and integration.
- Abstraction simplifies complex systems.

By leveraging these principles, OOP facilitates the creation of modular, scalable, and maintainable software, closely modelling real-world entities and relationships.

2.2 Class (KT0202) (IAC0201)

A class is one of the fundamental concepts and serves as a blueprint or template for creating objects. It defines a set of attributes (data) and methods (functions) that encapsulate the properties and behaviours common to all objects of that type.

Key Aspects of a Class:

1. Blueprint for Objects:
 - A class specifies the structure and capabilities that its objects (instances) will have.

- It outlines what attributes (also called properties or fields) the objects will store and what methods (also known as functions or procedures) they can perform.

2. Encapsulation of Data and Behaviour:

- Classes encapsulate data and the functions that operate on that data into a single unit.
- This encapsulation promotes modularity and information hiding, ensuring that the internal representation of an object is hidden from the outside.

3. Attributes and Methods:

- Attributes: Variables that hold the state or characteristics of an object (e.g., colour, size, name).
- Methods: Functions that define the behaviours or actions that an object can perform (e.g., move, speak, calculate).

4. Instantiation:

- Objects are created from classes through a process called instantiation.
- Each object (also known as an instance) has its own set of attribute values but shares the structure and behaviour defined by the class.

Functions and Purpose of Classes in OOP:

- Modelling Real-World Entities:
 - Classes allow programmers to model complex real-world entities by encapsulating related data and behaviours.
- Code Reusability:
 - By defining a class once, you can create multiple objects from it without rewriting code, promoting reuse and reducing redundancy.
- Organisation and Modularity:
 - Classes help organise code into logical sections, making programs more manageable and scalable.
- Foundation for OOP Principles:
 - Classes are essential for implementing other OOP concepts like inheritance (classes inheriting properties from other classes), polymorphism (classes behaving differently based on context), and

abstraction (hiding complex realities while exposing only the necessary parts).

Analogy:

Think of a class as a blueprint or template for building a house. The blueprint contains all the specifications (number of rooms, layout, materials) needed to construct a house. However, the blueprint itself is not a house; it's just a plan. When you use the blueprint to build actual houses (objects), each house can have different paint colours or interior designs (attribute values), but they all share the same fundamental structure defined by the blueprint (class).

Summary:

- A class in OOP is a fundamental concept that defines the structure (attributes) and behaviour (methods) of objects.
- It acts as a blueprint for creating objects, encapsulating data and functions into a single unit.
- Classes promote code reusability, modularity, and provide a foundation for other OOP principles like inheritance, polymorphism, and abstraction.
- Understanding classes is crucial for designing and building organised, efficient, and scalable object-oriented programs.

2.3 Object (KT0203) (IAC0201)

An object is a fundamental unit that represents a real-world entity or an abstract concept within a software program. An object is an instance of a class, embodying the attributes (data) and methods (functions) defined by its class. Objects are the building blocks of OOP, encapsulating data and behaviour into a single, manageable, and reusable unit.

Key Aspects of an Object:

1. Instance of a Class:

- An object is created from a class through a process called instantiation.
- While the class serves as a blueprint, the object is the actual entity with concrete attribute values.

2. Attributes (State):

- Objects store data in variables known as attributes, properties, or fields.
- These attributes represent the object's current state or characteristics.
- For example, a Car object might have attributes like colour, make, model, and speed.

3. Methods (Behaviour):

- Objects have methods, which are functions defined within the class.
- Methods define the behaviours or actions that the object can perform.
- For example, a Car object might have methods like accelerate(), brake(), and turn().

4. Encapsulation of Data and Behaviour:

- Objects encapsulate their attributes and methods, bundling data and functionality together.
- This encapsulation promotes modularity and protects the object's internal state from external interference.

5. Identity:

- Each object has a unique identity that distinguishes it from other objects, even if they have the same attributes.
- In programming languages, this identity is often represented by the object's memory address.

Functions and Purpose of Objects in OOP:

- Modelling Real-World Entities:
 - Objects allow programmers to create representations of real-world entities or abstract concepts within the code.
 - This modelling makes programs more intuitive and aligns software structures with real-world structures.
- Interaction and Collaboration:
 - Objects can interact with one another by calling each other's methods and accessing attributes.

- This interaction enables complex behaviours and relationships, reflecting real-world interactions.
- State Management:
 - Objects maintain their own state through their attributes, which can change over time.
 - Methods can modify the object's state, allowing it to evolve during program execution.
- Reusability and Modularity:
 - Objects promote code reusability by encapsulating functionality that can be used in different parts of a program.
 - They enhance modularity, making it easier to manage and maintain large codebases.
- Foundation for OOP Principles:
 - Objects are central to other OOP concepts like inheritance (objects can inherit properties from parent classes), polymorphism (objects can be treated as instances of their parent class), and abstraction (objects expose only necessary features).

Analogy:

Think of an object as a specific house built from architectural plans:

- Class (Blueprint): The architectural plans for a house, detailing the design, materials, and specifications.
- Object (Actual House): The completed house built from those plans. It has specific features like colour, number of rooms, and furnishings.
- Each house (object) built from the same blueprint (class) can have different paint colours, interior designs, and furniture, but they share the same fundamental structure.

Summary:

- An object in OOP is a concrete instance of a class, encapsulating both data (attributes) and behaviour (methods).

- Objects represent real-world entities or abstract concepts within a program, making code more intuitive and aligned with real-life scenarios.
- They interact with other objects, maintain their own state, and can change over time through method calls.
- Understanding objects is essential for leveraging the power of OOP, as they are the primary means through which data and functionality are organised and manipulated within object-oriented programs.

2.4 Method (KT0204) (IAC0201)

A method is a function defined within a class that describes the behaviours or actions that an object created from the class can perform. Methods operate on the data contained within the object (its attributes) and can modify the object's state or perform computations relevant to the object's role.

Key Aspects of Methods:

1. Association with Classes and Objects:
 - o Methods are integral parts of a class and are invoked by the objects (instances) of that class.
 - o They provide a way for objects to interact with their own data and with other objects.
2. Encapsulation of Behaviour:
 - o Methods encapsulate the functionality that is specific to the class.
 - o They define how an object behaves and responds to messages or function calls.
3. Interaction with Attributes:
 - o Methods can access and modify the object's attributes (its internal data).
 - o This allows the object to change its state or produce outputs based on its current data.
4. Types of Methods:
 - o Instance Methods:
 - Operate on individual instances of a class.
 - Use the self-parameter to access or modify the object's attributes.
 - o Class Methods:
 - Use the @classmethod decorator and the cls parameter to access or modify class-level data shared among all instances.
 - o Static Methods:
 - Use the @staticmethod decorator.

- Do not access or modify class or instance data; they are utility functions within the class namespace.

5. Method Overriding and Polymorphism:

- Subclasses can override methods from a parent class to provide specific implementations.
- This allows for polymorphic behaviour, where the same method call can have different effects depending on the object's class.

Functions and Purpose of Methods in OOP:

- Defining Object Behaviour:
 - Methods specify the actions that an object can perform, defining its behaviour within the program.
 - They enable objects to perform tasks, respond to events, or interact with other objects.
- Encapsulation and Data Protection:
 - Methods provide controlled access to an object's attributes.
 - They promote data integrity by restricting direct access to the object's internal state.
- Code Reusability and Modularity:
 - Methods allow for the reuse of code within different parts of a program or in different programs.
 - They help organise code into logical, manageable sections, enhancing readability and maintainability.
- Inter-Object Communication:
 - Methods enable objects to communicate and collaborate by calling each other's methods.
 - This interaction facilitates complex operations and relationships within the program.

Example:

Defining a Circle Class with Methods:

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius # Attribute representing the circle's
radius

    def area(self):
        return math.pi * self.radius ** 2 # Method calculating the
area

    def circumference(self):
        return 2 * math.pi * self.radius # Method calculating the
circumference

    def scale(self, factor):
        self.radius *= factor # Method modifying the object's state
```

Using the Circle Class:

```
# Creating an instance of Circle
circle = Circle(5)

# Calling methods on the object
print(f"Area: {circle.area():.2f}") # Output: Area: 78.54
print(f"Circumference: {circle.circumference():.2f}") # Output:
Circumference: 31.42

# Modifying the object's state using a method
circle.scale(2)
print(f"New radius: {circle.radius}") # Output: New radius:
10
print(f"New area: {circle.area():.2f}") # Output: New area:
314.16
```

Explanation:

- Attributes:
 - radius: Represents the radius of the circle.
- Methods:
 - area(): Calculates and returns the area of the circle.
 - circumference(): Calculates and returns the circumference.
 - scale(factor): Scales the radius by a given factor, modifying the object's state.

Analogy:

Think of methods as the actions or capabilities of an object, like the functions a device might have. For example, a smartphone (object) has functions like making calls, sending messages, and taking photos (methods). These methods operate using the data stored within the smartphone (attributes like contacts, messages, and settings).

Summary:

- A method in OOP is a function associated with a class that defines the behaviors or actions of its objects.
- Methods enable objects to interact with their own data and with other objects, encapsulating functionality within a class.
- They are essential for defining how objects behave and respond within a program, allowing for dynamic and interactive software.
- Understanding methods is crucial for leveraging the power of OOP, as they bring objects to life by providing them with behaviours that mirror real-world actions.

2.5 Inheritance (KT0205) (IAC0201)

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP). It allows a new class, known as a child class or subclass, to acquire the properties and behaviours (attributes and methods) of an existing class, called a parent class or superclass. Inheritance promotes code reusability, hierarchical organisation, and enhances the ability to create complex systems by building upon existing code.

Key Aspects of Inheritance:

1. Code Reusability:

- Reuse Existing Code: Inheritance enables child classes to inherit and use the code defined in parent classes without rewriting it.
- Extend Functionality: Child classes can introduce additional attributes and methods, extending the capabilities of the parent class.

2. Hierarchical Classification:

- Organise Classes: Inheritance allows for the creation of a class hierarchy that mirrors real-world relationships.
- "Is-a" Relationship: Establishes that the child class is a type of the parent class (e.g., a Car is a type of Vehicle).

3. Method Overriding:

- Customize Behaviour: Child classes can override methods from the parent class to provide specific implementations.
- Polymorphism: Allows objects to be treated as instances of their parent class but behave differently based on their actual class.

4. Types of Inheritance:

- Single Inheritance: A child class inherits from one parent class.
- Multiple Inheritance: A child class inherits from more than one parent class (supported in languages like Python).
- Multilevel Inheritance: A class inherits from a child class, forming a chain of inheritance.
- Hierarchical Inheritance: Multiple child classes inherit from a single parent class.

Functions and Purpose of Inheritance in OOP:

- Promotes Code Reusability:
 - Reduces redundancy by sharing common code across multiple classes.
 - Simplifies maintenance since changes in the parent class propagate to child classes.
- Facilitates Polymorphism:
 - Allows for writing generic code that works with a superclass type but executes subclass-specific behaviour.
 - Enhances flexibility and integration within the codebase.
- Reflects Real-World Relationships:
 - Models natural hierarchies and relationships between entities.
 - Makes the code more intuitive and easier to understand.
- Enables Extensibility:
 - New features can be added to the system by creating new subclasses without altering existing code.
 - Encourages the open/closed principle where classes are open for extension but closed for modification.

Example:

Parent Class (Vehicle):

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

    def info(self):
        print(f"This is a {self.make} {self.model}.")

    def start_engine(self):
        print("The engine has started.")
```

Child Class (Car) Inheriting from Vehicle:

```
class Car(Vehicle):
    def __init__(self, make, model, doors):
        super().__init__(make, model) # Call the parent class constructor
        self.doors = doors

    def info(self):
        super().info()
        print(f"It has {self.doors} doors.")

    def start_engine(self):
        print("The car's engine roars to life.")
```

Usage:

```
# Creating an instance of Car
my_car = Car("Toyota", "Corolla", 4)

# Calling methods
my_car.info()
```

```
my_car.start_engine()
```

Output (vbnet):

```
This is a Toyota Corolla.  
It has 4 doors.  
The car's engine roars to life.
```

Explanation:

- Inheritance of Attributes and Methods:
 - Car inherits make, model, and the `__init__` method from Vehicle.
- Method Overriding:
 - Car overrides the `start_engine` method to provide a specific implementation.
- Using `super()`:
 - The `super()` function is used to call the parent class's methods, allowing the child class to build upon them.

Analogy:

Imagine a family tree:

- Parent (Superclass): A parent has certain characteristics and behaviours.
- Child (Subclass): A child inherits these characteristics but can also have unique traits and behaviours.

In the same way, in programming:

- A parent class defines general attributes and methods.
- A child class inherits from the parent class and can introduce its own attributes and methods or modify existing ones.

Summary:

- Inheritance allows a class to inherit attributes and methods from another class, promoting code reusability and hierarchical structuring.
- It supports the creation of a natural class hierarchy, reflecting real-world relationships and enhancing code organisation.
- Inheritance facilitates polymorphism, enabling objects to be treated as instances of their parent class while exhibiting subclass-specific behaviour.
- By using inheritance, developers can create extensible and maintainable codebases, reducing redundancy and improving efficiency.

2.6 Encapsulation (KT0206) (IAC0201)

Encapsulation is a fundamental principle that combines data (attributes) and the methods (functions) that operate on that data into a single unit, typically a class. Encapsulation restricts direct access to some of an object's components, which is a means of preventing accidental or unauthorized interference and misuse of the data.

Key Aspects of Encapsulation:

1. Bundling of Data and Methods:

- o Encapsulation involves bundling the data (attributes) and the methods that manipulate that data within a single unit or class.
- o This bundling ensures that the object's internal state is hidden from the outside, exposing only what is necessary.

2. Access Control:

- o Encapsulation allows for the control of access to the components of a class.
- o It typically involves using access modifiers (like private, protected, and public) to restrict access to the internal state of the object.
- o In languages like Python, although access modifiers are not enforced, naming conventions (like prefixing attribute names with underscores) indicate intended privacy.

3. Data Hiding:

- o Encapsulation promotes data hiding, which means that the internal representation of an object is hidden from the outside.
- o Users of an object cannot see the inner workings but can interact with the object through a public interface.

4. Interface vs. Implementation:

- o Encapsulation separates the interface of a class (what operations can be performed) from its implementation (how these operations are performed).
- o This allows for changes in the implementation without affecting external code that uses the class.

Functions and Purpose of Encapsulation in OOP:

- Protection of Data Integrity:
 - By restricting access to the object's internal state, encapsulation prevents external code from making unintended or harmful modifications.
 - It ensures that data can only be changed in controlled ways through defined methods.
- Simplification and Modularity:
 - Encapsulation simplifies the interaction with objects by exposing a clear and concise interface.
 - It promotes modularity, making it easier to understand, maintain, and update code.
- Ease of Maintenance and Flexibility:
 - Changes to the internal implementation of a class can be made without affecting external code that relies on the class.
 - This makes it easier to update and maintain code over time.
- Improved Security:
 - Encapsulation helps in securing sensitive data by preventing unauthorized access and manipulation.
- Facilitation of Abstraction:
 - Encapsulation supports the concept of abstraction by hiding complex implementation details and exposing only necessary features.

Example:

Defining a BankAccount Class with Encapsulation:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number # Private attribute
        self.__balance = balance # Private attribute

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

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

    def get_balance(self):
        return self.__balance

    def get_account_number(self):
        return self.__account_number
```

Usage:

```
account = BankAccount("123456789", 1000)

account.deposit(500)          # Deposited $500. New balance: $1500.
account.withdraw(200)         # Withdrew $200. New balance: $1300.

print(f"Account Number: {account.get_account_number()}") # Account Number:
123456789
print(f"Current Balance: ${account.get_balance()}")       # Current Balance:
$1300
```

Explanation:

- Private Attributes:
 - Attributes `__account_number` and `__balance` are prefixed with double underscores, indicating that they are intended to be private.
- Public Methods:
 - Methods `deposit()`, `withdraw()`, `get_balance()`, and `get_account_number()` provide controlled access to the private attributes.
- Data Protection:
 - External code cannot directly access or modify the private attributes, ensuring data integrity.

Analogy:

Think of encapsulation like a capsule (medicine pill) that encapsulates the drug inside a protective outer layer. The capsule controls how the drug is released into the body, protecting it from being destroyed by stomach acids and ensuring it reaches the right place. Similarly, encapsulation in OOP protects the internal data of an object and controls how it is accessed and modified.

Summary:

- Encapsulation is a core principle of OOP that combines data and the methods that manipulate that data into a single unit, typically a class.
- It restricts direct access to an object's internal state, promoting data integrity and security.
- Encapsulation provides a clear interface for interacting with objects while hiding the complex implementation details.
- By promoting modularity, ease of maintenance, and flexibility, encapsulation contributes to the creation of robust, scalable, and maintainable software.

2.7 Polymorphism (KT0207) (IAC0201)

Polymorphism is a fundamental principle that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying data types, and the ability of different classes to respond to the same method call in their own unique way. The term polymorphism comes from the Greek words "poly" (many) and "morph" (form), meaning "many forms."

Key Aspects of Polymorphism:

1. Method Overriding (Runtime Polymorphism):

o Definition:

- Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
- The method in the subclass has the same name, return type, and parameters as in the parent class.

o Function:

- Allows subclasses to customize or extend the behaviour of methods inherited from the parent class.
- At runtime, the method corresponding to the object's actual class is invoked, enabling dynamic method dispatch.

2. Method Overloading (Compile-Time Polymorphism):

o Definition:

- Involves having multiple methods in the same class with the same name but different parameters (number, type, or order).
- Note: In Python, method overloading is not natively supported as in some other languages but can be mimicked using default parameters or variable-length arguments.

o Function:

- Allows methods to perform similar functions with different types or numbers of inputs.

- The method to be called is determined at compile-time based on the method signature.

3. Polymorphic Variables:

- Definition:
 - A reference variable (like a pointer) that can refer to objects of different classes at different times during execution.
 - The variable is declared as the type of the superclass and can hold any of its subclass objects.
- Function:
 - Enables writing code that is more general and flexible.
 - Allows for dynamic binding, where the method to be executed is determined at runtime.

4. Interfaces and Abstract Classes:

- Definition:
 - Interfaces and abstract classes define methods that must be implemented by subclasses.
 - They provide a way to specify methods that can be polymorphically implemented by different classes.
- Function:
 - Enforce a contract for subclasses to follow, ensuring consistent method signatures.
 - Allow for multiple classes to implement the same interface in different ways.

Functions and Purpose of Polymorphism in OOP:

- Flexibility and Extensibility:
 - Polymorphism allows programs to be designed with interchangeable components.
 - Enhances flexibility by enabling one interface to be used for a general class of actions.
- Code Reusability:

- Promotes the use of a single piece of code to work with objects of different classes.
- Reduces redundancy and simplifies maintenance.
- Ease of Maintenance and Scalability:
 - Makes it easier to add new classes and methods without changing existing code.
 - Improves scalability by allowing systems to grow and adapt over time.
- Dynamic Method Binding:
 - The method that gets executed is determined at runtime based on the object's actual class.
 - Supports late binding, which is essential for implementing dynamic and flexible systems.

Example:

Creating a Shape Hierarchy with Polymorphism

Base Class (Shape):

```
class Shape:
    def area(self):
        pass # Abstract method, to be overridden by subclasses
```

Subclass (Rectangle):

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

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

Subclass (Circle):

```
import math

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

    def area(self):
        return math.pi * self.radius ** 2
```

Usage with Polymorphism:

```
# List of shape objects
shapes = [
    Rectangle(5, 10),
    Circle(7),
    Rectangle(3, 4)
]

# Iterating through shapes and calculating areas
for shape in shapes:
    print(f"The area is: {shape.area():.2f}")
```

Output:

```
The area is: 50.00
The area is: 153.94
The area is: 12.00
```

Explanation:

- Polymorphic Behaviour:
 - The area() method is called on different objects of type Shape.

- Each subclass (Rectangle, Circle) provides its own implementation of the area() method.
- The correct area() method is invoked based on the actual object's class at runtime.
- Single Interface:
 - The code that iterates through the shapes and calls area() does not need to know the specific type of shape.
 - This demonstrates the power of polymorphism in treating objects of different classes through a common interface.

Analogy:

Imagine a music player application that can play various audio file formats like MP3, WAV, and AAC.

- Player Interface:
 - The application uses a common interface play() to play audio files.
- Different Formats (Classes):
 - Each audio format (MP3, WAV, AAC) has its own way of decoding and playing audio data.
- Polymorphism in Action:
 - When play() is called on an audio file, the application doesn't need to know the specific format.
 - The correct decoding method is used based on the file type, allowing the application to handle multiple formats seamlessly.

Summary:

- Polymorphism is a core concept in OOP that allows objects of different classes to be treated as objects of a common superclass.
- It enables a single interface to represent different underlying forms, and methods to behave differently based on the object's actual class.
- Polymorphism enhances flexibility, code reusability, and scalability by allowing methods to interact with objects in a general way.

- Key mechanisms supporting polymorphism include method overriding, polymorphic variables, and the use of interfaces and abstract classes.

2.8 Data abstraction (KT0208) (IAC0201)

Data abstraction is a fundamental principle that involves the process of hiding the complex implementation details of a class and exposing only the essential features to the user. It focuses on the idea of showing what an object does instead of how it does it, allowing programmers to work with higher-level interfaces without needing to understand the underlying complexities.

Key Aspects of Data Abstraction:

1. Simplification of Complex Systems:

o Essential Features Exposure:

- Only the necessary attributes and methods are exposed to the outside world.
- Internal workings and implementation details are hidden.

o Reduction of Complexity:

- Users interact with a simplified interface.
- Complexity is managed within the class or module.

2. Encapsulation Relationship:

o Encapsulation as a Means to Achieve Abstraction:

- Encapsulation hides the internal state and functionality of an object.
- Abstraction uses encapsulation to present a simplified interface.

o Separation of Interface and Implementation:

- The interface defines what actions can be performed.
- The implementation defines how these actions are performed internally.

3. Use of Abstract Classes and Interfaces:

o Abstract Classes:

- Classes that cannot be instantiated on their own.
- Contain one or more abstract methods without implementation.

- Serve as templates for subclasses that implement the abstract methods.
- Interfaces (in languages that support them):
 - Define a contract of methods that implementing classes must provide.
 - Allow for a form of multiple inheritances.

4. Polymorphism Support:

- Uniform Interface Usage:
 - Different classes can be used interchangeably if they implement the same abstract methods.
 - Promotes code flexibility and reusability.

Functions and Purpose of Data Abstraction in OOP:

- Improved Code Manageability:
 - Modularity:
 - Code is organised into separate, manageable sections.
 - Changes in implementation do not affect external code if the interface remains the same.
 - Ease of Maintenance:
 - Simplifies updates and debugging by isolating changes to specific modules.
- Enhanced Security and Integrity:
 - Data Protection:
 - Internal data is protected from unauthorized access and modification.
 - Reduces the likelihood of unintended interactions or side effects.
- Simplified Interaction:
 - User-Friendly Interfaces:
 - Users interact with objects through simple interfaces without needing to understand complex implementations.

- Focus on High-Level Operations:
 - Programmers can focus on what tasks need to be performed rather than how they are executed.
- Facilitation of Polymorphism:
 - Interchangeable Components:
 - Objects of different classes can be used interchangeably if they share the same abstract base class or interface.
 - Dynamic Binding:
 - The method to be executed is determined at runtime based on the object's actual class.

Example: Implementing Data Abstraction with Abstract Base Classes

Using Python's abc Module:

```
from abc import ABC, abstractmethod

# Abstract Base Class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

# Subclass implementing the abstract methods
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Subclass implementing the abstract methods
class Circle(Shape):
```

```

def __init__(self, radius):
    self.radius = radius

def area(self):
    import math
    return math.pi * self.radius ** 2

def perimeter(self):
    import math
    return 2 * math.pi * self.radius

```

Usage:

```

shapes = [
    Rectangle(4, 6),
    Circle(5)
]

for shape in shapes:
    print(f"Area: {shape.area():.2f}, Perimeter: {shape.perimeter():.2f}")

```

Explanation:

- Abstract Base Class (Shape):
 - Defines abstract methods `area()` and `perimeter()` without implementation.
 - Cannot be instantiated directly due to the `@abstractmethod` decorators.
- Concrete Subclasses (Rectangle, Circle):
 - Provide specific implementations of the `area()` and `perimeter()` methods.
 - Can be instantiated and used polymorphically through the Shape interface.
- Data Abstraction in Action:
 - The user interacts with the shapes through the common interface provided by the Shape class.

- The complex calculations for area and perimeter are hidden within the subclasses.
- The user does not need to know the internal implementation details to use the shapes.

Analogy:

Consider driving a car:

- User Interface (Dashboard Controls):
 - You have access to the steering wheel, pedals, and gear shift.
 - These controls represent the abstract interface.
- Internal Mechanisms (Engine, Transmission):
 - The complex workings of the engine, fuel injection, and transmission are hidden.
 - You do not need to understand how these components work to drive the car.
- Data Abstraction in the Car:
 - The car manufacturers provide a simplified interface for drivers.
 - The complexity is encapsulated and abstracted away, allowing you to focus on operating the vehicle.

Summary:

- Data Abstraction is a fundamental concept in OOP that involves hiding the internal implementation details of a class and exposing only the essential features needed by the user.
- It promotes:
 - Simplification: By presenting a clear and simple interface.
 - Modularity: By encapsulating complex code within classes.
 - Security: By protecting internal data from external interference.

- Flexibility: By allowing different implementations to be used interchangeably through a common interface.
- Key Mechanisms:
 - Abstract Classes and Methods: Define interfaces without implementation, to be fulfilled by subclasses.
 - Encapsulation: Hides the internal state and functionality, exposing only what is necessary.
- By leveraging data abstraction, developers can create programs that are easier to understand, maintain, and extend, ultimately leading to more robust and scalable software systems.



Formative Assessment Activity [2]

Fundamental Concepts of Object-Oriented Programming

Complete the formative activity in your **KM4 Learner Workbook**.

Knowledge Topic KM-04-KT03:

Topic Code	KM-04-KT03
Topic	Basics of Algorithms
Weight	10%

This knowledge topic will cover the following topic elements:

- KT0301 Concept, definition, and functions
- KT0302 Dynamic programming algorithms
- KT0303 Searching and sorting algorithms
- KT0304 Mathematical algorithms
- KT0305 Data structures algorithms
- KT0306 Flow charts (on paper)

After working through this knowledge topic, your competence will be assessed according to the following criterion/criteria:

- IAC0301 Definitions, functions and features of Python algorithms are understood and explained

KM-04-KT03 Basics of Algorithms (IAC0301)

3.1 Concept, definition and functions (KT0301) (IAC0301)

An algorithm is a finite, well-defined sequence of instructions or computational steps designed to perform a specific task or solve a particular problem. It serves as a step-by-step procedure that takes an input, processes it through a series of operations, and produces an output.

Concept of Algorithms:

1. Finite Sequence of Steps:

- o An algorithm must have a clear starting point and a finite number of steps leading to an end point.
- o It should terminate after completing its defined instructions.

2. Well-Defined Instructions:

- o Each step in an algorithm must be precisely and unambiguously specified.
- o Instructions should be clear enough to be executed without interpretation or guesswork.

3. Input and Output:

- o Input: Algorithms accept zero or more inputs, which are the data to be processed.
- o Output: They produce one or more outputs as a result of the computation.

4. Deterministic or Non-Deterministic:

- o Deterministic Algorithms: Produce the same output every time for a given input.
- o Non-Deterministic Algorithms: May produce different outputs for the same input due to elements like randomness or parallelism.

5. Effectiveness:

- The steps should be basic enough to be carried out, in principle, by a human using paper and pencil.

6. Language Independence:

- Algorithms are conceptual and can be implemented in any programming language or expressed in pseudocode.

Functions and Purpose of Algorithms:

1. Problem Solving:

- Algorithms provide systematic methods for solving computational and mathematical problems efficiently and effectively.
- They are essential tools in computer science, mathematics, engineering, and related fields.

2. Automation of Tasks:

- By defining a clear set of instructions, algorithms enable the automation of tasks that would otherwise require manual effort.
- They are the foundation of all computer programs and software applications.

3. Efficiency Optimization:

- Algorithms are designed to optimize resource usage, such as time (speed of execution) and space (memory consumption).
- Efficient algorithms are crucial for handling large datasets and complex computations.

4. Data Processing and Analysis:

- They are used to process, manipulate, and analyse data, including sorting, searching, encryption, and compression.
- Algorithms transform raw data into meaningful information.

5. Decision Making:

- Algorithms can incorporate decision-making logic to handle various scenarios and inputs.
- They enable systems to make choices based on predefined criteria.

6. Foundation for Advanced Technologies:

- o Underpin technologies like artificial intelligence, machine learning, cryptography, and network security.
- o They enable innovations by solving problems that are computationally intensive or complex.

Examples of Basic Algorithms:

- Sorting Algorithms:
 - o Bubble Sort: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
 - o Quick Sort: Divides the list into smaller partitions and sorts them recursively.
- Searching Algorithms:
 - o Linear Search: Checks each element in the list sequentially until the desired element is found.
 - o Binary Search: Efficiently finds an element in a sorted list by repeatedly dividing the search interval in half.
- Mathematical Algorithms:
 - o Euclidean Algorithm: Computes the greatest common divisor (GCD) of two integers.
 - o Sieve of Eratosthenes: Finds all prime numbers up to a specified integer.

Algorithms are the fundamental building blocks of computer science and programming. They provide clear and efficient methods for solving problems and performing tasks. Understanding the basics of algorithms involves recognizing their structure, purpose, and how they can be applied to various domains to automate processes, optimize performance, and enable advanced technological solutions. Mastery of algorithms is essential for developing effective software and for innovation in technology and science.

3.2 Dynamic programming algorithms (KT0302) (IAC0301)

Dynamic Programming (DP) is an algorithmic technique used for solving complex problems by breaking them down into simpler subproblems. It is applicable when the problem can be divided into overlapping subproblems with optimal substructure. Dynamic programming stores the results of these subproblems to avoid redundant computations, thereby optimizing the overall performance.

Key Concepts of Dynamic Programming:

1. Optimal Substructure:

- o Definition:

- A problem has an optimal substructure if an optimal solution to the problem contains optimal solutions to its subproblems.

- o Function:

- Allows the problem to be solved recursively, building up solutions from smaller, optimal solutions.

2. Overlapping Subproblems:

- o Definition:

- The problem can be broken down into subproblems that are reused multiple times.

- o Function:

- Identifies that solving and storing the results of subproblems can reduce computation time by avoiding redundant calculations.

3. Memoization (Top-Down Approach):

- o Definition:

- An optimization technique where the results of expensive function calls are cached and returned when the same inputs occur again.

- o Function:

- Enhances recursive algorithms by storing the results of subproblems to avoid recomputation.

4. Tabulation (Bottom-Up Approach):

- o Definition:
 - An approach where the problem is solved by filling up a table (usually an array) iteratively, starting from the simplest subproblems.
- o Function:
 - Builds solutions to larger subproblems based on the solutions to smaller subproblems, avoiding recursion.

Functions and Purpose of Dynamic Programming:

1. Efficiency Optimization:

- o Time Complexity Reduction:
 - Converts exponential time algorithms into polynomial time by storing and reusing subproblem solutions.
- o Space-Time Tradeoff:
 - Uses additional memory to store results, trading space for time efficiency.

2. Problem Solving:

- o Complex Problem Decomposition:
 - Breaks down complex problems into manageable subproblems.
- o Systematic Solution Building:
 - Constructs solutions in a structured way, ensuring all possible cases are considered.

3. Applicability to Various Domains:

- o Computer Science and Mathematics:
 - Used in algorithms related to optimization, combinatorics, and numerical computations.
- o Real-World Applications:

- Applied in fields like operations research, economics, bioinformatics, and more.

Examples of Dynamic Programming Algorithms:

1. Fibonacci Sequence Calculation:

- Problem:
 - Compute the nth Fibonacci number, where each number is the sum of the two preceding ones.
- Dynamic Programming Solution:
 - Stores previously calculated Fibonacci numbers in an array or memoization cache to avoid redundant calculations.

Recursive Solution with Memoization:

```
def fibonacci(n, memo={}):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
    return memo[n]
```

2. Knapsack Problem:

- Problem:
 - Given a set of items with weights and values, determine the maximum value that can be carried in a knapsack of a fixed capacity.
- Dynamic Programming Solution:
 - Builds a table to store the maximum value that can be achieved with a given weight capacity, considering each item.

3. Longest Common Subsequence (LCS):

- Problem:
 - Find the longest subsequence common to two sequences.
- Dynamic Programming Solution:

- Constructs a matrix to store lengths of LCS for substrings, building up the solution from smaller substrings.

4. Shortest Path Algorithms:

- Bellman-Ford Algorithm:
 - Computes the shortest paths from a single source vertex to all other vertices in a weighted graph, accommodating negative weights.
- Floyd-Warshall Algorithm:
 - Finds the shortest paths between all pairs of vertices in a weighted graph.

How Dynamic Programming Works:

1. Identify if DP is Applicable:

- Check for optimal substructure and overlapping subproblems in the problem.

2. Define the Structure:

- Formulate the problem recursively by defining the relation between the problem and its subproblems.

3. Choose Memoization or Tabulation:

- Memoization: Use when a top-down approach is more intuitive.
- Tabulation: Use when a bottom-up approach is efficient, and recursion overhead is undesirable.

4. Implement the Algorithm:

- Memoization Example:

```
def fib(n, memo={}):
    if n in memo:
        return memo[n]

    if n <= 1:
        return n

    memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    return memo[n]

print(fib(10)) # Output: 55
```

- Tabulation Example:

```
def fib(n):
    table = [0] * (n + 1)
    table[0] = 0
    table[1] = 1

    for i in range(2, n + 1):
        table[i] = table[i - 1] + table[i - 2]

    return table[n]

print(fib(10)) # Output: 55
```

Advantages of Dynamic Programming:

- Efficiency: Significantly reduces computation time for problems with overlapping subproblems.
- Optimal Solutions: Guarantees finding an optimal solution when applicable.
- Versatility: Applicable to a wide range of problems in various fields.

Limitations of Dynamic Programming:

- Space Complexity: May require significant memory to store subproblem results.
- Problem Suitability: Not all problems exhibit the properties required for dynamic programming.
- Complexity in Formulation: Can be challenging to identify the subproblems and recursive relations.

Dynamic Programming is a powerful algorithmic paradigm used to solve complex problems efficiently by breaking them down into simpler, overlapping subproblems. By storing the results of these subproblems, dynamic programming avoids redundant computations, optimizing both time and space resources. It is characterized by two main properties:

- Optimal Substructure: The optimal solution can be constructed from optimal solutions of its subproblems.
- Overlapping Subproblems: The problem can be broken down into subproblems that are reused multiple times.

3.3 Searching and sorting algorithms (KT0303) (IAC0301)

Searching and sorting algorithms are fundamental computational procedures used to organise and retrieve data efficiently. Searching algorithms focus on finding a specific element within a data structure, while sorting algorithms arrange the elements of a data structure in a particular order (ascending or descending). These algorithms are essential for optimizing data processing and are foundational to computer science and programming.

Concepts of Searching Algorithms:

1. Purpose of Searching Algorithms:

- o Data Retrieval: Enable the efficient location of specific elements within data structures like arrays, lists, or databases.
- o Optimization: Reduce the time complexity of data access operations.

2. Types of Searching Algorithms:

o Linear Search (Sequential Search):

▪ Concept:

- Checks each element in the data structure sequentially until the desired element is found or the end is reached.

▪ Characteristics:

- Simple to implement.
- Time Complexity: $O(n)$, where n is the number of elements.
- Does not require the data to be sorted.

▪ Example:

```
def linear_search(arr, target):  
    for index, element in enumerate(arr):  
        if element == target:  
            return index # Target found  
    return -1 # Target not found
```

o Binary Search:

- Concept:
 - Efficiently finds an element in a sorted array by repeatedly dividing the search interval in half.
- Characteristics:
 - Time Complexity: $O(\log n)$.
 - Requires the data to be sorted.
 - More efficient than linear search for large datasets.
- Example:

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid # Target found
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1 # Target not found
```

- Hash Table Search:

- Concept:
 - Uses a hash function to map keys to indices in a table, allowing for near-constant time search operations.
- Characteristics:
 - Time Complexity: $O(1)$ on average.
 - Efficient for large datasets with unique keys.
 - Requires additional space for the hash table.

3. Functions of Searching Algorithms:

- o Data Retrieval:
 - Quickly access and retrieve data elements from storage.
- o Performance Optimization:
 - Improve the efficiency of applications by reducing search times.
- o Foundation for Complex Algorithms:
 - Serve as building blocks for more advanced computational procedures.

Concepts of Sorting Algorithms:

1. Purpose of Sorting Algorithms:

- o Data Organisation:
 - Arrange data elements in a specific order to facilitate efficient access and processing.
- o Preprocessing:
 - Often a prerequisite for other algorithms, such as binary search.

2. Types of Sorting Algorithms:

- o Bubble Sort:
 - Concept:
 - Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
 - Characteristics:
 - Simple but inefficient for large datasets.
 - Time Complexity: $O(n^2)$.
 - Example:

```

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
arr = [64, 25, 12, 22, 11]
bubble_sort(arr)
print("Bubble Sorted:", arr)

```

- o Selection Sort:

- Concept:
 - Selects the minimum element from the unsorted portion and swaps it with the first unsorted element.
- Characteristics:
 - Time Complexity: $O(n^2)$.
 - Simple implementation.
- Example:

```

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Selection Sorted:", arr)

```

- o Insertion Sort:

- Concept:
 - Builds the sorted array one item at a time by inserting elements into their correct position.
- Characteristics:
 - Time Complexity: $O(n^2)$.
 - Efficient for small or nearly sorted datasets.
- Example:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

- o Merge Sort:

- Concept:
 - Divides the list into halves, recursively sorts them, and then merges the sorted halves.
- Characteristics:
 - Time Complexity: $O(n \log n)$.
 - Stable sorting algorithm.
 - Requires additional space for merging.

- Example:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursive calls to sort both halves
        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        # Merge the two sorted halves
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Copy remaining elements of left_half if any
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        # Copy remaining elements of right_half if any
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

- Quick Sort:

- Concept:

- Selects a 'pivot' element and partitions the array into sub-arrays, then recursively sorts the sub-arrays.

- Characteristics:
 - Time Complexity: Average case $O(n \log n)$, worst-case $O(n^2)$.
 - In-place sorting algorithm (does not require additional storage).
 - Efficient for large datasets.
- Example:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quick_sort(less) + [pivot] + quick_sort(greater)
```

o Heap Sort:

- Concept:
 - Builds a max-heap from the data, then repeatedly extracts the maximum element to create a sorted array.
- Characteristics:
 - Time Complexity: $O(n \log n)$.
 - In-place algorithm.
 - Not stable (relative order of equal elements may not be preserved).
- Example:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1 # Left child index
    r = 2 * i + 2 # right child index
```

```

if l < n and arr[l] > arr[largest]:
    largest = l
if r < n and arr[r] > arr[largest]:
    largest = r

if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]
    heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from heap
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

```

3. Functions of Sorting Algorithms:

- o Improved Data Access:
 - Sorted data allows for faster searching and retrieval operations.
- o Data Analysis:
 - Facilitates easier analysis and visualization of data trends.
- o Preparation for Algorithms:
 - Many algorithms require sorted data as input.

Applications of Searching and Sorting Algorithms:

- Databases and File Systems:
 - o Indexing, query optimization, and data retrieval.

- E-commerce Platforms:
 - Sorting products by price, rating, or relevance.
- Operating Systems:
 - Task scheduling and resource management.
- Networking:
 - Routing algorithms and organising data packets.
- Machine Learning and Data Science:
 - Preprocessing data, feature ranking, and organising datasets.
- Computer Graphics and Games:
 - Sorting objects for rendering, collision detection.

Choosing the Right Algorithm:

- Dataset Size:
 - Small datasets: Simple algorithms like insertion sort or selection sort may suffice.
 - Large datasets: More efficient algorithms like merge sort or quick sort are preferred.
- Data Characteristics:
 - Nearly sorted data: Insertion sort or bubble sort can be efficient.
 - Random data: Algorithms with better average-case performance like quick sort or heap sort.
- Memory Constraints:
 - Limited memory: In-place algorithms like heap sort are advantageous.
 - No memory constraints: Stable algorithms like merge sort may be used.
- Stability Requirements:
 - If maintaining the relative order of equal elements is important, use stable sorts like merge sort or insertion sort.

Summary:

- Searching Algorithms:
 - Essential for finding elements within data structures.
 - Linear search is simple but inefficient for large datasets.
 - Binary search is efficient but requires sorted data.
- Sorting Algorithms:
 - Crucial for organising data to enhance performance of other operations.
 - Choice of algorithm depends on dataset size, data characteristics, and specific requirements.

3.4 Mathematical algorithms (KT0304) (IAC0301)

Mathematical algorithms are precise, step-by-step computational procedures used to solve mathematical problems or perform mathematical computations. They are fundamental tools in both mathematics and computer science, enabling the efficient and systematic handling of calculations, data analysis, and problem-solving tasks.

Concept of Mathematical Algorithms

1. Precision and Rigor

- Exact Procedures: Mathematical algorithms are designed with precise instructions that leave no ambiguity, ensuring consistent results.
- Logical Sequence: They follow a logical sequence of steps based on mathematical principles and theories.

2. Problem-Solving Framework

- General Applicability: These algorithms provide a framework for solving a wide range of mathematical problems, from simple arithmetic to complex numerical computations.
- Abstraction: They often abstract complex problems into simpler components, making them more manageable.

3. Computational Efficiency

- Optimization: Mathematical algorithms aim to perform computations in the most efficient manner possible, reducing time and resource usage.
- Complexity Analysis: Understanding the computational complexity of algorithms helps in selecting the most appropriate one for a given problem.

4. Foundation for Advanced Algorithms

- Building Blocks: They serve as foundational elements for more advanced algorithms in fields like cryptography, data science, machine learning, and operations research.

- Interdisciplinary Applications: Mathematical algorithms are essential in various disciplines, including physics, engineering, economics, and biology.

Functions and Purpose of Mathematical Algorithms

1. Numerical Computations

- Arithmetic Operations: Performing basic operations like addition, subtraction, multiplication, and division efficiently.
- Complex Calculations: Handling more complex computations such as exponentiation, logarithms, trigonometric functions, and calculus operations.

2. Problem Solving

- Equation Solving: Finding roots of equations, solutions to systems of equations, and optimization problems.
- Mathematical Modelling: Simulating real-world phenomena using mathematical models and algorithms to predict outcomes.

3. Data Analysis and Processing

- Statistical Algorithms: Computing statistical measures like mean, median, variance, and standard deviation.
- Signal Processing: Algorithms for filtering, transforming, and analysing signals in various domains.

4. Cryptography and Security

- Encryption Algorithms: Using mathematical principles to secure data through encryption and decryption processes.
- Hash Functions: Generating fixed-size outputs from variable-size inputs for data integrity and authentication.

5. Computational Geometry

- Geometric Calculations: Algorithms for calculating distances, areas, volumes, and other geometric properties.
- Graph Algorithms: Solving problems related to graphs, such as shortest paths, spanning trees, and network flows.

6. Numerical Methods

- o Approximation Techniques: Methods like numerical integration, differentiation, and root-finding algorithms.
- o Error Analysis: Understanding and minimizing errors in numerical computations.

Examples of Mathematical Algorithms

1. Euclidean Algorithm

- o Purpose: Computes the greatest common divisor (GCD) of two integers.
- o Concept:
 - Based on the principle that the GCD of two numbers also divides their difference.
- o Algorithm Steps:
 - Given two integers a and b , where $a > b$, replace a with b and b with $a \% b$ (the remainder of a divided by b), and repeat until b becomes zero. The GCD is the last non-zero value of a .
- o Example:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

2. Sieve of Eratosthenes

- o Purpose: Finds all prime numbers up to a specified integer n .
- o Concept:
 - Iteratively marks the multiples of each prime number starting from 2.
- o Algorithm Steps:
 - Create a list of numbers from 2 to n .
 - Start with the first number in the list (2), and eliminate all of its multiples.

- Move to the next unmarked number and repeat the process.
 - The remaining unmarked numbers are primes.
- o Example:

```
def sieve_of_eratosthenes(n):
    primes = [True] * (n + 1)
    p = 2

    while p * p <= n:
        if primes[p]:
            for i in range(p * p, n + 1, p):
                primes[i] = False
        p += 1

    return [p for p in range(2, n + 1) if primes[p]]
```

3. Newton-Raphson Method

- o Purpose: Finds successively better approximations to the roots (zeros) of a real-valued function.
- o Concept:
 - Uses the function and its derivative to iteratively converge to a root.
- o Algorithm Steps:
 - Start with an initial guess x_0 .
 - Iterate using the formula $x_1 = x_0 - f(x_0) / f'(x_0)$.
 - Repeat until the desired level of accuracy is achieved.
- o Example:

```
def newton_raphson(f, df, x0, tolerance=1e-7, max_iterations=1000):
    for _ in range(max_iterations):
        x1 = x0 - f(x0) / df(x0)
        if abs(x1 - x0) < tolerance:
            return x1
        x0 = x1
    raise ValueError("Root not found within maximum iterations")
```

4. Fast Fourier Transform (FFT)

- o Purpose: Computes the discrete Fourier transform (DFT) and its inverse efficiently.
- o Concept:
 - Breaks down a DFT of any composite size n into many smaller DFTs.
- o Applications:
 - Signal processing, image processing, solving partial differential equations.

5. Dijkstra's Algorithm

- o Purpose: Finds the shortest path between nodes in a graph with non-negative edge weights.
- o Concept:
 - Uses a priority queue to greedily select the next closest vertex.

Importance of Mathematical Algorithms

- Foundation of Computational Mathematics:
 - o Mathematical algorithms are essential for performing numerical computations that computers cannot handle analytically.
- Enabling Technological Advancements:
 - o They play a critical role in the development of technologies like computer graphics, simulations, and data encryption.
- Problem Solving and Innovation:
 - o Provide tools for scientists and engineers to model complex systems, analyze data, and develop solutions to real-world problems.

Mathematical algorithms are precise computational procedures designed to solve mathematical problems efficiently and accurately. They are fundamental to various fields, providing the tools necessary for numerical computations, data analysis, cryptography, and more.

3.5 Data structures algorithms (KT0305) (IAC0301)

Data Structures and Algorithms are fundamental concepts in computer science that work together to enable efficient data processing, storage, and retrieval. A data structure is a particular way of organising and storing data in a computer so that it can be accessed and modified efficiently. An algorithm is a step-by-step procedure or set of rules designed to perform a specific task or solve a particular problem.

Concept of Data Structures

1. Organisation of Data:

- o Data structures provide a means to manage large amounts of data efficiently, such as large databases and internet indexing services.
- o They define the way data is stored, accessed, and manipulated.

2. Types of Data Structures:

o Primitive Data Structures:

- Basic structures directly operated upon by machine-level instructions.
- Examples: Integers, Floats, Characters, Pointers.

o Non-Primitive Data Structures:

- More complex structures built from primitive data types.

- Divided into two categories:

- Linear Data Structures:

- Elements form a sequence or a linear list.

- Examples: Arrays, Linked Lists, Stacks, Queues.

- Non-Linear Data Structures:

- Elements are not in a sequence; they form a hierarchical relationship.

- Examples: Trees, Graphs, Heaps, Hash Tables.

3. Functions of Data Structures:

o Efficient Data Management:

- Optimize storage and retrieval of data.
- Facilitate operations like insertion, deletion, traversal, searching, and sorting.
- Abstraction:
 - Provide a level of abstraction to handle data efficiently without worrying about low-level details.
- Problem Solving:
 - Enable the implementation of efficient algorithms to solve complex computational problems.

Concept of Algorithms in Data Structures

1. Algorithms Utilizing Data Structures:
 - Algorithms are designed to perform operations on data structures.
 - The choice of data structure can greatly affect the efficiency of an algorithm.
2. Common Algorithms Associated with Data Structures:
 - Traversal Algorithms:
 - Access each data element exactly once to process it.
 - Examples: In-order, Pre-order, Post-order traversal of trees; Breadth-First Search (BFS) and Depth-First Search (DFS) in graphs.
 - Insertion and Deletion Algorithms:
 - Methods to add or remove elements from data structures.
 - Examples: Inserting a node into a linked list; Deleting a node from a binary search tree.
 - Searching Algorithms:
 - Find the location of a target value within a data structure.
 - Examples: Linear Search in arrays or lists; Binary Search in sorted arrays or binary search trees.
 - Sorting Algorithms:
 - Arrange elements in a particular order.

- Examples: Quick Sort, Merge Sort, Heap Sort, applicable to arrays and linked lists.
- Hashing Algorithms:
 - Map data of arbitrary size to data of fixed size.
 - Used in hash tables to enable fast data retrieval.

3. Efficiency Considerations:

- Time Complexity:
 - Measure of the time an algorithm takes to run as a function of the length of the input.
 - Big O notation is commonly used to describe time complexity.
- Space Complexity:
 - Measure of the amount of memory space required by an algorithm as a function of the input size.
- Optimization:
 - Selecting the most appropriate data structure and algorithm to optimize performance for specific operations.

Importance of Data Structures and Algorithms

1. Foundation of Software Development:
 - Essential for writing efficient and optimized code.
 - Enable developers to handle data effectively in applications.
2. Problem-Solving Skills:
 - Understanding data structures and algorithms enhances analytical thinking.
 - Helps in devising efficient solutions to complex computational problems.
3. Performance Improvement:

- Proper use of data structures and algorithms can significantly improve the performance of software applications.
- Critical in areas like database management, networking, artificial intelligence, and more.

4. Interview Preparation:

- Knowledge of data structures and algorithms is crucial for technical interviews in the software industry.

Examples of Data Structures and Their Algorithms

1. Arrays:

- Description:
 - Collection of elements identified by index or key.
 - Stored in contiguous memory locations.
- Algorithms:
 - Traversal: Accessing each element in the array.
 - Insertion/Deletion: Adding or removing elements (may require shifting elements).

2. Linked Lists:

- Description:
 - Sequence of nodes where each node contains data and a reference to the next node.
- Algorithms:
 - Insertion: Adding a node at the beginning, end, or middle.
 - Deletion: Removing a node by updating links.
 - Traversal: Visiting each node sequentially.

3. Stacks (LIFO – Last In, First Out)

Description:

A stack allows elements to be added and removed only from the top of the structure.

Common Operations:

- `push()`: Add an element to the top of the stack.
- `pop()`: Remove and return the top element.
- `peek()`: View the top element without removing it.

Python Example:

```
# Creating a simple stack using a list
stack = []

# Push items onto the stack
stack.append("Package A")
stack.append("Package B")

# Peek at the top item
print("Top item:", stack[-1]) # Output: Package B

# Pop the top item
delivered = stack.pop()
print("Delivered:", delivered) # Output: Package B
print("Remaining Stack:", stack) # Output: ['Package A']
```

4. Queues (FIFO – First In, First Out)

Description:

A queue allows elements to be added to the rear and removed from the front.

Common Operations:

- `enqueue()`: Add an element to the rear of the queue.
- `dequeue()`: Remove and return the front element.
- `peek()`: View the front element without removing it.

Python Example:

```
# Creating a simple queue using a list
```

```

from collections import deque

queue = deque()

# Enqueue items
queue.append("Package 1")
queue.append("Package 2")

# Peek at the front item
print("Next delivery:", queue[0]) # Output: Package 1

# Dequeue the front item
delivered = queue.popleft()
print("Delivered:", delivered) # Output: Package 1
print("Remaining Queue:", list(queue)) # Output: ['Package 2']

```

5. Trees

Description:

A tree is a non-linear data structure that represents hierarchical relationships. Each element in a tree is called a **node**, and each node has:

- A value
- A reference to child nodes (except for the **leaf nodes**, which have none)
- A single **root node** (the starting point)

A **Binary Tree** is a common type where each node has at most two children (left and right).

Basic Binary Tree Structure in Python

Python Example:

```

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None

```

```

    self.right = None

# Create nodes
root = Node("Package Center")
root.left = Node("Area A")
root.right = Node("Area B")

# Access nodes
print("Root:", root.value)
print("Left child:", root.left.value)
print("Right child:", root.right.value)

```

Tree Traversal Algorithms

Traversal is the process of visiting every node in the tree:

- **In-order:** Left → Root → Right
- **Pre-order:** Root → Left → Right
- **Post-order:** Left → Right → Root

A binary tree could represent a hierarchy of delivery regions or package categories, where searching through branches allows fast access to regional hubs.

6. Graphs

Description:

A graph is a non-linear data structure made up of:

- **Vertices (Nodes):** Represent locations, packages, or hubs.
- **Edges (Connections):** Represent paths or routes between vertices.

Graphs can be:

- **Directed (one-way routes)** or **Undirected (two-way routes)**
- **Weighted** (edges have values like distance or cost)

Graph Representation in Python

Adjacency List Example:

```
# Undirected graph
graph = {
    "Hub A": [ "Hub B", "Hub C"],
    "Hub B": [ "Hub A", "Hub D"],
    "Hub C": [ "Hub A"],
    "Hub D": [ "Hub B"]
}

# Print connections
for hub in graph:
    print(f"{hub} connects to {graph[hub]}")
```

Graph Traversal Algorithms

Breadth-First Search (BFS) – explores layer by layer

Depth-First Search (DFS) – explores as far as possible before backtracking

Shortest Path: Dijkstra's Algorithm, Bellman-Ford Algorithm.

Cycle Detection, Topological Sorting, etc.

Graphs are ideal for representing delivery networks. For example, you can use a **weighted graph** to calculate the shortest delivery route (like in Dijkstra's Algorithm).

7. Hash Tables:

- o Description:
 - Data structure that implements an associative array abstract data type.
 - Uses hash functions to compute an index into an array of buckets.
- o Algorithms:
 - Insertion: Placing an element based on its hash value.

- Deletion: Removing an element by locating it via its hash value.
- Searching: Retrieving an element using its key.

Data structures and algorithms are integral to computer science, enabling efficient data management and problem-solving. Understanding the characteristics of various data structures and the algorithms that operate on them allows developers to select the most appropriate tools for a given task, optimizing performance and resource utilization.

3.6 Flow charts (on paper) (KT0306) (IAC0301)

A flowchart is a graphical representation of an algorithm or a step-by-step process, using various symbols to denote different types of actions or steps, and arrows to show the flow of control from one step to the next. Flowcharts provide a visual way to understand the flow and logic of an algorithm, making it easier to analyse, design, and communicate complex processes.

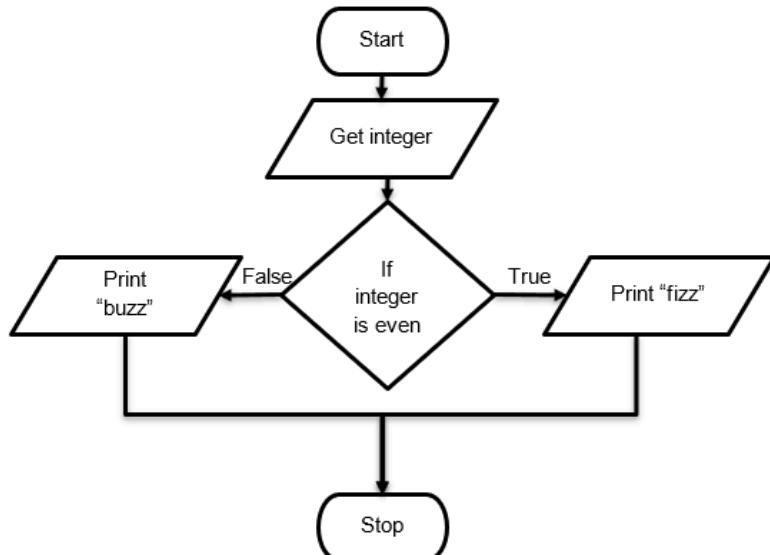
Flowcharts can be thought of as a graphical implementation of pseudocode. This is because they are more visually appealing and probably more readable than a "text-based" version of pseudocode. An example of pseudocode and the flowchart that would visualise the algorithm , is shown below:

Example

An algorithm that requests an integer from the user and prints "fizz" if the number is even or "buzz" if it is odd.

Pseudocode

```
request an integer from the user
if the integer is even
    print "fizz"
else if the integer is odd
    print "buzz"
```



There are a number of shapes or symbols used with flowcharts to denote the type of instruction or function you are using for each step of the algorithm. The most common symbols used for flowcharts are shown in the table below:

Symbol	Use
(oval)	Indicates the start and end of an algorithm.
(parallelogram)	Used to get or display any data involved in the algorithm.
(rectangle)	Indicates any processing or calculations that need to be done.
(diamond)	Indicates any decisions in an algorithm.

Purpose of Flowcharts in Algorithms:

1. Visualization of Processes:
 - Flowcharts translate textual algorithms into visual diagrams, helping to clarify the sequence of operations.

- They make it easier to comprehend the logic and flow of an algorithm, especially for complex or lengthy processes.

2. Problem Solving and Planning:

- Aid in the planning and development of algorithms by allowing programmers to map out the logic before coding.
- Help identify potential issues, inefficiencies, or errors in the logic early in the design phase.

3. Communication Tool:

- Serve as a universal language to communicate algorithms and processes among team members, regardless of programming language expertise.
- Useful for documentation, presentations, and educational purposes.

4. Debugging and Optimization:

- Facilitate the debugging process by providing a clear overview of the algorithm's flow.
- Help in identifying bottlenecks or redundant steps that can be optimized.

Common Flowchart Symbols:

Flowcharts use standardized symbols to represent different types of actions or steps. Here are some of the most used symbols:

1. Terminator (Oval):

- Purpose: Represents the start and end points of a flowchart.
- Usage: Labelled with "Start" or "End".

2. Process (Rectangle):

- Purpose: Denotes a process, action, or operation.
- Usage: Contains statements like calculations, assignments, or procedural steps.

3. Input/Output (Parallelogram):

- Purpose: Represents input or output operations.
- Usage: Reading data from input devices or displaying results.

4. Decision (Diamond):

- o Purpose: Indicates a decision point where the flow can branch based on a condition.
- o Usage: Contains a question or condition with branches for "Yes/No" or "True/False".

5. Connector (Circle):

- o Purpose: Used to connect different parts of a flowchart, especially when arrows cross or the flowchart spans multiple pages.
- o Usage: Labelled with letters or numbers to indicate continuation points.

6. Flow Lines (Arrows):

- o Purpose: Show the direction of the flow from one step to the next.
- o Usage: Connect symbols to indicate the sequence of operations.

7. Predefined Process (Rectangle with double-struck vertical edges):

- o Purpose: Represents a sub-process or a function that is defined elsewhere.
- o Usage: Used when a complex process is simplified into a single step.

8. Document (Rectangle with a wavy base):

- o Purpose: Indicates a document or report.
- o Usage: Used when the process involves generating or handling documents.

Creating a Flowchart (On Paper):

1. Identify the Algorithm's Steps:

- o Break down the algorithm into individual steps or actions.
- o Determine the sequence and logic flow, including decisions and loops.

2. Select Appropriate Symbols:

- o Use standard flowchart symbols to represent each step.
- o Ensure consistency and clarity in symbol usage.

3. Layout the Flowchart:

- o Begin with the "Start" terminator symbol at the top.
- o Arrange symbols from top to bottom or left to right, following the logical flow.
- o Use arrows to connect the symbols, indicating the direction of flow.

4. Incorporate Decision Points:

- o Use diamond-shaped decision symbols for branching logic.
- o Clearly label the conditions and the branches (e.g., "Yes/No" or "True/False").

5. Include Loops and Iterations:

- o Represent loops by connecting flow lines back to previous steps.
- o Ensure that loops have a termination condition to prevent infinite loops.

6. End with the "End" Terminator:

- o Conclude the flowchart with an "End" symbol to indicate the completion of the process.

Example: Flowchart for Finding the Maximum of Two Numbers

1. Start
2. Input: Read two numbers, A and B.
3. Decision: Is A > B?
 - o Yes: Proceed to step 4.
 - o No: Proceed to step 5.
4. Process: Set Max = A.
5. Process: Set Max = B.
6. Output: Display Max.
7. End

In the flowchart, you would represent this using the appropriate symbols, connecting them with arrows to show the flow.

Benefits of Using Flowcharts:

- Clarity and Understanding:
 - Visual representation aids in understanding complex algorithms.
 - Simplifies the communication of logic to others.
- Error Detection:
 - Easier to spot logical errors or omissions in the algorithm.
 - Facilitates testing of different scenarios.
- Efficient Coding:
 - Provides a clear blueprint for writing code.
 - Reduces the likelihood of bugs during implementation.
- Documentation:
 - Serves as valuable documentation for future reference or for new team members.
 - Enhances maintainability of the codebase.

Tips for Creating Effective Flowcharts:

- Keep it Simple:
 - Avoid overcrowding the flowchart; break it into smaller sections if necessary.
 - Use clear and concise labelling within symbols.
- Maintain Consistency:
 - Use standard symbols and consistent notation throughout the flowchart.
 - Ensure that the flow of control is logical and easy to follow.
- Use Connectors Wisely:
 - Employ connectors to handle complex flows or to continue on another page.
 - Label connectors clearly to prevent confusion.

- Validate the Flowchart:
 - Walk through the flowchart with different inputs to test its logic.
 - Confirm that all possible paths lead to a valid end point.

Flowcharts are invaluable tools in the basics of algorithms, providing a visual means to design, analyse, and communicate algorithms effectively. By representing the flow of control and the sequence of operations, flowcharts help in understanding the logic of an algorithm and serve as a bridge between conceptual design and actual implementation in code.



Formative Assessment Activity [number]

Basics of Algorithms

Complete the formative activity in your **KM4 Learner Workbook**,

Knowledge Topic KM-04-KT04:

Topic Code	KM-04-KT04
Topic	File Handling
Weight	20%

This knowledge topic will cover the following topic elements:

- KT0401 Concept, definition, and functions
- KT0402 Syntax
- KT0403 Inbuilt function
- KT0404 File operation
- KT0405 Directory
- KT0406 File modes
- KT0407 Zip files
- KT0408 Readline()
- KT0409 Binary files

After working through this knowledge topic, your competence will be assessed according to the following criterion/criteria:

- IAC0401 Definitions, functions and features of file handling in Python are understood and explained

KM-04-KT04 File Handling (IAC0401)

4.1 Concept, definition and functions (KT0401) (IAC0401)

File handling in Python refers to the process of interacting with files on the disk or other storage devices. It involves performing operations such as creating, opening, reading, writing, and closing files. File handling is essential for storing data persistently, retrieving data, and manipulating files within a Python program.

Concept of File Handling in Python

1. File Operations:

- o Opening a File:
 - Before performing any operation on a file (read or write), it must be opened using the built-in open() function.
 - The open() function returns a file object, which is used to interact with the file.
- o Reading from a File:
 - Data can be read from a file using methods like read(), readline(), or readlines().
- o Writing to a File:
 - Data can be written to a file using methods like write() or writelines().
- o Closing a File:
 - After all operations are completed, the file should be closed using the close() method to free up system resources.

2. File Modes:

- o Read Mode ('r'):
 - Opens a file for reading (default mode).
 - The file pointer is placed at the beginning of the file.
 - If the file does not exist, an error occurs.
- o Write Mode ('w'):

- Opens a file for writing.
 - Overwrites the file if it exists.
 - Creates a new file if it does not exist.
- Append Mode ('a'):
 - Opens a file for appending.
 - The file pointer is at the end of the file if it exists.
 - Creates a new file if it does not exist.
 - Read and Write Modes ('r+', 'w+', 'a+'):
 - Combine read and write operations.
 - 'r+'. Read and write. File must exist.
 - 'w+'. Write and read. Overwrites existing file or creates a new one.
 - 'a+'. Append and read. Creates a new file if it does not exist.
 - Binary Modes ('rb', 'wb', etc.):
 - Used for handling binary files like images, videos, or executable files.
 - Modes are the same as above but with a 'b' appended.

3. File Objects:

- File Pointer:
 - Keeps track of the current position in the file.
 - Methods like seek() and tell() are used to manipulate and obtain the file pointer position.
- Methods and Attributes:
 - read(size): Reads up to size bytes from the file.
 - readline(): Reads a single line from the file.
 - readlines(): Reads all lines and returns them as a list.
 - write(string): Writes a string to the file.
 - writelines(list): Writes a list of strings to the file.
 - close(): Closes the file.

Functions and Purpose of File Handling in Python

1. Data Persistence:

- o Storage of Data:

- Files allow data to persist beyond the life of the program execution.
- Useful for saving user data, application settings, logs, and more.

2. Data Retrieval and Manipulation:

- o Reading Data:

- Programs can read data from files for processing, analysis, or display.
- Enables applications to use pre-existing data.

- o Writing Data:

- Programs can output results, logs, or processed data to files.
- Facilitates data sharing and reporting.

3. Data Exchange:

- o Interoperability:

- Files serve as a medium for data exchange between different programs or systems.
- Common formats include text files, CSV, JSON, XML, etc.

4. File Management:

- o Organising Data:

- File handling allows for organising data into directories and files for better management.
- Supports operations like copying, moving, renaming, and deleting files.

5. Automation:

- o Batch Processing:

- Automate tasks like processing multiple files, generating reports, or transforming data.
- Essential in scripting and automation tasks.

Basic File Operations in Python

1. Opening a File

Use the `open()` function to open a file:

```
# General syntax
```

```
file_object = open('filename.txt', 'mode')

# Example: Open a file for reading
f = open('example.txt', 'r') # Opens the file in read mode
```

- Example:

```
f = open('example.txt', 'r')
content = f.read()
print(content)
f.close()
```

2. Reading from a File

- Reading the Entire File:

```
f = open('example.txt', 'r')
content = f.read()
print(content)
f.close()
```

- Reading Line by Line:

```
f = open('example.txt', 'r')
for line in f:
    print(line.strip())
f.close()
```

- Reading Specific Number of Characters:

```
f = open('example.txt', 'r')
content = f.read(10) # Reads the first 10 characters
print(content)
f.close()
```

3. Writing to a File

- Writing a String:

```
f = open('output.txt', 'w') # Opens the file in write mode
f.write('Hello, World!')
f.close()
```

- Writing Multiple Lines:

```
lines = ['Line 1\n', 'Line 2\n', 'Line 3\n']
f = open('output.txt', 'w')
f.writelines(lines)
f.close()
```

4. Closing a File

Always close the file after completing operations:

```
f.close() # Ensures data is saved and resources are freed
```

Using Context Managers

Python provides a `with`-statement that automatically handles opening and closing files using a context manager.

- Syntax:

```
with open('output.txt', 'w') as file_object:  
    # Perform file operations  
    file_object.write("Hello from a context manager!\n")  
    file_object.writelines(['Another line\n', 'Final line\n'])
```

- Example:

```
with open('example.txt', 'r') as f:  
    content = f.read()  
    print(content)  
# File is automatically closed after the with block
```

Example: Reading and Writing Files

Reading from a File and Writing to Another File:

```
# Reading from a file
```

```
with open('input.txt', 'r') as infile:  
    data = infile.read()  
  
    # Processing the data (e.g., converting to uppercase)  
    processed_data = data.upper()  
  
    # Writing the processed data to a new file  
    with open('output.txt', 'w') as outfile:  
        outfile.write(processed_data)  
  
    print("Data has been processed and written to output.txt.")
```

Explanation:

- Reading:
 - The input.txt file is opened in read mode.
 - The content is read into the variable data.
- Processing:
 - The data is converted to uppercase using the upper() method.
- Writing:
 - The output.txt file is opened in write mode.
 - The processed data is written to the file.
- Context Managers:
 - Using with ensures files are properly closed after their block is executed.

Error Handling in File Operations

When working with files, it's important to handle exceptions that may occur, such as FileNotFoundError, IOError, etc.

Example:

```
try:  
    with open('nonexistent_file.txt', 'r') as f:  
        data = f.read()  
except FileNotFoundError:  
    print("The file does not exist.")  
except IOError:  
    print("An error occurred while reading the file.")
```

Explanation:

- Try-Except Block:
 - Attempts to open and read the file.

- o Catches specific exceptions and handles them gracefully.

Summary

- File handling in Python allows for interaction with files on the storage device, enabling data persistence and manipulation.
- Key Operations:
 - o Opening files using `open()` with appropriate modes ('r', 'w', 'a', etc.).
 - o Reading from files using methods like `read()`, `readline()`, and iterating over the file object.
 - o Writing to files using `write()` and `writelines()`.
 - o Closing files using `close()` or by using context managers (`with` statement) for automatic handling.
- File Modes determine how files are opened and what operations are allowed.
- Context Managers provide a safe and efficient way to work with files, ensuring they are properly closed.
- Error Handling is crucial to manage exceptions that may arise during file operations.

4.2 Syntax (KT0402) (IAC0401)

File handling in Python involves using built-in functions and methods to interact with files on your system. Understanding the syntax is crucial for performing operations like creating, reading, writing, and closing files effectively.

Basic File Handling Syntax

1. Opening a File

To open a file in Python, use the `open()` function, which returns a file object.

```
file_object = open('filename', 'mode')
```

- `filename`: A string representing the name or path of the file.

- mode: A string specifying the mode in which the file is opened (e.g., 'r' for reading, 'w' for writing).

Common Modes:

- 'r': Read mode (default). Opens a file for reading.
- 'w': Write mode. Creates a new file or overwrites an existing file.
- 'a': Append mode. Adds content to the end of the file.
- 'r+'. Read and write mode. Reads and writes to the same file.
- Adding 'b' to any mode opens the file in binary mode (e.g., 'rb').

Example:

```
f = open('example.txt', 'r')
```

2. Reading from a File

a. Reading the Entire Content:

```
content = f.read()
```

- Reads the entire file content as a single string.

b. Reading Line by Line:

```
line = f.readline() # Reads one line at a time
```

c. Reading All Lines into a List:

```
lines = f.readlines() # Returns a list of lines
```

Example:

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

3. Writing to a File

a. Writing a String:

```
f.write('This is a line of text.\n')
```

b. Writing Multiple Lines:

```
lines = ['First line.\n', 'Second line.\n', 'Third line.\n']  
f.writelines(lines)
```

Example:

```
with open('output.txt', 'w') as f:  
    f.write('Hello, World!\n')
```

4. Closing a File

When not using a context manager (with statement), close the file using:

```
f.close()
```

- Closes the file and frees up system resources.

5. Using Context Managers

The with statement automatically handles opening and closing files.

Syntax:

```
with open('filename', 'mode') as file_object:  
    # Perform file operations
```

Example:

```
with open('data.txt', 'r') as f:  
    for line in f:  
        print(line.strip())
```

- Explanation: The file data.txt is opened in read mode. Each line is read and printed without extra whitespace.

File Positioning Methods

- f.tell(): Returns the current position of the file pointer.

```
position = f.tell()
```

- f.seek(offset, whence): Moves the file pointer to a specific location.
 - offset: Number of bytes to move.
 - whence: Reference point (0 for beginning, 1 for current position, 2 for end of file).

```
f.seek(0) # Move to the beginning of the file
```

Example: Complete File Handling

```
# Writing to a file  
with open('students.txt', 'w') as f:  
    f.write('Alice\n')  
    f.write('Bob\n')  
    f.write('Charlie\n')  
  
# Reading from the file  
with open('students.txt', 'r') as f:  
    students = f.readlines()  
  
# Displaying the content  
for student in students:  
    print(student.strip())
```

Explanation:

- Writing: Opens students.txt in write mode and writes three names.
- Reading: Opens the same file in read mode and reads all lines into a list.
- Displaying: Iterates over the list and prints each name.

Binary File Handling

When working with non-text files (like images or executables), use binary mode.

Example: Copying a Binary File:

```
with open('image.jpg', 'rb') as source_file:  
    data = source_file.read()  
  
with open('copy_image.jpg', 'wb') as dest_file:  
    dest_file.write(data)
```

- Explanation: Reads binary data from image.jpg and writes it to copy_image.jpg.

Error Handling

Use try-except blocks to handle exceptions.

Example:

```
try:  
    with open('nonexistent.txt', 'r') as f:  
        content = f.read()  
except FileNotFoundError:  
    print("File not found.")
```

- Explanation: Attempts to open a file that doesn't exist and handles the FileNotFoundError.

Summary of Key Syntax Elements

- Opening a File:

```
f = open('filename', 'mode')
```

- Reading Data:

```
data = f.read()
line = f.readline()
lines = f.readlines()
```

- Writing Data:

```
f.write('text')
f.writelines(['line1', 'line2'])
```

- Closing a File:

```
f.close()
```

- Using Context Managers:

```
with open('filename', 'mode') as f:
    # File operations
```

- File Pointer Methods:

```
f.seek(offset, whence)
position = f.tell()
```

By mastering the syntax of file handling in Python, you can efficiently read from and write to files, handle data processing tasks, and ensure that your programs interact with the file system correctly and safely.

4.3 Inbuilt function (KT0403) (IAC0401)

Inbuilt functions (also known as built-in functions) are functions that are provided by the Python interpreter and are always available without the need to import any additional modules. These functions perform a wide range of tasks and simplify programming by providing ready-to-use functionality.

Inbuilt Functions for File Handling in Python

File handling in Python relies on several key inbuilt functions and methods that allow you to interact with files on your system. These functions enable you to create, open, read, write, and close files efficiently.

1. open() Function

- Purpose:
 - The `open()` function is used to open a file and returns a file object, which allows you to perform operations on the file (e.g., read or write).
- Syntax:

```
file_object = open(file_name, mode)
```

- `file_name`: A string representing the name or path of the file.
- `mode`: A string specifying the mode in which the file is opened (e.g., '`r`' for reading, '`w`' for writing).

- Example:

```
f = open('example.txt', 'r') # Opens 'example.txt' in read mode
```

2. close() Method

- Purpose:
 - The `close()` method is used to close an open file. Closing a file frees up system resources and ensures that all data is properly written to the file.
- Syntax:

```
file_object.close()
```

- Example:

```
f.close() # Closes the file associated with file object 'f'
```

3. read() Method

- Purpose:
 - The read() method reads the content of a file.
- Syntax:

```
content = file_object.read(size)
```

- size (optional): The number of bytes to read. If omitted or negative, reads the entire file.
- Example:

```
content = f.read() # Reads the entire content of the file
```

4. readline() Method

- Purpose:
 - The readline() method reads a single line from a file.
- Syntax:

```
line = file_object.readline()
```

- Example:

```
line = f.readline() # Reads one line from the file
```

5. readlines() Method

- Purpose:
 - The `readlines()` method reads all lines from a file and returns them as a list of strings.
- Syntax:

```
lines = file_object.readlines()
```

- Example:

```
lines = f.readlines() # Reads all lines and stores them in a list
```

6. write() Method

- Purpose:
 - The `write()` method writes a string to a file.
- Syntax:

```
file_object.write(string)
```

- Example:

```
f.write('Hello, World!\n') # Writes 'Hello, World!' to the file
```

7. writelines() Method

- Purpose:
 - The `writelines()` method writes a list of strings to a file.
- Syntax:

```
file_object.writelines(list_of_strings)
```

- Example:

```
lines = ['Line 1\n', 'Line 2\n', 'Line 3\n']
f.writelines(lines) # Writes multiple lines to the file
```

8. tell() Method

- Purpose:
 - The tell() method returns the current position of the file pointer within the file.
- Syntax:

```
position = file_object.tell()
```

- Example:

```
pos = f.tell() # Gets the current file pointer position
```

9. seek() Method

- Purpose:
 - The seek() method moves the file pointer to a specified location within the file.
- Syntax:

```
file_object.seek(offset, whence)
```

- offset: The number of bytes to move the file pointer.
- whence (optional): The reference point (0 for beginning of the file, 1 for current position, 2 for end of the file). Default is 0.

- Example:

```
f.seek(0) # Moves the file pointer to the beginning of the file
```

10. with Statement (Context Manager)

- Purpose:
 - While not a function, the with statement is used with file handling to ensure that files are properly closed after their suite finishes, even if an exception is raised.
- Syntax:

```
with open(file_name, mode) as file_object:  
    # Perform file operations
```

- Example:

```
# Open the file in read mode  
with open('example.txt', 'r') as f:  
    content = f.read()  
    print(content)  
# File is automatically closed when the block ends
```

Example: Using Inbuilt Functions for File Handling

Reading from a File:

```
# Open the file in read mode  
with open('sample.txt', 'r') as f:  
    # Read the entire content  
    content = f.read()  
    print(content)
```

Writing to a File:

```
# Open the file in write mode  
with open('output.txt', 'w') as f:  
    # Write a string to the file  
    f.write('This is an example of writing to a file.\n')
```

Appending to a File:

```
# Open the file in append mode
with open('output.txt', 'a') as f:
    # Write another line to the file
    f.write('This line is appended to the file.\n')
```

Reading Lines from a File:

```
with open('sample.txt', 'r') as f:
    # Read lines into a list
    lines = f.readlines()

    # Iterate over the list and print each line
    for line in lines:
        print(line.strip())
```

Summary

- Inbuilt functions in Python for file handling provide a straightforward and efficient way to interact with files.
- Key Functions and Methods:
 - `open()`: Opens a file and returns a file object.
 - `close()`: Closes an open file.
 - `read()`, `readline()`, `readlines()`: Read data from a file.
 - `write()`, `writelines()`: Write data to a file.
 - `tell()`, `seek()`: Manage the file pointer position.
- Best Practices:
 - Use the `with` statement to handle files, as it ensures proper closure of the file.
 - Always specify the correct mode when opening a file ('`r`', '`w`', '`a`', etc.).

- o Handle exceptions using try-except blocks when working with files to manage errors like FileNotFoundError.

4.4 File operation (KT0404) (IAC0401)

File operations refer to the various actions you can perform on files, such as creating, opening, reading, writing, appending, and closing files. These operations allow your Python programs to interact with files stored on your computer's file system, enabling data persistence and manipulation.

Key File Operations in Python

1. Opening a File

- o Purpose:
 - Before you can read from or write to a file, you must open it using the `open()` function.
 - Opening a file establishes a connection between the file and your program.
- o Syntax:

```
file_object = open('filename', 'mode')
```

- `filename`: The name or path of the file.
- `mode`: The mode in which the file is opened (e.g., `'r'` for reading, `'w'` for writing).

- o Example:

```
f = open('data.txt', 'r') # Opens 'data.txt' in read mode
```

2. Reading from a File

- o Purpose:
 - To extract data from a file for processing.

- o Methods:
 - `read(size)`: Reads up to size bytes from the file (if size is omitted, reads the entire file).
 - `readline()`: Reads one line from the file.
 - `readlines()`: Reads all lines and returns them as a list of strings.

- o Example:

```
with open('data.txt', 'r') as f:  
    content = f.read() # Reads the entire file  
    print(content)
```

3. Writing to a File

- o Purpose:
 - To write data to a file, creating a new file or overwriting an existing one.
- o Methods:
 - `write(string)`: Writes a string to the file.
 - `writelines(list_of_strings)`: Writes a list of strings to the file.
- o Example:

```
with open('output.txt', 'w') as f:  
    f.write('Hello, World!\n')
```

4. Appending to a File

- o Purpose:
 - To add new data to the end of an existing file without altering the existing content.
- o Example:

```
with open('log.txt', 'a') as f:  
    f.write('New log entry\n')
```

5. Closing a File

- o Purpose:
 - To release the resources associated with the file and ensure data is properly saved.
- o Method:
 - `close()`: Closes the file object.
- o Note:
 - When using the `with` statement (context manager), files are automatically closed after the block of code is executed.

6. Using the with Statement

- o Purpose:
 - Simplifies file handling by automatically managing file closure.
- o Syntax:

```
with open('filename', 'mode') as file_object:  
    # Perform file operations
```

- o Example:

```
with open('data.txt', 'r') as f:  
    content = f.read()  
# No need to call f.close()
```

7. File Positioning

- o Purpose:
 - To control where in the file reading or writing begins.
- o Methods:
 - `seek(offset, whence)`: Moves the file pointer to a specific location.
 - `offset`: Number of bytes to move.
 - `whence`:

- 0 (default): Beginning of the file.
 - 1: Current position.
 - 2: End of the file.
- `tell()`: Returns the current position of the file pointer.

- Example:

```
with open('data.txt', 'r') as f:  
    f.seek(10)                      # Move to the 11th byte in the file (index starts  
at 0)  
    data = f.read(5)                  # Read the next 5 bytes  
    print(f.tell())                  # Outputs the current file pointer position
```

8. Deleting and Renaming Files

- Purpose:

- To manage files at the system level using the `os` module.

- Methods:

- `os.remove('filename')`: Deletes a file.
- `os.rename('old_name', 'new_name')`: Renames a file.

- Example:

```
import os  
  
os.rename('old.txt', 'new.txt')
```

Common File Modes

Mode	Description
'r'	Read mode (default). Opens a file for reading.
'w'	Write mode. Creates a new file or overwrites existing content.
'a'	Append mode. Adds new data to the end of the file.
'r+'	Read and write mode. File must exist.
'w+'	Write and read mode. Overwrites existing file or creates new one.
'a+'	Append and read mode. Creates file if it doesn't exist.
'rb', 'wb', 'ab'	Binary modes for reading, writing, appending.

Example: Reading and Writing Files

Reading a File:

```
# Read the entire content of a file
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

Writing to a File:

```
# Write data to a file (overwrites if file exists)
with open('output.txt', 'w') as file:
    file.write('This is a sample text.\n')
    file.write('Writing to files is easy!\n')
```

Appending to a File:

```
# Append data to an existing file
with open('output.txt', 'a') as file:
    file.write('Appending a new line.\n')
```

Reading a File Line by Line:

```
# Read Lines one by one
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

Using File Positioning:

```
with open('example.txt', 'r') as file:
    file.seek(0, 2)           # Move to the end of the file
    position = file.tell()    # Get the current position (end of file)
    print(f'File size: {position} bytes')
```

Best Practices for File Operations

1. Use the with Statement:

- o Automatically handles closing files, even if an error occurs.
- o Improves code readability and safety.

2. Exception Handling:

- o Anticipate and handle potential errors using try-except blocks.
- o Example:

```
try:
    with open('nonexistent.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print('File not found.')
```

3. Choose the Correct File Mode:

- o Be cautious with modes like 'w' and 'w+' as they overwrite existing files.

- o Use 'a' or 'a+' to append data without losing existing content.

4. Handle Binary Files Appropriately:

- o Use binary modes ('rb', 'wb', 'ab') when working with non-text files like images or executables.

Summary

- File operations in Python are essential for interacting with files, enabling data to be stored and retrieved from the file system.
- Key Operations Include:
 - o Opening files with `open()`.
 - o Reading data using methods like `read()`, `readline()`, and `readlines()`.
 - o Writing data with `write()` and `writelines()`.
 - o Appending data to existing files.
 - o Closing files using `close()` or by leveraging the `with` statement for automatic management.
 - o File positioning using `seek()` and `tell()` to control where data is read or written.
- Best Practices:
 - o Use the `with` statement for cleaner and safer code.
 - o Handle exceptions to make your program robust.
 - o Choose appropriate file modes to prevent data loss.

4.5 Directory (KT0405) (IAC0401)

A directory (also known as a folder) is a file system structure that contains files and other directories. Directories are used to organise files in a hierarchical manner, allowing for efficient management and access to data stored on a computer or storage device.

Concept of Directories in Python

1. File System Hierarchy:

- o The file system of an operating system is organised in a hierarchical structure, starting from a root directory and branching into subdirectories and files.
- o Directories can contain files as well as other directories, forming a tree-like structure.

2. Pathnames:

- o Absolute Path:
 - Specifies the complete path from the root directory to the target file or directory.
 - It is independent of the current working directory.
 - Example (Windows): C:\Users\Username\Documents\file.txt
 - Example (Unix/Linux): /home/username/documents/file.txt
- o Relative Path:
 - Specifies the path relative to the current working directory.
 - Example: If the current working directory is /home/username, then documents/file.txt is a relative path to /home/username/documents/file.txt.

3. Current Working Directory (CWD):

- o The directory in which a Python script or interactive session is currently operating.
- o You can get or change the CWD using Python's os module.

Functions and Purpose of Directories in Python File Handling

1. Organisation of Files:

- o Directories help in logically organising files, making it easier to manage and access data.
- o They allow grouping related files together, improving file management and navigation.

2. Navigating the File System:

- o Python provides modules and functions to navigate the file system, access files in different directories, and perform operations like creating, deleting, or listing directories.

3. File Management Operations:

- o Creating Directories: You can create new directories to organise files.
- o Deleting Directories: Remove directories when they are no longer needed.
- o Listing Contents: Retrieve a list of files and subdirectories within a directory.
- o Changing Directories: Move the current working directory to a different location in the file system.

Using the os and os.path Modules

Python's built-in `os` module provides functions for interacting with the operating system, including directory operations. The `os.path` module contains functions for manipulating pathnames.

Common Directory Operations:

1. Importing the Modules:

```
import os
```

2. Getting the Current Working Directory:

```
cwd = os.getcwd()  
print(f"Current Working Directory: {cwd}")
```

3. Changing the Current Working Directory:

```
os.chdir('/path/to/new/directory')
```

4. Listing the Contents of a Directory:

```
contents = os.listdir('/path/to/directory')  
print(contents)
```

5. Creating a New Directory:

- o Create a Single Directory:

```
os.mkdir('new_directory')
```

- o Create Nested Directories:

```
os.makedirs('parent_directory/child_directory')
```

6. Removing a Directory:

- o Remove an Empty Directory:

```
os.rmdir('empty_directory')
```

- o Remove a Directory and Its Contents:

```
import shutil  
shutil.rmtree('directory_to_remove')
```

Note: Be cautious when using `shutil.rmtree()` as it permanently deletes the directory and all its contents.

7. Checking if a Path is a File or Directory:

```
path = '/path/to/check'
if os.path.isdir(path):
    print(f"{path} is a directory")
elif os.path.isfile(path):
    print(f"{path} is a file")
else:
    print(f"{path} does not exist")
```

Example: Working with Directories

Scenario: You want to organise a set of files into a new directory.

Steps:

1. Create a New Directory:

```
import os

new_dir = 'organized_files'
if not os.path.exists(new_dir):
    os.mkdir(new_dir)
    print(f"Directory '{new_dir}' created.")
else:
    print(f"Directory '{new_dir}' already exists.")
```

2. List Files in the Current Directory:

```
files = [f for f in os.listdir('.') if os.path.isfile(f)]
print("Files in the current directory:")
for file in files:
    print(file)
```

3. Move Files into the New Directory:

```
import shutil

for file in files:
    shutil.move(file, new_dir)
    print(f"Moved '{file}' to '{new_dir}'")
```

Explanation:

- Checking if the Directory Exists:
 - os.path.exists(new_dir) checks whether the directory already exists to avoid an error when trying to create it.
- Listing Files:
 - os.listdir('') lists all items in the current directory.
 - [f for f in os.listdir('') if os.path.isfile(f)] filters the list to include only files.
- Moving Files:
 - shutil.move(src, dst) moves files from the source (src) to the destination (dst).

Cross-Platform Path Handling with os.path and pathlib

When working with file paths, it's important to handle differences between operating systems.

1. Using os.path.join():

- Joins path components intelligently using the correct path separator for the operating system.

```
path = os.path.join('folder', 'subfolder', 'file.txt')
print(path) # Outputs 'folder/subfolder/file.txt' on Unix, 'folder\subfolder\file.txt'
```

2. Using the pathlib Module (Python 3.4+):

- o Provides an object-oriented approach to handling file paths.

```
from pathlib import Path

# Create a Path object
p = Path('folder') / 'subfolder' / 'file.txt'
print(p) # Outputs 'folder/subfolder/file.txt' or 'folder\subfolder\file.txt'

# Check if a path exists
if p.exists():
    print(f"{p} exists")
```

Summary

- Directories are essential for organising files within the file system, enabling efficient data management.
- Python Modules for Directory Operations:
 - o The os module provides functions for interacting with the operating system, including directory manipulation.
 - o The os.path module offers utilities for handling file paths.
 - o The shutil module provides functions for high-level file operations like copying and removing directories.
 - o The pathlib module (available in Python 3.4 and later) provides an object-oriented interface for file system paths.
- Common Directory Operations Include:
 - o Getting and changing the current working directory.
 - o Listing the contents of a directory.
 - o Creating and removing directories.
 - o Navigating the file system and handling file paths.
- Best Practices:

- o Use `os.path.join()` or `pathlib` to construct file paths in a way that is compatible across different operating systems.
- o Check for the existence of files or directories before performing operations to prevent errors.
- o Handle exceptions that may arise during file and directory operations to make your code robust.

4.6 File modes (KT0406) (IAC0401)

File modes are strings that specify the type of operations you intend to perform on a file when you open it using the `open()` function. File modes determine how the file will be handled—whether you want to read from it, write to it, append data, or perform both reading and writing. They also specify whether the file should be treated as a text file or a binary file.

Key File Modes in Python

When working with files, you must specify the mode in which you want to open the file. The most used file modes are:

1. Read Mode ('r'):

- o Purpose: Opens a file for reading only.
- o Behaviour:
 - The file pointer is placed at the beginning of the file.
 - If the file does not exist, a `FileNotFoundException` is raised.
- o Syntax:

```
f = open('filename.txt', 'r')
```

- o Example:

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

2. Write Mode ('w'):

- o Purpose: Opens a file for writing.
- o Behaviour:
 - If the file exists, it is truncated (emptied) before writing.
 - If the file does not exist, it is created.
- o Syntax:

```
f = open('filename.txt', 'w')
```

- o Example:

```
with open('output.txt', 'w') as file:  
    file.write('This is a new file.\n')
```

3. Append Mode ('a'):

- o Purpose: Opens a file for appending.
- o Behaviour:
 - The file pointer is at the end of the file.
 - If the file does not exist, it is created.
 - Data written to the file is added after the existing content.
- o Syntax:

```
f = open('filename.txt', 'a')
```

- o Example:

```
with open('log.txt', 'a') as file:  
    file.write('New log entry.\n')
```

4. Read and Write Mode ('r+'):

- o Purpose: Opens a file for both reading and writing.
- o Behaviour:
 - The file pointer is placed at the beginning of the file.
 - If the file does not exist, a FileNotFoundError is raised.
- o Syntax:

```
f = open('filename.txt', 'r+')
```

- Example:

```
with open('notes.txt', 'r+') as file:  
    content = file.read()  
    file.write('\nAdditional notes.')
```

5. Write and Read Mode ('w'):

- Purpose: Opens a file for writing and reading.
- Behaviour:
 - If the file exists, it is truncated.
 - If the file does not exist, it is created.
- Syntax:

```
f = open('filename.txt', 'w+')
```

- Example:

```
with open('data.txt', 'w+') as file:  
    file.write('Starting fresh.')  
    file.seek(0)  
    print(file.read())
```

6. Append and Read Mode ('a'):

- Purpose: Opens a file for both appending and reading.
- Behaviour:
 - The file pointer is at the end of the file for writing.
 - If the file does not exist, it is created.
 - Allows reading from any position in the file.
- Syntax:

```
f = open('filename.txt', 'a+')
```

- o Example:

```
with open('diary.txt', 'a+') as file:  
    file.write('New entry.\n')  
    file.seek(0)  
    print(file.read())
```

7. Binary Modes:

- o Purpose: Used when working with binary files like images, audio files, or executables.
- o Common Binary Modes:
 - 'rb': Read binary.
 - 'wb': Write binary.
 - 'ab': Append binary.
 - 'rb+': Read and write binary.
 - 'wb+': Write and read binary.
 - 'ab+': Append and read binary.
- o Syntax:

```
f = open('image.png', 'rb')
```

- o Example:

```
# Copying a binary file  
with open('source.jpg', 'rb') as source_file:  
    data = source_file.read()  
with open('copy.jpg', 'wb') as dest_file:  
    dest_file.write(data)
```

Understanding File Mode Behaviours

- Truncation:
 - In 'w' and 'w+' modes, if the file already exists, its content is erased before writing begins.
- Creation of New Files:
 - Modes like 'w', 'w+', 'a', and 'a+' will create the file if it does not exist.
- File Pointer Position:
 - In 'r' and 'r+' modes, the file pointer starts at the beginning of the file.
 - In 'a' and 'a+' modes, the file pointer is at the end of the file for writing operations.

Choosing the Right File Mode

- When to Use 'r':
 - To read data from an existing file.
 - Raises an error if the file does not exist.
- When to Use 'w':
 - To create a new file or overwrite an existing file.
 - Use with caution to avoid unintentional data loss.
- When to Use 'a':
 - To add new data to the end of an existing file without altering its existing content.
 - Creates the file if it does not exist.
- When to Use 'r+':
 - To read from and write to the same file without truncating it.
 - File must exist; otherwise, an error is raised.
- When to Use 'w+':
 - To write to and read from a file.

- Truncates the file if it exists or creates a new one.
- When to Use 'a+':
 - To append to and read from a file.
 - The file pointer is at the end of the file for writing.
- When to Use Binary Modes ('b'):
 - When dealing with non-text files (e.g., images, audio files).
 - Ensures data is read and written as bytes.

Examples of Using File Modes

Reading a File ('r'):

```
try:  
    with open('example.txt', 'r') as file:  
        content = file.read()  
        print(content)  
except FileNotFoundError:  
    print("The file does not exist.")
```

Writing to a File ('w'):

```
with open('new_file.txt', 'w') as file:  
    file.write('This is a new file created for writing.\n')
```

Appending to a File ('a'):

```
with open('log.txt', 'a') as file:  
    file.write('Appended new log entry.\n')
```

Reading and Writing ('r+'):

```
with open('data.txt', 'r+') as file:  
    data = file.read()  
    file.seek(0)  
    file.write('Updated data.\n')
```

Binary File Handling ('rb', 'wb'):

```
with open('picture.jpg', 'rb') as file:  
    binary_data = file.read()  
with open('copy_picture.jpg', 'wb') as file:  
    file.write(binary_data)
```

Important Considerations

- Avoiding Data Loss:
 - Be cautious when using 'w' or 'w+' modes, as they will overwrite existing files.
- File Existence:
 - 'r' and 'r+' modes require the file to exist.
 - 'w', 'w+', 'a', and 'a+' will create the file if it does not exist.
- Text vs. Binary Files:
 - Text modes (default) handle encoding and decoding of data.
 - Binary modes read and write data as bytes, which is essential for binary files.

Summary

- File modes in Python are essential for specifying how a file should be opened and what operations are permitted.

- Choosing the correct file mode ensures that your program interacts with files safely and as intended.
- Understanding the behaviour of each mode helps prevent common errors, such as data loss or file not found exceptions.
- Best Practices:
 - Always use the appropriate mode for your use case.
 - Handle exceptions when opening files, especially in read modes.
 - Use binary modes for non-text files to maintain data integrity.

4.7 Zip files (KT0407) (IAC0401)

A Zip file is a compressed archive that contains one or more files or directories bundled together into a single file for easier transportation, storage, and compression. In Python, handling Zip files involves creating, reading, writing, and extracting compressed files using the built-in zipfile module. This allows for efficient file management, reduced storage space, and convenient file distribution.

Concept of Zip Files in Python

1. Compression and Archiving:

- o Compression: Reduces the size of files by eliminating redundancies, making storage and transfer more efficient.
- o Archiving: Combines multiple files and directories into a single file, preserving the directory structure.

2. The zipfile Module:

- o Purpose: Provides tools for creating, reading, writing, appending, and extracting ZIP files.
- o Classes and Methods:
 - ZipFile class: Core class for working with ZIP archives.
 - Methods like write(), read(), extract(), extractall(), namelist(), etc.

3. Modes of Opening Zip Files:

- o Read Mode ('r'): Open an existing ZIP file for reading.
- o Write Mode ('w'): Create a new ZIP file or overwrite an existing one.
- o Append Mode ('a'): Append files to an existing ZIP archive.
- o Exclusive Creation Mode ('x'): Create a new ZIP file, failing if it already exists.

4. Compression Types:

- o ZIP_STORED: No compression is applied; files are stored as-is.
- o ZIP_DEFLATED: Uses the Deflate compression algorithm (requires zlib).
- o ZIP_BZIP2: Uses BZIP2 compression (requires Python 3.3+).
- o ZIP_LZMA: Uses LZMA compression (requires Python 3.3+).

Functions and Purpose of Zip Files in Python File Handling

1. Efficient Storage:

- o Reduced File Size: Compressing files saves disk space.
- o Batch File Management: Bundling multiple files into one simplifies storage.

2. Data Transfer:

- o Easier Distribution: Sharing one archive file is more convenient than multiple files.
- o Faster Upload/Download: Smaller file sizes reduce transfer times.

3. Data Organisation:

- o Archiving: Keeps related files together, maintaining directory structures.
- o Backup and Preservation: Useful for creating backups of files and directories.

4. Security (with encryption):

- o Password Protection: Although the zipfile module has limited support for encryption, it can handle basic password-protected ZIP files.

Basic Operations with Zip Files in Python

1. Importing the zipfile Module

```
import zipfile
```

2. Creating a Zip File

- Writing Files to a Zip Archive:

```
with zipfile.ZipFile('archive.zip', 'w') as zipf:  
    zipf.write('file1.txt')  
    zipf.write('file2.txt')
```

- o Explanation:

- 'archive.zip': Name of the ZIP file to create.
 - 'w': Write mode; creates a new ZIP file.
 - zipf.write('filename'): Adds the specified file to the ZIP archive.
- Adding Files from a Directory:

```
import os

with zipfile.ZipFile('my_archive.zip', 'w') as zipf:
    for foldername, subfolders, filenames in os.walk('my_folder'):
        for filename in filenames:
            filepath = os.path.join(foldername, filename)
            zipf.write(filepath, arcname=os.path.relpath(filepath, 'my_folder'))
```

- Explanation:

- Uses os.walk() to traverse the directory.
- arcname parameter ensures the directory structure is preserved inside the ZIP file.

3. Reading a Zip File

- Listing Contents:

```
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    print(zipf.namelist())
```

- Output: List of file names in the ZIP archive.

- Reading a Specific File:

```
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    with zipf.open('file1.txt') as file:
        content = file.read()
        print(content.decode('utf-8'))
```

- Explanation:

- zipf.open('file1.txt'): Opens a file within the ZIP archive.

- `content.decode('utf-8')`: Decodes the byte content to a string.

4. Extracting Files from a Zip Archive

- Extracting All Files:

```
with zipfile.ZipFile('archive.zip', 'r') as zipf:  
    zipf.extractall('extracted_files')
```

- Explanation:

- Extracts all contents to the directory 'extracted_files'.

- Extracting a Specific File:

```
with zipfile.ZipFile('archive.zip', 'r') as zipf:  
    zipf.extract('file1.txt', 'extracted_files')
```

- Explanation:

- Extracts 'file1.txt' to the directory 'extracted_files'.

5. Appending to an Existing Zip File

```
with zipfile.ZipFile('archive.zip', 'a') as zipf:  
    zipf.write('new_file.txt')
```

- Explanation:

- 'a': Append mode; adds 'new_file.txt' to 'archive.zip'.

6. Compressing with Different Compression Types

```
with zipfile.ZipFile('compressed.zip', 'w', compression=zipfile.ZIP_DEFLATED) as zipf:  
    zipf.write('large_file.txt')
```

- Explanation:

- `compression=zipfile.ZIP_DEFLATED`: Applies Deflate compression to reduce file size.

Example: Creating and Extracting a Zip File

Creating a Zip Archive:

```
import zipfile

# Files to be zipped
files_to_zip = ['document.txt', 'image.png', 'data.csv']

# Creating a ZIP file
with zipfile.ZipFile('archive.zip', 'w') as zipf:
    for file in files_to_zip:
        zipf.write(file)

print("Zip file 'archive.zip' created successfully.")
```

Extracting the Zip Archive:

```
# Extracting all files
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    zipf.extractall('extracted')

print("Files extracted to 'extracted' directory.")
```

Handling Password-Protected Zip Files

- Creating a Password-Protected Zip File:
 - The zipfile module does not support creating encrypted Zip files directly.
 - You can use third-party libraries like pyminizip or zipfile36.
- Extracting a Password-Protected Zip File:

```
with zipfile.ZipFile('protected.zip', 'r') as zipf:
    zipf.extractall(pwd=b'my password')
```

- o Note:
 - Password must be provided as bytes (b'my password').

Advantages of Using Zip Files

1. Storage Efficiency:
 - o Compressed files take up less disk space.
 - o Useful for archiving large amounts of data.
2. Data Transfer Optimization:
 - o Smaller file sizes lead to faster uploads and downloads.
 - o Simplifies sending multiple files by bundling them into one archive.
3. Organisational Benefits:
 - o Keeps related files together.
 - o Preserves directory structures within the archive.
4. Platform Independence:
 - o Zip files are widely supported across different operating systems.

Best Practices

- Use Context Managers:
 - o Always use `with` statements to ensure files are properly closed after operations.
- Handle Exceptions:
 - o Use `try-except` blocks to catch and handle exceptions like `FileNotFoundException` or `zipfile.BadZipFile`.

```
try:  
    with zipfile.ZipFile('archive.zip', 'r') as zipf:  
        zipf.extractall()  
except zipfile.BadZipFile:  
    print("Error: The file is not a valid ZIP archive.")
```

- Verify Integrity:
 - Use zipf.testzip() to check for bad files within the archive.

```
with zipfile.ZipFile('archive.zip', 'r') as zipf:  
    if zipf.testzip() is not None:  
        print("Warning: The ZIP file contains corrupt files.")
```

- Compression Choices:
 - Choose the appropriate compression method based on your needs for speed and compression ratio.

Summary

- Zip files allow you to compress and archive multiple files and directories into a single file, making storage and transfer more efficient.
- Python's zipfile module provides a comprehensive set of tools for working with ZIP archives, including:
 - Creating new ZIP files and adding files to them.
 - Reading and extracting files from existing ZIP archives.
 - Listing the contents of ZIP files.
 - Handling different compression methods.
- Key Operations:
 - Creating Zip Files: Use ZipFile in write mode and add files using write().
 - Reading Zip Files: Open in read mode and use methods like namelist() and open().
 - Extracting Files: Use extract() for individual files or extractall() for all contents.
 - Appending to Zip Files: Open in append mode and add files.
- Best Practices:
 - Use context managers (with statements) to manage resources.
 - Handle exceptions to ensure robustness.
 - Be mindful of compression types and encryption requirements.

4.8 Readline() (KT0408) (IAC0401)

The readline() method is a built-in function used to read a single line from a file. When you open a file for reading, readline() allows you to retrieve one line at a time, making it particularly useful for processing files line by line without loading the entire file into memory.

Key Aspects of readline():

1. Reads One Line at a Time:

o Functionality:

- Each call to readline() reads the text from the current file position up to and including the next newline character (\n).
- The method returns the line as a string, including the newline character at the end unless it's the last line of the file.

2. Advances the File Pointer:

- o After reading a line, the file pointer moves to the beginning of the next line, ready for the subsequent call to readline().

3. Optional size Parameter:

o Usage:

- readline(size=-1) accepts an optional size argument.
- If size is provided, the method reads up to that number of bytes, which may result in reading only a part of the line if the size is smaller than the line length.
- If size is omitted or negative, readline() reads until the end of the line.

4. Returns an Empty String at EOF:

- o When the end of the file is reached, readline() returns an empty string (""), which can be used as a condition to terminate reading loops.

Basic Syntax:

```
line = file_object.readline(size=-1)
```

- `file_object`: The file object returned by `open()`.
- `size (optional)`: The maximum number of bytes to read.

Examples of Using `readline()`:

1. Reading the First Line of a File:

```
with open('example.txt', 'r') as file:  
    first_line = file.readline()  
    print(first_line)
```

- Explanation:
 - Opens `example.txt` in read mode.
 - Reads the first line and prints it.

2. Reading a File Line by Line Using a Loop:

```
with open('example.txt', 'r') as file:  
    line = file.readline()  
    while line != '':  
        print(line.strip()) # Strips newline and prints  
        line = file.readline()
```

- Explanation:
 - Reads and processes each line until an empty string is returned (end of file).
 - `line.strip()` removes any leading and trailing whitespace, including the newline character.

3. Using a for Loop Alternative:

While `readline()` can be used in a loop, iterating directly over the file object is more Pythonic and efficient:

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

- Note:
 - Iterating over the file object automatically reads one line at a time.
 - This method is preferred for its simplicity and efficiency.

4. Reading with the size Parameter:

```
with open('example.txt', 'r') as file:  
    line = file.readline(10)  
    print(line)
```

- Explanation:
 - Reads up to 10 bytes from the first line.
 - If the line is longer than 10 bytes, only a portion of it is read.

Use Cases for readline():

1. Memory-Efficient Processing:
 - Useful when working with large files that cannot be loaded entirely into memory.
 - Allows processing of each line individually.
2. Custom Line Processing:
 - When specific control over the reading process is needed.
 - For example, when you need to read lines up to a certain condition or need to manipulate the file pointer manually.
3. Partial Line Reading:
 - By specifying the size parameter, you can read parts of a line, which can be useful in certain parsing scenarios.

Important Considerations:

- Newline Characters:
 - `readline()` includes the newline character (`\n`) at the end of each line.
 - Use `line.strip()` or `line.rstrip('\n')` to remove the newline character if necessary.
- End of File Detection:
 - An empty string ("") returned by `readline()` indicates that the end of the file has been reached.
 - This can be used to terminate reading loops.
- Binary Mode:
 - When reading a file in binary mode ('rb'), `readline()` returns bytes instead of strings.
 - Lines are separated by `b'\n'` in binary mode.

```
with open('binaryfile.bin', 'rb') as file:  
    line = file.readline()  
    while line:  
        # Process the binary line  
        line = file.readline()
```

Comparison with Other Reading Methods:

- `read():`
 - Reads the entire file or a specified number of bytes.
 - Not memory-efficient for large files.
- `readlines():`
 - Reads all lines into a list.
 - Also not memory-efficient for large files.
- Iterating Over the File Object:
 - Preferred method for reading lines in a loop.

- o Example:

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

Example Application: Counting Lines in a File

```
line_count = 0  
with open('example.txt', 'r') as file:  
    while file.readline():  
        line_count += 1  
print(f"Total number of lines: {line_count}")
```

- Explanation:
 - o Reads each line using readline() and increments a counter.
 - o Useful for counting the number of lines without loading the entire file.

Summary:

- The readline() method in Python is a fundamental tool for file handling when you need to read files line by line.
- It reads one line at a time from the file, including the newline character.
- Ideal for processing large files where loading the entire file into memory is impractical.
- Provides control over the reading process, allowing for customized file parsing and processing.

4.9 Binary files (KT0409) (IAC0401)

Binary files are files that contain data in a format that is not human-readable and is intended to be interpreted by a computer program. They store data in the same format

as it is held in memory, consisting of binary digits (bits) that represent various types of information, such as images, audio files, video files, executable programs, and more. Unlike text files, which store data as sequences of characters encoded in a specific character set (like UTF-8), binary files store data as raw bytes.

Key Aspects of Binary Files:

1. Raw Byte Data:
 - o Storage Format:
 - Binary files store data as a sequence of bytes, which can represent any type of information, including non-textual data.
 - The data is stored in the exact format as it appears in memory, without any encoding or interpretation.
2. Not Human-Readable:
 - o Interpretation Required:
 - Binary files cannot be read or understood directly by humans.
 - They require specific programs or libraries to interpret and display the data meaningfully.
3. Usage in Python:
 - o Binary Modes:
 - When handling binary files in Python, files must be opened in binary mode by adding 'b' to the file mode (e.g., 'rb', 'wb').
 - This tells Python to read or write the file as binary data.

Differences Between Text Files and Binary Files:

- Text Files:
 - o Store data as a sequence of characters.
 - o Use specific encoding schemes (e.g., UTF-8, ASCII).
 - o Can be opened and read by text editors.
 - o Line endings are handled (e.g., \n on Unix/Linux, \r\n on Windows).

- Binary Files:
 - Store data as a sequence of bytes.
 - Do not use character encoding.
 - Cannot be properly displayed in text editors.
 - Line endings and encoding are not managed automatically.

Functions and Purpose of Handling Binary Files in Python:

1. Processing Non-Textual Data:

- Images and Media:
 - Reading and writing image files (e.g., JPEG, PNG), audio files (e.g., MP3, WAV), and video files.
- Executable Files:
 - Handling binary executables, libraries, or compiled code.
- Data Serialization:
 - Working with serialized data formats (e.g., pickled objects, protocol buffers).

2. Data Integrity:

- Exact Data Representation:
 - Binary mode ensures that data is read and written exactly as is, without any transformation.
- Avoids Encoding Issues:
 - Prevents issues arising from character encoding conversions that can corrupt binary data.

3. Performance Optimization:

- Efficient Data Processing:
 - Enables efficient reading and writing of large amounts of data.
- Random Access:

- Supports random access to file contents using methods like `seek()` and `tell()`.

Handling Binary Files in Python:

1. Opening a Binary File:

Use the `open()` function with a binary mode:

- Read Binary ('rb'):

```
file = open('filename', 'rb')
```

- Write Binary ('wb'):

```
file = open('filename', 'wb')
```

- Append Binary ('ab'):

```
file = open('filename', 'ab')
```

- Read and Write Binary ('rb+' or 'wb+'):

```
file = open('filename', 'rb+')
```

2. Reading from a Binary File:

- Read Entire File:

```
with open('image.jpg', 'rb') as file:  
    data = file.read()  
    # 'data' is a bytes object containing the binary data
```

- Read Fixed Number of Bytes:

```
with open('data.bin', 'rb') as file:  
    chunk = file.read(1024) # Reads 1024 bytes
```

3. Writing to a Binary File:

- Write Bytes Data:

```
with open('output.bin', 'wb') as file:  
    data = b'\x00\xFF\x00\xFF' # Example bytes data  
    file.write(data)
```

- Writing Binary Data from Variables:

```
with open('numbers.bin', 'wb') as file:  
    numbers = [10, 20, 30, 40]  
    byte_array = bytearray(numbers)  
    file.write(byte_array)
```

4. Working with Binary Data:

- Converting Data to Bytes:
 - Use the struct module to convert between Python values and C structs represented as Python bytes objects.

```
import struct  
  
# Pack integers into bytes  
data = struct.pack('iiii', 1, 2, 3, 4)
```

- Reading Binary Data into Variables:

```
import struct  
  
with open('numbers.bin', 'rb') as file:  
    data = file.read(16) # Read 16 bytes (4 integers)  
    numbers = struct.unpack('iiii', data)  
    print(numbers) # Outputs: (1, 2, 3, 4)
```

5. Copying Binary Files:

```
with open('source.bin', 'rb') as source_file:
    with open('destination.bin', 'wb') as dest_file:
        while True:
            chunk = source_file.read(4096) # Read in 4KB chunks
            if not chunk:
                break
            dest_file.write(chunk)
```

- Explanation:
 - Reads the binary file in chunks to handle large files efficiently.
 - Writes each chunk to the destination file.

Important Considerations:

1. Bytes vs. Strings:

- Bytes Objects:
 - In binary mode, data is read and written as bytes objects.
 - Bytes literals are prefixed with b, e.g., b'Hello'.
- Conversion:
 - To convert bytes to strings, specify the encoding:

```
byte_data = b'Hello, World!'
string_data = byte_data.decode('utf-8')
```

- To convert strings to bytes:

```
string_data = 'Hello, World!'
byte_data = string_data.encode('utf-8')
```

2. No Automatic Encoding/Decoding:

- Unlike text mode, binary mode does not perform any encoding or decoding of data.
- Data is read and written exactly as-is.

3. Line Endings and EOF:

- o In binary mode, there is no special handling of newline characters (\n) or end-of-file markers.
- o Line-oriented methods like readline() may not behave as expected.

4. Universal Newline Support:

- o Not available in binary mode.
- o Line endings are preserved as they are in the file.

5. Platform Differences:

- o Opening files in binary mode avoids issues with line ending translations on different operating systems (e.g., Windows vs. Unix/Linux).

Example: Reading and Writing an Image File

Reading an Image and Creating a Copy:

```
# Reading the image file in binary mode
with open('original_image.jpg', 'rb') as source_file:
    image_data = source_file.read()

# Writing the image data to a new file
with open('copy_image.jpg', 'wb') as dest_file:
    dest_file.write(image_data)

print("Image copied successfully.")
```

- Explanation:

- Opens the original image in 'rb' mode and reads the binary data.
- Writes the binary data to a new file in 'wb' mode.
- The copied image is identical to the original.

Use Case: Saving and Loading Binary Data Structures

Saving a Dictionary to a Binary File Using Pickle:

```
import pickle

data = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Writing the dictionary to a binary file
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)
```

Loading the Dictionary from the Binary File:

```
import pickle

# Reading the dictionary from the binary file
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)
```

```
print(loader_data)
```

- Explanation:
 - Uses the pickle module to serialize (convert) the dictionary into bytes and write it to a binary file.
 - Deserializes the bytes back into a Python dictionary when reading.

Best Practices:

1. Always Specify Binary Mode for Binary Files:
 - Ensure that files containing binary data are opened with a mode that includes 'b'.
2. Use Context Managers:
 - Utilize with statements to handle files, which ensures proper closure of files.
3. Handle Exceptions:
 - Implement error handling to manage exceptions like FileNotFoundError, IOError, etc.

```
try:  
    with open('data.bin', 'rb') as file:  
        data = file.read()  
except FileNotFoundError:  
    print("File not found.")
```

4. Be Cautious with Data Types:
 - Remember that in binary mode, read operations return bytes objects, and write operations require bytes objects.
 - Convert data appropriately using encode() and decode() or the struct module.

Summary:

- Binary files store data as raw bytes, suitable for non-textual data like images, audio, video, executables, and serialized objects.
- Binary mode is specified by including 'b' in the file mode (e.g., 'rb', 'wb').
- Handling binary files requires working with bytes objects rather than strings.
- Key operations include reading and writing bytes, copying files, and working with binary data structures.
- Important considerations involve proper data type conversion, avoiding automatic encoding/decoding, and ensuring platform-independent handling of data.



Formative Assessment Activity [4]

File Handling

Complete the formative activity in your **KM4 Learner Workbook**,

Knowledge Topic KM-04-KT05:

Topic Code	KM-04-KT05
Topic	Exception Handling
Weight	25%

This knowledge topic will cover the following topic elements:

- KT0501 Concept, definition, and functions
- KT0502 Syntax
- KT0503 Exceptions
- KT0504 Common examples of exception
- KT0505 Rules of exceptions
- KT0506 Exception handling
- KT0507 Keywords (try, catch, finally)
- KT0508 Important Python errors and exceptions
- KT0509 Error vs exception
- KT0510 User-defined exception

After working through this knowledge topic, your competence will be assessed according to the following criterion/criteria:

- IAC0501 Definitions, functions and features of Python exception handling are understood and explained

KM-04-KT05 Exception Handling (IAC0501)

5.1 Concept, definition and functions (KT0501) (IAC0501)

Exception handling in Python is a mechanism that allows a program to detect and respond to runtime errors, known as exceptions, in a controlled and graceful manner. Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions, such as division by zero, file not found, or invalid input. Exception handling enables a program to continue running or terminate gracefully rather than crashing abruptly.

Concept of Exception Handling in Python

1. What Are Exceptions?

- o Exceptions vs. Errors:

- Errors are serious issues that a reasonable application should not try to handle (e.g., syntax errors).
- Exceptions are conditions that occur during runtime that can be anticipated and handled (e.g., ZeroDivisionError, FileNotFoundError).

- o Exception Hierarchy:

- All exceptions are derived from the base class Exception.
- Common built-in exceptions include ValueError, TypeError, IndexError, etc.

2. The Exception Handling Mechanism:

- o try Block:

- Contains code that might raise an exception.
- If no exception occurs, the code in the try block runs normally.

- o except Block:

- Catches and handles exceptions raised in the try block.
- Specific exceptions can be caught, or a general exception handler can be used.

- o else Block:

- Executes code if no exceptions were raised in the try block.
- Useful for code that should run only when no errors occur.
- o finally Block:
 - Contains code that will always execute, regardless of whether an exception occurred.
 - Used for cleanup actions like closing files or releasing resources.

3. Raising Exceptions:

- o Use the raise statement to trigger an exception intentionally.
- o Syntax: `raise ExceptionType("Error message")`

4. Creating Custom Exceptions:

- o Custom exceptions are defined by creating a new class that inherits from `Exception`.
- o Allows for more specific error handling tailored to the application's needs.

Functions and Purpose of Exception Handling in Python

1. Graceful Error Handling:

- o Prevent Crashes:
 - Allows the program to handle unexpected situations without terminating abruptly.
- o User Feedback:
 - Provides meaningful error messages to users.

2. Resource Management:

- o Ensure Cleanup:
 - Use finally blocks or context managers to release resources (e.g., closing files).

3. Control Flow Management:

- o Alternate Execution Paths:

- Enables the program to take corrective actions or switch logic when exceptions occur.

4. Improved Code Reliability:

- Robustness:

- Anticipating potential errors makes the code more reliable and easier to maintain.

- Debugging:

- Helps identify and fix issues by catching exceptions and logging error information.

Basic Syntax of Exception Handling

1. Try-Except Block:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code to handle the exception
```

2. Try-Except-Else Block:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code to handle the exception  
else:  
    # Code to execute if no exception occurs
```

3. Try-Except-Finally Block:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code to handle the exception  
finally:  
    # Code that will always execute
```

4. Try-Except-Else-Finally Block:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Exception handling code  
else:  
    # Code that runs if no exceptions occur  
finally:  
    # Code that always runs
```

Examples of Exception Handling

Example 1: Handling Division by Zero

```
try:  
    numerator = int(input("Enter the numerator: "))  
    denominator = int(input("Enter the denominator: "))  
    result = numerator / denominator  
    print(f"The result is {result}")  
except ZeroDivisionError:  
    print("Error: Cannot divide by zero.")  
except ValueError:  
    print("Error: Invalid input. Please enter integers only.")
```

Explanation:

- try Block:
 - Attempts to perform division.
- except ZeroDivisionError:

- Catches division by zero errors.
- except ValueError:
 - Handles invalid input when converting to integers.

Example 2: Using finally for Resource Cleanup

```
try:
    file = open('data.txt', 'r')
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
finally:
    if 'file' in locals():
        file.close()
        print("File closed.")
```

Explanation:

- finally Block:
 - Ensures the file is closed whether an exception occurs or not.

Example 3: Raising an Exception

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    else:
        print(f"Your age is {age}")

try:
    age_input = int(input("Enter your age: "))
    check_age(age_input)
except ValueError as e:
    print(f"Error: {e}")
```

Explanation:

- `raise`:
 - Manually raises a `ValueError` if the age is negative.
- Exception Handling:
 - The exception is caught and handled in the `except` block.

Example 4: Custom Exception Class

```
class InsufficientFundsError(Exception):
    """Exception raised when attempting to withdraw more funds than
available."""
    pass

def withdraw(amount, balance):
    if amount > balance:
        raise InsufficientFundsError("Insufficient funds for withdrawal.")
    else:
        balance -= amount
        return balance

try:
    balance = 500
    withdrawal_amount = 600
    new_balance = withdraw(withdrawal_amount, balance)
    print(f"New balance: {new_balance}")
except InsufficientFundsError as e:
    print(f"Error: {e}")
```

Explanation:

- Custom Exception:
 - `InsufficientFundsError` is a user-defined exception.
- Usage:
 - Raised when withdrawal amount exceeds the balance.

- Handling:
 - Caught in the except block and provides a custom error message.

Important Concepts

1. Catching Multiple Exceptions:

```
try:  
    # Code that may raise multiple exceptions  
except (TypeError, ValueError) as e:  
    print(f"Error: {e}")
```

2. Accessing Exception Details:

- Using as:
 - Assigns the exception to a variable for more information.
- Example:

```
except ExceptionType as e:  
    print(f"An error occurred: {e}")
```

3. The else Clause:

- Purpose:
 - Runs code if no exceptions were raised in the try block.
- Example:

```
try:  
    result = compute_value()  
except Exception as e:  
    print(f"Error: {e}")  
else:  
    print(f"Computation successful: {result}")
```

4. The finally Clause:

- o Purpose:
 - Executes code regardless of exceptions.
- o Use Cases:
 - Releasing resources, closing files, or cleanup tasks.

5. Re-raising Exceptions:

- o Syntax:

```
except Exception as e:  
    # Perform some action  
    raise
```

- o Purpose:
 - Allows an exception to propagate after being handled or logged.

Best Practices

1. Catch Specific Exceptions:

- o Why:
 - Improves code clarity and avoids hiding bugs.

2. Avoid Bare Excepts:

- o Do Not Use:

```
except:  
    # This catches all exceptions, including system exits
```

- o Use Instead:

```
except Exception:  
    # This catches most exceptions but not system exits
```

3. Clean Up Resources:

- o Use finally or Context Managers:

- Ensures resources are released properly.
 - Example with finally:

```
try:  
    file = open('data.txt', 'r')  
    # Work with the file  
finally:  
    file.close()
```

- Example with Context Manager:

```
with open('data.txt', 'r') as file:  
    # Work with the file
```

4. Provide Meaningful Error Messages:

- Enhances User Experience:
 - Helps users understand what went wrong.
- Example:

```
except ValueError as e:  
    print(f"Invalid input: {e}")
```

5. Logging Exceptions:

- Use Logging Module:

```
import logging  
  
try:  
    # Code that may fail  
except Exception as e:  
    logging.error("An error occurred", exc_info=True)
```

6. Re-raise Exceptions When Appropriate:

- o Allows Higher-Level Handling:
 - If an exception can't be handled adequately, re-raising allows it to be managed elsewhere.

Summary

- Exception Handling is a crucial part of Python programming that allows developers to anticipate and manage runtime errors gracefully.
- Key Components:
 - o try Block: Code that might raise exceptions.
 - o except Block: Code that handles exceptions.
 - o else Block: Executes if no exceptions occur.
 - o finally Block: Executes regardless of exceptions.
- Benefits:
 - o Robust Programs: Prevents crashes and ensures resources are managed.
 - o User-Friendly: Provides clear error messages.
 - o Maintainable Code: Separates error handling from business logic.
- Best Practices:
 - o Catch specific exceptions.
 - o Avoid bare except clauses.
 - o Use finally or context managers for resource cleanup.
 - o Log exceptions for debugging.
 - o Provide informative error messages.

5.2 Syntax (KT0502) (IAC0501)

Exception handling in Python allows you to manage and respond to runtime errors gracefully. The syntax involves the use of try, except, else, and finally blocks to catch and handle exceptions, ensuring your program can continue running or exit cleanly.

Basic Syntax of Exception Handling

1. The try and except Blocks

The fundamental structure for exception handling starts with the try block followed by one or more except blocks.

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code to handle the exception
```

- try Block: Contains code that may potentially raise an exception.
- except Block: Handles the exception if it occurs.

Example:

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

2. Catching Specific Exceptions

You can specify multiple except blocks to handle different exceptions separately.

```
try:  
    # Code that might raise multiple exceptions  
except ValueError:  
    # Handle ValueError  
except TypeError:  
    # Handle TypeError
```

Or catch multiple exceptions in a single block:

```
try:  
    # Code that might raise exceptions  
except (ValueError, TypeError):  
    # Handle ValueError and TypeError
```

3. Accessing Exception Details

Use the as keyword to access the exception object.

```
try:  
    # Code that might raise an exception  
except ExceptionType as e:  
    # Use 'e' to get details about the exception
```

Example:

```
try:  
    num = int("abc")  
except ValueError as error:  
    print(f"Invalid input: {error}")
```

4. The else Block

The else block executes if no exceptions are raised in the try block.

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Handle exception  
else:  
    # Code to execute if no exception occurs
```

Example:

```
try:  
    with open('data.txt', 'r') as file:  
        content = file.read()  
except FileNotFoundError:  
    print("File not found.")  
else:  
    print(content)
```

5. The finally Block

The `finally` block executes regardless of whether an exception was raised or not. It's typically used for cleanup actions.

```
try:  
    file = open('data.txt', 'r')  
    # Perform file operations  
except FileNotFoundError:  
    print("File not found.")  
finally:  
    file.close()  
    print("File closed.")
```

Example:

```
try:  
    file = open('data.txt', 'r')  
    # Perform file operations  
    content = file.read()  
    print(content)  
except FileNotFoundError:  
    print("File not found.")  
finally:  
    file.close()  
    print("File closed.")
```

6. Raising Exceptions

You can raise exceptions using the `raise` statement.

```
raise ExceptionType("Error message")
```

Example:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print(f"Age is set to {age}")

try:
    set_age(-5)
except ValueError as e:
    print(f"Error: {e}")
```

Summary of Exception Handling Syntax

- try: Contains code that might raise an exception.
- except: Catches and handles specific exceptions.
- else: Executes if no exceptions occur in the try block.
- finally: Executes code regardless of whether an exception occurred; useful for cleanup tasks.
- raise: Manually triggers an exception.

5.3 Exceptions (KT0503) (IAC0501)

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. It is a runtime error that arises when a program encounters an unexpected situation or an invalid operation, such as dividing by zero, accessing a non-existent file, or using an incorrect data type.

Concept of Exceptions in Python

1. What Are Exceptions?

- o Runtime Errors:
 - Exceptions are errors detected during execution. They are different from syntax errors, which are detected during code parsing.
- o Exception Objects:
 - When an exception occurs, Python creates an object representing the error. This object contains information about the type of error and details about the exception.

2. Exception Hierarchy:

- o BaseException Class:
 - All exception classes in Python inherit from the built-in BaseException class.
- o Common Exception Classes:
 - Exception: The base class for most built-in exceptions.
 - Specific Exceptions:
 - ZeroDivisionError: Raised when division by zero occurs.
 - TypeError: Raised when an operation is applied to an object of inappropriate type.
 - ValueError: Raised when a function receives an argument of the correct type but an inappropriate value.
 - IndexError: Raised when a sequence subscript is out of range.
 - KeyError: Raised when a dictionary key is not found.
- o Custom Exceptions:

- Programmers can create custom exception classes by inheriting from the Exception class to handle specific errors in their programs.

3. Raising Exceptions:

- o Automatic Raising:
 - Python automatically raises exceptions when it encounters errors during execution.
- o Manual Raising:
 - Programmers can intentionally raise exceptions using the raise statement to signal that an error condition has occurred.

Functions and Purpose of Exceptions

1. Error Signalling:

- o Interrupt Execution:
 - Exceptions signal that an error has occurred, interrupting the normal flow of the program.
- o Provide Error Information:
 - Exception objects carry information about the error type and context, aiding in debugging and error handling.

2. Facilitate Exception Handling:

- o Control Flow Management:
 - Exceptions allow programs to manage errors gracefully using try, except, else, and finally blocks.
- o Prevent Program Crashes:
 - Proper exception handling prevents programs from crashing unexpectedly, enhancing user experience and program stability.

3. Promote Robust Code:

- o Encourage Defensive Programming:
 - Anticipating potential exceptions leads to more reliable and maintainable code.

- o Enhance Debugging:
 - Clear exception messages help identify and fix issues during development.

Example of an Exception

Division by Zero Example:

```
# Attempting to divide by zero
numerator = 10
denominator = 0
result = numerator / denominator # Raises ZeroDivisionError
```

- Explanation:
 - o The above code will raise a ZeroDivisionError because dividing a number by zero is mathematically undefined.

Handling Exceptions

Exceptions can be caught and managed using exception handling constructs to prevent program termination and to provide meaningful feedback.

Example: Handling Division by Zero

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print(f"The result is {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

- Explanation:
 - o The try block contains code that may raise an exception.

- The except ZeroDivisionError block catches the specific exception and handles it.

Creating Custom Exceptions

```
Defining a Custom Exception:
class NegativeNumberError(Exception):
    """Exception raised when a negative number is encountered where it is not
allowed."""
    pass

def square_root(number):
    if number < 0:
        raise NegativeNumberError("Cannot compute the square root of a
negative number.")
    return number ** 0.5

try:
    result = square_root(-9)
    print(f"The result is {result}")
except NegativeNumberError as e:
    print(f"Error: {e}")
```

- Explanation:
 - Custom Exception Class:
 - NegativeNumberError is a user-defined exception inheriting from Exception.
 - Raising the Exception:
 - The square_root function raises NegativeNumberError if the input is negative.
 - Handling the Exception:
 - The try-except block catches the custom exception and prints an error message.

Importance of Exceptions

1. Error Detection and Handling:

- o Immediate Notification:
 - Exceptions alert the programmer or user that something went wrong.
- o Controlled Response:
 - Allows the program to respond appropriately, such as retrying an operation or providing user instructions.

2. Maintaining Program Flow:

- o Preventing Crashes:
 - By handling exceptions, programs can avoid unexpected termination.
- o Alternative Execution Paths:
 - Enables the implementation of fallback mechanisms or default behaviours when errors occur.

3. Debugging and Maintenance:

- o Detailed Error Information:
 - Exceptions provide stack traces and error messages that are invaluable for debugging.
- o Cleaner Code Structure:
 - Separates error-handling code from regular code, making programs easier to read and maintain.

4. User Experience:

- o Informative Feedback:
 - Users receive clear messages about what went wrong, improving usability.
- o Data Integrity:
 - Proper exception handling ensures that resources are released, and data remains consistent even when errors occur.

Summary:

Exceptions in Python are a critical component of robust programming. They represent errors that occur during execution and provide a mechanism to detect, signal, and handle these errors gracefully. By understanding exceptions and implementing proper exception handling, developers can create programs that are more reliable, maintainable, and user-friendly.

Key Takeaways:

- Exceptions signal runtime errors that disrupt normal execution.
- Exception Handling uses try, except, else, and finally blocks to manage errors.
- Built-in Exceptions cover common error types; custom exceptions can handle specific cases.
- Proper Handling enhances program stability and user experience.

5.4 Common examples of exception (KT0504) (IAC0501)

Exceptions are errors that occur during the execution of a program, disrupting its normal flow. Recognizing common exceptions is essential for writing robust code and handling errors gracefully. Below are some of the most frequently encountered built-in exceptions in Python, along with explanations and examples.

1. ZeroDivisionError

- Description: Raised when attempting to divide a number by zero.
- Common Scenario:

```
numerator = 10
denominator = 0
result = numerator / denominator # Raises ZeroDivisionError
```

- Handling Example:

```
try:
    result = numerator / denominator
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

2. TypeError

- Description: Occurs when an operation is applied to an object of inappropriate type.
- Common Scenario:

```
# This will raise a TypeError
result = '10' + 5 # Raises TypeError
```

- Handling Example:

```
try:
```

```
result = '10' + 5
except TypeError:
    print("Error: Cannot add a string and an integer.")
```

3. ValueError

- Description: Raised when a function receives an argument of the correct type but an inappropriate value.
- Common Scenario:

```
number = int('abc') # Raises ValueError
```

- Handling Example:

```
try:
    number = int('abc')
except ValueError:
    print("Error: Invalid literal for integer conversion.")
```

4. IndexError

- Description: Occurs when trying to access an index that is out of range in a sequence (like a list or tuple).
- Common Scenario:

```
my_list = [1, 2, 3]
item = my_list[5] # Raises IndexError
```

- Handling Example:

```
try:
    item = my_list[5]
except IndexError:
    print("Error: List index out of range.")
```


5. KeyError

- Description: Raised when a dictionary key is not found.
- Common Scenario:

```
my_dict = {'name': 'Alice', 'age': 30}  
value = my_dict['address'] # Raises KeyError
```

- Handling Example:

```
try:  
    value = my_dict['address']  
except KeyError:  
    print("Error: Key 'address' not found in dictionary.")
```

6. AttributeError

- Description: Occurs when an attribute reference or assignment fails.
- Common Scenario:

```
my_list = [1, 2, 3]  
my_list.push(4) # Raises AttributeError
```

- Handling Example:

```
try:  
    my_list.push(4)  
except AttributeError:  
    print("Error: 'list' object has no attribute 'push'.")
```

7. FileNotFoundError

- Description: Raised when a file or directory is requested but doesn't exist.
- Common Scenario:

```
with open('nonexistent_file.txt', 'r') as file:  
    content = file.read() # Raises FileNotFoundError
```

- Handling Example:

```
try:  
    with open('nonexistent_file.txt', 'r') as file:  
        content = file.read()  
except FileNotFoundError:  
    print("Error: The file was not found.")
```

8. ImportError and ModuleNotFoundError

- Description:
 - ImportError: Raised when an import statement fails.
 - ModuleNotFoundError: A subclass of ImportError raised when a module cannot be found.
- Common Scenario:

```
import nonexistent_module # Raises ModuleNotFoundError
```

- Handling Example:

```
try:  
    import nonexistent_module  
except ModuleNotFoundError:  
    print("Error: Module not found.")
```

9. NameError

- Description: Occurs when a local or global name is not found.
- Common Scenario:

```
print(value) # Raises NameError if 'value' is not defined
```

- Handling Example:

```
try:  
    print(value)  
except NameError:  
    print("Error: Variable 'value' is not defined.")
```

10. OverflowError

- Description: Raised when the result of an arithmetic operation is too large to be represented.
- C

```
try:  
    print(value)  
except NameError:  
    print("Error: Variable 'value' is not defined.")
```

- Handling Example:

```
import math
```

```
try:  
    result = math.exp(1000)  
except OverflowError:  
    print("Error: Numerical result out of range.")
```

11. MemoryError

- Description: Occurs when an operation runs out of memory.
- Common Scenario:

```
large_list = [1] * (10**10) # May raise MemoryError
```

12. RuntimeError

- Description: Raised when an error is detected that doesn't fall into any other category.
- Common Scenario:

```
def recursive_function():
    return recursive_function()
recursive_function() # May raise RuntimeError due to maximum recursion depth
```

- Handling Example:

```
try:
    recursive_function()
except RuntimeError as e:
    print(f"Error: {e}")
```

13. StopIteration

- Description: Raised to signal the end of an iterator.
- Common Scenario:

```
iterator = iter([1, 2, 3])
while True:
    item = next(iterator) # Raises StopIteration after last item
```

- Handling Example:

```
try:
    while True:
        item = next(iterator)
        print(item)
except StopIteration:
    print("End of iterator reached.")
```

14. AssertionError

- Description: Raised when an assert statement fails.
- Common Scenario:

```
x = -1
assert x >= 0, "x must be non-negative" # Raises AssertionError
```

- Handling Example:

```
try:
```

```
    assert x >= 0, "x must be non-negative"
except AssertionError as e:
    print(f"Assertion failed: {e}")
```

15. KeyboardInterrupt

- Description: Raised when the user interrupts program execution, usually by pressing Ctrl+C.
- Common Scenario:

```
try:  
    while True:  
        pass # Infinite Loop  
except KeyboardInterrupt:  
    print("Program interrupted by user.")
```

- Handling Example:

```
try:  
    while True:  
        # Long-running process  
        pass  
except KeyboardInterrupt:  
    print("Program interrupted by user.")
```

Understanding these common exceptions enables you to anticipate potential errors in your programs and handle them effectively. By catching and managing exceptions, you can improve the robustness of your code, provide meaningful feedback to users, and prevent unexpected crashes.

5.5 Rules of exceptions (KT0505) (IAC0501)

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions. Exception handling allows programmers to manage these errors gracefully, ensuring that the program can handle unexpected situations without crashing. Understanding the rules of exceptions is crucial for writing robust and maintainable code.

Key Rules of Exceptions in Python

1. Exceptions Are Objects Derived from the BaseException Class

- o Hierarchy:

- All exceptions in Python inherit from the BaseException class.
- The standard exceptions are organised into a hierarchy, allowing for organised and specific exception handling.

- o Custom Exceptions:

- You can create custom exceptions by subclassing Exception or any of its subclasses.

2. Raising Exceptions

- o Automatic Raising:

- Python raises exceptions automatically when an error occurs during execution.
- Example: Division by zero raises a ZeroDivisionError.

- o Manual Raising:

- Use the raise statement to raise an exception explicitly.
- Syntax:

```
raise ExceptionType("Error message")
```

- Example:

```
if age < 0:
```

```
raise ValueError("Age cannot be negative.")
```

3. Catching Exceptions with Try-Except Blocks

- o Structure:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code to handle the exception
```

- o Rules:

- The except block catches exceptions raised in the try block.
- If an exception occurs, Python looks for an except block that matches the exception type.
- If no matching except block is found, the exception propagates up the call stack.

4. Exception Propagation (Stack Unwinding)

- o Propagation:

- If an exception is not caught in the current function, it propagates to the caller.
- This process continues until the exception is caught or the program terminates.

- o Uncaught Exceptions:

- If an exception reaches the top-level of the program without being caught, it results in a traceback, and the program exits.

5. Order of Except Blocks Matters

- o Specific to General:

- Place more specific exceptions before general ones.
- Example:

```
try:
    # Code that may raise exceptions
except ValueError:
    # Handle ValueError
except Exception:
    # Handle any other exceptions
```

- Rule:
 - If a general exception handler (like except Exception:) is placed before a specific one, the specific handler will never be executed because the general handler will catch all exceptions.

6. The Else Clause Executes Only If No Exceptions Occur

- Rule:
 - The else block is executed only if the try block does not raise an exception.

7. The Finally Clause Always Executes

- Structure:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Handle exception
finally:
    # Code that always executes
```

- Rule:
 - The finally block is executed whether an exception occurred or not.
 - Used for cleanup actions like closing files or releasing resources.

8. Re-Raising Exceptions

- o Rule:

- You can re-raise an exception using raise without specifying the exception again.
- Example:

```
try:  
    # Code that may raise an exception  
except Exception as e:  
    # Perform some handling  
    raise # Re-raises the caught exception
```

9. Multiple Exceptions Can Be Handled in a Single Except Block

- o Syntax:

```
except (ExceptionType1, ExceptionType2) as e:  
    # Handle exceptions of type ExceptionType1 and ExceptionType2
```

- o Rule:

- Group exceptions in a tuple to handle them with the same block of code.

10. Custom Exceptions Should Inherit from Exception

- o Rule:

- When creating custom exceptions, they should inherit from the built-in Exception class or one of its subclasses.

- o Example:

```
class CustomError(Exception):  
    pass
```

11. Avoid Catching BaseException Directly

- o Rule:
 - Do not catch BaseException or use a bare except: clause, as it also catches system-exiting exceptions like SystemExit, KeyboardInterrupt, and GeneratorExit.
- o Preferred Practice:
 - Catch Exception instead to avoid intercepting system-level exceptions.

12. Cleanup Actions Should Be Placed in Finally or Use Context Managers

- o Rule:
 - Use finally blocks or context managers (with statements) to ensure resources are released properly.
- o Example with Finally:

```
try:  
    file = open('data.txt', 'r')  
    # Work with the file  
except IOError:  
    print("Error opening file.")  
finally:  
    file.close()
```

- o Example with Context Manager:

```
with open('data.txt', 'r') as file:  
    # Work with the file  
    # File is automatically closed when exiting the block
```

13. Exceptions Can Be Chained

- o Rule:
 - Use the from keyword to chain exceptions, indicating that one exception was directly caused by another.
- o Example:

```
try:  
    # Some code  
except Exception as e:  
    raise AnotherException("An error occurred") from e
```

14. Assertions Use the Assert Statement

- o Rule:
 - The assert statement tests a condition and raises an AssertionError if the condition is false.
- o Example:

```
assert x >= 0, "x must be non-negative"
```

15. Avoid Swallowing Exceptions Unintentionally

- o Rule:
 - Ensure that exception handling does not hide errors silently.
- o Example of Bad Practice:

```
try:  
    # Code that may fail  
except Exception:  
    pass # Swallows all exceptions without handling
```

- o Preferred Practice:
 - Handle exceptions appropriately or re-raise them after logging or performing necessary actions.

Best Practices Based on the Rules

- Be Specific in Exception Handling:

- Catch specific exceptions to avoid handling unintended exceptions.
- Maintain the Order of Except Blocks:
 - Place more specific exception handlers before more general ones.
- Use Finally for Cleanup:
 - Place code that must run regardless of exceptions in the finally block.
- Leverage Else for Code that Should Run When No Exception Occurs:
 - Use the else block for code that should only execute if the try block succeeds.
- Create Meaningful Custom Exceptions:
 - Define custom exceptions to handle application-specific errors.
- Avoid Bare Except Clauses:
 - Do not use except: without specifying an exception type.
- Use Context Managers When Possible:
 - Utilize with statements to manage resources automatically.
- Do Not Suppress Exceptions Silently:
 - Always handle exceptions in a way that provides feedback or logs the error.
- Re-Raise Exceptions When Necessary:
 - If an exception cannot be handled properly, re-raise it to be handled at a higher level.

Understanding the rules of exceptions in Python is essential for writing robust and error-resistant programs. By following these rules, you can effectively manage errors, ensure resources are properly handled, and provide a better experience for users and developers alike. Exception handling is not just about preventing crashes but about anticipating potential issues and responding to them in a controlled and meaningful way.

Key Takeaways:

- Exceptions are integral to Python's error handling mechanism.
- Properly structured try, except, else, and finally blocks allow for graceful error management.
- The order and specificity of exception handlers are crucial.
- Resource management and cleanup should be handled carefully to prevent resource leaks.
- Custom exceptions enhance the clarity and specificity of error handling.

5.6 Exception handling (KT0506) (IAC0501)

Exception handling in Python is a programming practice that manages errors or unexpected events (called exceptions) that occur during the execution of a program. Instead of allowing the program to crash upon encountering an error, exception handling enables the program to respond gracefully, either by correcting the issue, informing the user, or safely terminating the program.

Key Concepts of Exception Handling

1. Exceptions:

- o What Are They?
 - Exceptions are runtime errors that disrupt the normal flow of a program.
 - Common exceptions include ZeroDivisionError, ValueError, TypeError, and FileNotFoundError.

2. Try-Except Blocks:

- o Purpose:
 - Used to catch and handle exceptions.
- o Syntax:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code to handle the exception
```

o Example:

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

3. Else Clause:

- o Purpose:

- Executes code if no exceptions occur in the try block.

- o Syntax:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Exception handling code  
else:  
    # Code to execute if no exceptions occur
```

4. Finally Clause:

- o Purpose:

- Contains code that should always execute, regardless of whether an exception was raised.

- o Syntax:

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Exception handling code  
finally:  
    # Code that always executes
```

5. Raising Exceptions:

- o Purpose:

- Manually trigger an exception using the raise keyword.

- o Syntax:

```
if condition_not_met:  
    raise ExceptionType("Error message")
```

- o Example:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
```

6. Custom Exceptions:

- o Purpose:
 - Define user-specific exceptions for better error handling.
- o Syntax:

```
class CustomError(Exception):
    pass
```

Functions and Benefits of Exception Handling

- Graceful Error Management:
 - o Prevents abrupt program termination.
 - o Allows the program to continue running or terminate cleanly.
- Resource Management:
 - o Ensures resources like files or network connections are properly closed using finally or context managers.
- Debugging Aid:
 - o Provides detailed error messages and stack traces.
 - o Helps in identifying and fixing issues efficiently.
- Improved User Experience:
 - o Offers informative feedback to users.
 - o Helps users understand what went wrong and how to fix it.

Example: Handling Multiple Exceptions

```
try:
    number = int(input("Enter an integer: "))
    result = 100 / number
    print(f"The result is {result}")
except ValueError:
    print("Error: Invalid input. Please enter an integer.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print("Operation completed successfully.")
finally:
    print("Thank you for using the program.")
```

Explanation:

- try Block:
 - Attempts to convert user input to an integer and perform division.
- except Blocks:
 - Catches ValueError if the input is not an integer.
 - Catches ZeroDivisionError if the user enters zero.
- else Block:
 - Executes if no exceptions occur.
- finally Block:
 - Executes regardless of whether an exception was raised.

Exception handling in Python is essential for creating robust and reliable programs. By anticipating potential errors and implementing appropriate handling mechanisms, developers can:

- Prevent Crashes: Ensure the program doesn't terminate unexpectedly.
- Maintain Data Integrity: Protect data by handling exceptions that could corrupt it.
- Enhance Readability and Maintenance: Keep error-handling code separate from regular code.

- Provide Better User Feedback: Inform users about issues in a clear and user-friendly manner.

Key Takeaways:

- Use try and except blocks to catch and handle exceptions.
- Implement else and finally clauses for additional control over program flow.
- Raise exceptions intentionally with raise when necessary.
- Create custom exceptions for specific error handling.
- Always aim for clear and informative error messages to aid debugging and user understanding.

5.7 Keywords (try, catch, finally) (KT0507) (IAC0501)

The primary keywords used for exception handling in Python are try, except, else, and finally. It seems there might be a slight confusion with the keyword catch, which is used in languages like Java and C++. In Python, we use except instead of catch.

Keywords Used in Python Exception Handling

1. try

- o Purpose:

- The try block contains code that might raise an exception.
 - It allows you to test a block of code for errors.

- o Syntax:

```
try:  
    # Code that may raise an exception
```

- o Example:

```
try:  
    result = 10 / divisor
```

2. except

- o Purpose:

- The except block catches and handles exceptions that occur in the try block.
 - You can specify the type of exception to catch.

- o Syntax:

```
except ExceptionType:  
    # Code to handle the exception
```

- o Example:

```
except ZeroDivisionError:  
    print("Error: Division by zero is not allowed.")
```

- o Note:

- If you want to catch all exceptions, you can use:

```
except:  
    # Handle any exception
```

- However, it's good practice to catch specific exceptions.

3. else

- o Purpose:

- The else block executes if no exceptions were raised in the try block.
 - It's useful for code that should run only when the try block succeeds.

- o Syntax:

```
else:  
    # Code to execute if no exception occurs
```

- o Example:

```
else:  
    print("Division successful. The result is:", result)
```

4. finally

- o Purpose:

- The finally block contains code that will run no matter what, whether an exception occurred or not.
 - It's typically used for cleanup actions, such as closing files or releasing resources.
- Syntax:

```
finally:  
    # Code that always executes
```

- Example:

```
finally:  
    print("Execution completed.")
```

How These Keywords Work Together

The typical structure of exception handling in Python using these keywords is as follows:

```
try:  
    # Code that may raise an exception  
except ExceptionType1:  
    # Code to handle ExceptionType1  
except ExceptionType2:  
    # Code to handle ExceptionType2  
else:  
    # Code that runs if no exceptions occur  
finally:  
    # Code that runs no matter what
```

Detailed Explanation with Example

Scenario: You want to divide two numbers input by the user and handle potential errors such as division by zero or invalid input.

```
try:  
    numerator = float(input("Enter the numerator: "))  
    denominator = float(input("Enter the denominator: "))
```

```

        result = numerator / denominator
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter numeric values.")
else:
    print(f"The result is {result}")
finally:
    print("Program execution has ended.")

```

Explanation:

- try Block:
 - Attempts to execute code that may raise an exception (e.g., division operation, user input conversion).
- except ZeroDivisionError:
 - Catches a division by zero error and provides an appropriate message.
- except ValueError:
 - Catches errors resulting from invalid input (e.g., entering letters instead of numbers).
- else Block:
 - Executes if the try block runs without raising any exceptions, displaying the result.
- finally Block:
 - Executes regardless of whether an exception was raised, indicating the end of the program.

Key Points to Remember

- No catch Keyword in Python:
 - Unlike some other programming languages (like Java or C++), Python uses except instead of catch to handle exceptions.
- Order of except Blocks:

- When specifying multiple except blocks, place more specific exceptions before more general ones to ensure proper handling.
- The finally Block Always Executes:
 - Use the finally block to perform actions that must occur whether an exception is raised or not, such as closing files or releasing resources.
- Catching Multiple Exceptions:
 - You can catch multiple exceptions in a single except block by using a tuple:

```
except (TypeError, ValueError) as e:  
    print(f"An error occurred: {e}")
```

Summary

- try: Contains code that might raise an exception.
- except: Handles the exception if one is raised.
- else: Executes code if the try block does not raise an exception.
- finally: Executes code regardless of whether an exception occurred.

Understanding and using these exception handling keywords effectively allows you to write programs that can deal with errors gracefully, providing a better user experience and making your code more robust and maintainable.

Additional Tip:

Always aim to catch specific exceptions rather than using a bare except: clause. This practice helps in debugging and ensures that unexpected exceptions don't pass silently, potentially causing bugs that are hard to trace.

5.8 Important Python errors and exceptions (KT0508) (IAC0501)

Errors and exceptions are events that disrupt the normal flow of a program's execution. Understanding these exceptions is crucial for writing robust code that can handle unexpected situations gracefully. By anticipating common errors and handling them appropriately, you can prevent your programs from crashing and provide meaningful feedback to users.

Difference Between Errors and Exceptions

- Errors:
 - Syntax Errors (SyntaxError): Occur when the parser detects an incorrect statement. These are detected before the program execution.
 - Examples: Missing colons, unmatched parentheses, incorrect indentation.
- Exceptions:
 - Occur during program execution and can be handled using exception handling constructs (try, except).
 - Examples: Division by zero, accessing a non-existent list index, invalid type operations.

Common Built-in Exceptions in Python

Below are some of the most important Python exceptions that you might encounter, along with explanations and examples.

1. SyntaxError

- Description: Raised when the parser encounters a syntax error.
- When It Occurs:
 - Missing colons (:) after control statements.
 - Incorrect indentation.
 - Mismatched parentheses or brackets.

- Example:

```
if x > 10
    print("x is greater than 10")
# Missing colon after 'if x > 10' causes SyntaxError
```

2. IndentationError

- Description: A subclass of SyntaxError, raised when there's incorrect indentation.
- When It Occurs:
 - Inconsistent use of tabs and spaces.
 - Missing indentation in code blocks.
- Example:

```
def my_function():
print("Hello") # IndentationError due to missing indentation
```

3. TypeError

- Description: Raised when an operation is applied to an object of inappropriate type.
- When It Occurs:
 - Adding a string and an integer.
 - Calling a function with the wrong number of arguments.
- Example:

```
result = 'Hello' + 5 # TypeError: can't concatenate str and int
```

4. ValueError

- Description: Raised when a function receives an argument of the correct type but inappropriate value.
- When It Occurs:
 - Converting a non-numeric string to an integer.
 - Passing an invalid value to a function.
- Example:

```
number = int("abc") # ValueError: invalid literal for int() with base 10
```

5. IndexError

- Description: Raised when a sequence subscript is out of range.
- When It Occurs:
 - Accessing an index that doesn't exist in a list or tuple.
- Example:

```
my_list = [1, 2, 3]
item = my_list[5] # IndexError: list index out of range
```

6. KeyError

- Description: Raised when a dictionary key is not found.
- When It Occurs:
 - Accessing a non-existent key in a dictionary.
- Example:

```
my_dict = {'name': 'Alice'}
age = my_dict['age'] # KeyError: 'age'
```

7. AttributeError

- Description: Raised when an attribute reference or assignment fails.
- When It Occurs:
 - Trying to access or assign an attribute that doesn't exist.
- Example:

```
class Person:  
    pass  
  
p = Person()  
p.name # AttributeError: 'Person' object has no attribute 'name'
```

8. NameError

- Description: Raised when a local or global name is not found.
- When It Occurs:
 - Using a variable or function name that hasn't been defined.
- Example:

```
print(value) # NameError: name 'value' is not defined
```

9. ZeroDivisionError

- Description: Raised when division or modulo operation is performed with zero as the divisor.
- When It Occurs:
 - Dividing a number by zero.

- Example:

```
result = 10 / 0 # ZeroDivisionError: division by zero
```

10. FileNotFoundError

- Description: Raised when a file or directory is requested but doesn't exist.
- When It Occurs:
 - Trying to open a file that doesn't exist.
- Example:

```
with open('nonexistent.txt', 'r') as file:  
    content = file.read() # FileNotFoundError: [Errno 2] No such file or  
    directory
```

11. IOError (Python 2) / OSError (Python 3)

- Description: Raised when an input/output operation fails.
- When It Occurs:
 - Problems with file operations like reading or writing.
- Example:

```
with open('/root/secret.txt', 'r') as file:  
    content = file.read() # OSError: [Errno 13] Permission denied
```

12. ImportError and ModuleNotFoundError

- Description:
 - ImportError: Raised when an import statement fails.
 - ModuleNotFoundError: A subclass of ImportError for when the module cannot be found.
- When It Occurs:
 - Importing a module that doesn't exist or cannot be found.
- Example:

```
import nonexistent_module # ModuleNotFoundError: No module named  
'nonexistent_module'
```

13. RuntimeError

- Description: Raised when an error is detected that doesn't fall into any other category.
- When It Occurs:
 - Typically used for errors that don't have a specific exception type.
- Example:

```
def recursive_function():  
    return recursive_function()  
  
recursive_function() # RuntimeError: maximum recursion depth exceeded
```

14. StopIteration

- Description: Raised to signal the end of an iterator.
- When It Occurs:
 - When next() is called on an iterator that has no more items.

- Example:

```
iterator = iter([1, 2, 3])
while True:
    item = next(iterator) # Raises StopIteration when items are exhausted
```

15. AssertionError

- Description: Raised when an assert statement fails.
- When It Occurs:
 - Using assert to check conditions during development.
- Example:

```
x = -1
assert x >= 0, "x must be non-negative" # AssertionError: x must be
non-negative
```

16. KeyboardInterrupt

- Description: Raised when the user interrupts program execution, usually by pressing Ctrl+C.
- When It Occurs:
 - During long-running processes that the user decides to interrupt.
- Example:

```
try:
    while True:
        pass # Infinite Loop
except KeyboardInterrupt:
    print("Program interrupted by user.")
```

17. MemoryError

- Description: Raised when an operation runs out of memory.
- When It Occurs:
 - Creating excessively large data structures.
- Example:

```
large_list = [1] * (10**10) # MemoryError
```

18. OverflowError

- Description: Raised when the result of an arithmetic operation is too large to be represented.
- When It Occurs:
 - Exceeding the maximum limit for numerical types.
- Example:

```
import math

result = math.exp(1000) # OverflowError: math range error
```

19. NotImplementedError

- Description: Raised when an abstract method that needs to be implemented in a subclass is not overridden.
- When It Occurs:
 - Defining an interface or base class that requires subclasses to implement certain methods.

- Example:

```
class BaseClass:  
    def method(self):  
        raise NotImplementedError("Subclasses must implement this method")  
  
class SubClass(BaseClass):  
    pass  
  
obj = SubClass()  
obj.method() # NotImplementedError: Subclasses must implement this method
```

20. EOFError

- Description: Raised when the input() function hits an end-of-file condition (EOF) without reading any data.
- When It Occurs:
 - Typically in interactive programs waiting for user input.
- Example:

```
try:  
    data = input("Enter something: ")  
except EOFError:  
    print("EOFError: No input provided.")
```

Handling Exceptions

To handle exceptions and prevent program crashes, you can use try, except, else, and finally blocks.

General Syntax:

```
try:  
    # Code that may raise an exception  
except ExceptionType:  
    # Code to handle the exception  
else:  
    # Code to execute if no exceptions occur  
finally:  
    # Code that will always execute
```

Example: Handling Multiple Exceptions

```
# Example: Handling Multiple Exceptions
```

```
try:
    numerator = int(input("Enter numerator: "))
    denominator = int(input("Enter denominator: "))
    result = numerator / denominator
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Please enter valid integers.")
else:
    print(f"The result is {result}")
finally:
    print("Calculation attempt complete.")
```

Best Practices

- Catch Specific Exceptions: Always try to catch specific exceptions rather than using a bare except: clause.
- Provide Meaningful Messages: Offer clear and informative error messages to help users understand what went wrong.
- Avoid Suppressing Exceptions: Do not use empty except blocks; handle exceptions appropriately or let them propagate.
- Use Finally for Cleanup: Use the finally block or context managers (with statement) to ensure resources are properly released.

Understanding and handling these important Python errors and exceptions is vital for developing robust applications. By anticipating potential issues and implementing proper exception handling, you can enhance the reliability of your programs and provide a better experience for users.

Key Takeaways:

- Exceptions are runtime errors that can be handled to prevent program crashes.
- Common exceptions include `TypeError`, `ValueError`, `IndexError`, and `FileNotFoundException`.
- Use exception handling constructs like `try`, `except`, `else`, and `finally` to manage errors.
- Write clean and informative error messages to aid in debugging and user comprehension.

5.9 Error vs exception (KT0509) (IAC0501)

Errors and exceptions are terms often used interchangeably, but they have distinct meanings and roles within the language's error-handling framework. Understanding the difference between errors and exceptions is crucial for writing robust code that can gracefully handle unexpected situations.

Errors in Python

1. Definition:

- o Errors are problems that occur during the compilation or parsing phase of the code, before the program is executed.
- o They represent issues with the syntax or structure of the code that prevent Python from interpreting it.

2. Types of Errors:

- o Syntax Errors (SyntaxError):
 - Occur when the parser encounters code that doesn't conform to Python's syntax rules.
 - Examples include missing colons, unmatched parentheses, or incorrect indentation.
- o Indentation Errors (IndentationError):
 - A subclass of SyntaxError, raised when the indentation is not consistent.
- o These errors must be corrected before the code can be executed.

3. Characteristics:

- o Detected at Compile Time:
 - Errors are identified during the parsing stage, not at runtime.
- o Non-Recoverable at Runtime:
 - Since the code cannot be parsed, there's no way to handle these errors using exception handling constructs (try, except).

4. Example of a Syntax Error:

```
def greet()
    print("Hello, world!")
# Missing colon after 'greet()' causes SyntaxError
```

- o Error Message (arduino):

```
File "script.py", line 1
def greet()
^
```

SyntaxError: invalid syntax

Exceptions in Python

1. Definition:

- o Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions.
- o They represent runtime errors caused by improper operations or unforeseen circumstances.

2. Types of Exceptions:

- o Built-in Exceptions:
 - ZeroDivisionError: Division by zero.
 - TypeError: Operation applied to an object of inappropriate type.
 - ValueError: Function receives an argument of correct type but inappropriate value.
 - IndexError: Sequence index out of range.
 - KeyError: Dictionary key not found.
- o User-Defined Exceptions:
 - Custom exceptions created by inheriting from the Exception class.

3. Characteristics:

- o Detected at Runtime:
 - Exceptions occur during the execution phase.
- o Recoverable:
 - Can be caught and handled using exception handling constructs (try, except, else, finally).
- o Hierarchy:
 - All exceptions inherit from the BaseException class.

4. Example of an Exception:

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

- o Output:

```
Cannot divide by zero.
```

Key Differences Between Errors and Exceptions

Aspect	Errors	Exceptions
Occurrence Phase	Compilation/Parsing Time	Runtime
Recoverability	Non-Recoverable at Runtime	Recoverable using exception handling
Detection	Detected by the Python parser	Detected during program execution
Handling Mechanism	Must fix code to resolve	Handled using try, except blocks
Examples	SyntaxError, IndentationError	ZeroDivisionError, TypeError, etc.

Why the Distinction Matters

1. Error Correction vs. Exception Handling:

- o Errors Require Code Modification:
 - Since errors prevent the code from running, you must correct them by modifying the code.
- o Exceptions Can Be Handled at Runtime:
 - Exceptions allow the program to handle unexpected situations without crashing.

2. Development and Debugging:

- o Errors Indicate Flaws in Code Syntax:
 - They often result from typos or misunderstandings of Python's syntax.
- o Exceptions Indicate Logical Issues or External Problems:
 - Such as invalid user input, unavailable resources, or logical errors in code.

3. Program Stability:

- o Handling Exceptions Enhances Robustness:
 - By anticipating exceptions, you can make your program more resilient.
- o Errors Must Be Eliminated for Execution:
 - A program with syntax errors cannot run until those errors are fixed.

Practical Implications in Exception Handling

1. Cannot Catch Errors with Try-Except:

- o Syntax Errors Cannot Be Handled:
 - Since the code doesn't execute, try-except blocks can't catch syntax errors.
- o Example:

```
try:  
    def func()  
        pass  
except SyntaxError:  
    print("Caught a syntax error.")
```

This code will raise a **SyntaxError at compile-time** because `def func()` is missing a colon (`:`). You cannot catch syntax errors using `try-except` in the same code block where the syntax error occurs. Python must parse the entire code before execution.

2. Can Catch Exceptions to Prevent Crashes:

- o Using Exception Handling Constructs:

```
try:  
    num = int(input("Enter a number: "))  
    result = 100 / num  
except ValueError:  
    print("Invalid input. Please enter a numeric value.")  
except ZeroDivisionError:  
    print("Cannot divide by zero.")  
else:  
    print(f"The result is {result}")
```

- o Explanation:

- Anticipate potential exceptions and handle them to maintain program flow.

Summary

- Errors:
 - o Occur at compile time due to incorrect syntax.
 - o Must be fixed by correcting the code.
 - o Examples: SyntaxError, IndentationError.

- Cannot be handled at runtime using exception handling constructs.
- Exceptions:
 - Occur at runtime due to improper operations or unforeseen circumstances.
 - Can be handled using try, except, else, and finally blocks.
 - Allow the program to continue running or exit gracefully.
 - Examples: ZeroDivisionError, TypeError, FileNotFoundError.

While errors indicate problems that prevent a program from starting, exceptions are runtime events that can be managed and handled gracefully. By writing code that anticipates and handles exceptions, you create robust programs that provide better user experiences and are easier to maintain.

Key Takeaways:

- Errors must be fixed before execution; exceptions can be managed during execution.
- Use exception handling constructs to catch and handle exceptions, not errors.
- Proper handling of exceptions leads to more resilient and user-friendly applications.

5.10 User-defined exception (KT0510) (IAC0501)

User-defined exceptions are custom exceptions created by programmers to represent specific error conditions that are not covered by the built-in exceptions. By defining your own exceptions, you can provide more meaningful error messages and handle unique error situations in your programs, making your code more robust and easier to maintain.

Concept of User-Defined Exceptions

1. Why Create User-Defined Exceptions?
 - Specific Error Handling:

- Built-in exceptions may not adequately describe all error conditions that can occur in your application.
 - Custom exceptions allow you to represent application-specific errors clearly.
- Improved Code Readability:
 - Custom exceptions make it easier to understand what went wrong when an error occurs.
 - They provide a clear indication of the type of error, aiding in debugging and maintenance.
 - Modularity and Reusability:
 - Custom exceptions can be organised in a hierarchy, promoting modular code design.
 - They can be reused across different parts of an application or in multiple projects.

2. How User-Defined Exceptions Work:

- Inheritance from Exception Class:
 - User-defined exceptions are created by defining a new class that inherits from Python's built-in `Exception` class or one of its subclasses.
 - This ensures that your custom exception behaves like a standard exception and can be caught using exception handling constructs.
- Exception Hierarchy:
 - You can create a hierarchy of custom exceptions by having them inherit from each other.
 - This allows for more granular exception handling and better organisation.

Creating User-Defined Exceptions

1. Defining a Basic User-Defined Exception

You can create a custom exception by defining a new class that inherits from the `Exception`

```
class MyCustomError(Exception):
    """Exception raised for custom error conditions."""
    pass
```

- Explanation:
 - class MyCustomError(Exception): defines a new exception class.
 - The pass statement indicates an empty class body.
 - The docstring provides a description of the exception.

2. Adding Attributes to Custom Exceptions

You can add attributes to your exception class to store additional information about the error.

```
class InsufficientBalanceError(Exception):
    """Exception raised when a withdrawal exceeds the account balance."""

    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Insufficient balance. Available: ${balance}, Attempted withdrawal: ${amount}")
```

- Explanation:
 - The __init__ method initializes the exception with specific details.
 - self.balance and self.amount store relevant data.
 - super().__init__() calls the base class constructor with a custom error message.

Using User-Defined Exceptions

Example Scenario: Banking Application

Suppose you're developing a banking application that needs to handle overdraft situations when a withdrawal amount exceeds the available balance.

Step 1: Define Custom Exceptions

```
class InsufficientBalanceError(Exception):
    """Raised when an attempt is made to withdraw more than the available
balance."""

    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(
            f"Insufficient balance: Available ${balance}, Requested
${amount}"
        )
```

Step 2: Implement Business Logic

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

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientBalanceError(self.balance, amount)
        self.balance -= amount
        return self.balance
```

Step 3: Handle the Custom Exception

```
def main():
    account = BankAccount(balance=500)

    try:
        amount_to_withdraw = float(input("Enter amount to withdraw: "))
        new_balance = account.withdraw(amount_to_withdraw)
```

```

        print(f"Withdrawal successful. New balance: ${new_balance}")
    except InsufficientBalanceError as e:
        print(f"Error: {e}")
    except ValueError:
        print("Error: Invalid input. Please enter a numeric value.")

if __name__ == "__main__":
    main()

```

- Explanation:

- The BankAccount class includes a withdraw method that raises InsufficientBalanceError if the withdrawal amount exceeds the balance.
- In the main function, the exception is caught and handled, providing a user-friendly error message.
- The ValueError is also handled in case the user enters a non-numeric value.

Creating an Exception Hierarchy

You can define a hierarchy of exceptions to represent different error conditions in a structured way.

Example:

```

class TransactionError(Exception):
    """Base class for transaction-related errors."""
    pass

class InsufficientBalanceError(TransactionError):
    """Raised when withdrawal exceeds balance."""
    pass

class NegativeAmountError(TransactionError):
    """Raised when a negative amount is provided."""
    pass

```

- Explanation:

- TransactionError is the base class for all transaction-related exceptions.
- Specific exceptions like InsufficientBalanceError and NegativeAmountError inherit from TransactionError.
- This hierarchy allows you to catch all transaction errors or handle specific ones.

Usage:

```
try:  
    # Simulated condition  
    amount = -100 # Change this value to test different exceptions  
  
    if amount < 0:  
        raise NegativeAmountError("Negative amount not allowed.")  
    elif amount > 500: # Simulated balance  
        raise InsufficientBalanceError("Insufficient funds for this  
transaction.")  
    else:  
        print("Transaction successful.")  
  
except InsufficientBalanceError as e:  
    print(f"Insufficient Balance Error: {e}")  
except NegativeAmountError as e:  
    print(f"Negative Amount Error: {e}")  
except TransactionError as e:  
    print(f"General Transaction Error: {e}")
```

Best Practices for User-Defined Exceptions

1. Inherit from the Right Base Class:

- Use Exception or Subclasses:
 - Always inherit from Exception or one of its subclasses, not from BaseException.

- This ensures that your exception behaves correctly with exception handling constructs.
2. Provide Meaningful Names and Messages:
- Descriptive Names:
 - Choose exception class names that clearly describe the error condition.
 - Informative Messages:
 - Include detailed error messages to aid in debugging.
3. Include Relevant Attributes:
- Store Error Details:
 - Add attributes to your exception class to hold information relevant to the error.
 - This allows handlers to access additional data when managing the exception.
4. Document Your Exceptions:
- Use Docstrings:
 - Provide docstrings for your exception classes to explain when they are raised and what they represent.
 - Explain Attributes:
 - Document any additional attributes your exception contains.
5. Organise Exceptions Hierarchically:
- Create Base Exception Classes:
 - Define base classes for groups of related exceptions.
 - Promote Reusability and Clarity:
 - A hierarchical structure makes it easier to manage exceptions and write cleaner code.
6. Avoid Overusing Exceptions:
- Use Exceptions Appropriately:

- Exceptions should represent unexpected or error conditions, not normal control flow.
- Don't Use Exceptions for Flow Control:
 - Rely on standard control structures (e.g., loops, conditionals) for regular program logic.

Advantages of Using User-Defined Exceptions

- Enhanced Readability and Maintenance:
 - Custom exceptions make it clear what kind of error has occurred.
 - They improve the self-documenting nature of the code.
- Better Error Handling:
 - Allows for more precise exception handling.
 - Handlers can catch specific exceptions and respond appropriately.
- Improved Debugging:
 - Provides more context about errors.
 - Easier to trace the source of an error when custom exceptions are used.

User-defined exceptions in Python empower developers to create more robust and maintainable applications by providing the ability to:

- Represent specific error conditions unique to the application.
- Handle errors gracefully with meaningful messages and appropriate actions.
- Organise exceptions in a hierarchical structure for better code organisation.

By following best practices and thoughtfully designing custom exceptions, you enhance your program's ability to handle errors effectively, leading to a better experience for both developers and users.



Formative Assessment Activity [5]

Exception Handling

Complete the formative activity in your **KM4 Learner Workbook**.

Structured query language (SQL)

SQL is a database language that is composed of commands that enable users to create databases or table structures, perform various types of data manipulation and data administration, as well as query the database to extract useful information. SQL is supported by all relational database management system (RDBMS) software. SQL is portable, which means that a user does not have to relearn the basics when moving from one RDBMS to another, because all RDBMSs use SQL in almost the same way.

SQL is easy to learn, as its vocabulary is relatively simple. Its basic command set has a vocabulary of fewer than 100 words. It is also a declarative language, which means that the user specifies what must be done and not how it should be done. Users do not need to know the physical data storage format or the complex activities that take place when an SQL command is executed in order to issue a command.

SQL functions fit into two general categories:

1. **Data definition language (DDL)** includes commands that enable users to define, edit, and delete data structures and schemas stored within a database management system. A database schema outlines the structure of a relational database, specifying how data is organised and connected. This includes defining the names and contents of tables, the types of data stored in each field, and how different tables relate to one another. Using commands like CREATE, ALTER, and DROP, a DDL can create new databases and tables, modify column definitions or table properties, add or remove constraints, and delete entire data structures when no longer needed. It also provides fine-grained control over the metadata systems that organise and categorise data.
2. **Data manipulation language (DML)** includes commands that enable users to access, modify, and manipulate data stored in a database. Key DML operations

map to the CRUD acronym – create new entries, read and retrieve existing records, update or edit stored values, and delete data. With commands like **SELECT**, **INSERT**, **UPDATE**, and **DELETE**, DML provides powerful tools for curating, shaping, cleansing, and managing large datasets, such as query filtering on specific criteria. Manipulating data dynamically is vital for impactful analytics.

The commands in each category are given in the tables below. We will explore some of these commands in the rest of this lesson.

The table below lists the SQL DDL commands:

Command	Description
CREATE SCHEMA AUTHORIZATION	Creates a database schema.
CREATE TABLE	Creates a new table in the user's database schema.
NOT NULL	Ensures that a column will not have null values.
UNIQUE	Ensures that a column will not have duplicate values.
PRIMARY KEY	Defines a primary key for a table.
FOREIGN KEY	Defines a foreign key for a table.
DEFAULT	Defines a default value for a column when no value is given.
CHECK	Used to validate data in an attribute.
CREATE INDEX	Creates an index for the table.
CREATE VIEW	Creates a dynamic subset of rows or columns from one or more tables.
ALTER TABLE	Modifies a table (adds, modifies, or deletes attributes or constraints).
CREATE TABLE AS	Creates a new table based on a query in the user's database schema.
DROP TABLE	Permanently deletes a table.
DROP INDEX	Permanently deletes an index.
DROP VIEW	Permanently deletes a view.

Table source (Rob & Coronel, 2009)

The table below lists the SQL DML commands:

Command	Description
INSERT	Inserts rows into a table.
SELECT	Select attributes from rows in one or more tables or views.
WHERE	Restricts the selection of rows based on a conditional expression.
GROUP BY	Groups the selected rows based on one or more attributes.
HAVING	Restricts the selection of grouped rows based on a condition.
ORDER BY	Orders the selected rows based on one or more attributes.
UPDATE	Modifies an attribute's values in one or more tables' rows.
DELETE	Deletes one or more rows from a table.
COMMIT	Permanently saves data changes.
ROLLBACK	Restores data to its original values.
Comparison Operators	=, <, >, <=, >=, <>
Logical Operators	AND, OR, NOT
Special Operators	Used in conditional expressions.
BETWEEN	Checks whether an attribute value is within a range.
IS NULL	Checks whether an attribute value is null.

LIKE	Checks whether an attribute value matches a given string pattern.
IN	Checks whether an attribute value matches any value within a value list .
EXISTS	Checks whether a subquery returns any rows.
DISTINCT	Limits values to unique values.
Aggregate Functions	Used with SELECT to return mathematical summaries on columns.
COUNT	Returns the number of rows with non-null values for a given column.
MIN	Returns the minimum attribute value found in a given column.
MAX	Returns the maximum attribute value found in a given column.
SUM	Returns the sum of all values for a given column.
AVG	Returns the average of all values for a given column.

Table source (Rob & Coronel, 2009)

Creating tables

To create new tables in SQL, you use the `CREATE TABLE` statement. Pass all the columns you want in the table, as well as their data types, as arguments to the `CREATE TABLE` function. The table will be organised by columns and rows, like a spreadsheet. Each column is called a field or attribute, and has a field name which functions like the column heading.

Each field holds data on a specific topic, where the field name usually indicates the topic. In the example table below, the fields are `StudentNumber`, `Name`, `Surname`, `CellNumber`, and `Address`. Each row, on the other hand, is called a record, and each record holds a full set of data for a particular entity/thing/person/etc.

In the example table below, the entities we're storing data about are students, and each row holds a record of a single student's data.

StudentNumber	Name	Surname	CellNumber	Address
4f8149817	Liano	Charook	082 283 9009	3 Maple Str
5e9285991	Bianca	Manan	072 329 5571	17 Willow Ave
9b7744992	Cameron	Devilliers	072 410 9077	9 Birch Lane

The syntax of the `CREATE TABLE` statement is shown below:

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ...
);
```

Note the optional constraint arguments. They are used to specify rules for data in a table. Constraints that are commonly used in SQL include:

- **NOT NULL:** Ensures that a specific column within a table cannot store null values, guaranteeing that data is always present in that column.
- **UNIQUE:** Ensures that all values in a specified column or combination of columns are distinct, preventing duplicate entries and maintaining data integrity.
- **DEFAULT:** Specifies a predefined value that is automatically inserted into a column if no other value is provided during data entry, ensuring data consistency and reducing null values.
- **INDEX:** Creates an index linked to a specific column or set of columns that is used to create and retrieve data from the database quickly. This works the same way as an index in a book, where keywords in the index enable us to quickly find those topics in the book, except that in a database the 'keyword' is a field name and indexing it helps the computer find and access the data in that field more quickly. For instance, if you frequently run queries to retrieve employees based on their `EmployeeID`, you should create an index on this column for faster lookup.

Additionally, note the use of a semicolon at the end of the statement. It indicates the completion of a single SQL command. While optional in some SQL environments, using semicolons consistently improves code readability and maintainability.

Let's look at another instance. To create a table called `Employee` that contains five columns (`EmployeeID`, `LastName`, `FirstName`, `Address`, and `PhoneNumber`), you would use the following SQL:

```
CREATE TABLE Employee (
    EmployeeID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    PhoneNumber varchar(255)
);
```

The `EmployeeID` column is of type `int` and will, therefore, hold an integer value. The `LastName`, `FirstName`, `Address`, and `PhoneNumber` columns are of type `varchar` and will, therefore, hold characters. The number in brackets indicates the maximum number of characters, which in this case is 255.

The `CREATE TABLE` statement above will create an empty `Employee` table that will look like this:

EmployeeID	LastName	FirstName	Address	PhoneNumber

When creating tables, it is advisable to add a **primary key** to one of the columns, as this will help keep entries unique and will speed up `SELECT` queries. Primary keys must contain unique values and cannot contain null values. A table can only contain one primary key; however, the primary key may consist of a single column or a combination of multiple columns.

You can add a primary key when creating the `Employee` table as follows:

```
CREATE TABLE Employee (
    EmployeeID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Address varchar(255),
    PhoneNumber varchar(255),
    PRIMARY KEY (EmployeeID)
);
```

To name a primary key constraint and define a primary key constraint on multiple columns, you would use the following SQL syntax:

```
CREATE TABLE Employee (
    EmployeeID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Address varchar(255),
    PhoneNumber varchar(255),
    CONSTRAINT PK_Employee PRIMARY KEY (EmployeeID, LastName)
);
```

In the example above, there is only one primary key named `PK_Employee`. However, the value of the primary key is made up of two columns: `EmployeeID` and `LastName`.

Inserting rows

The table that we have just created is empty and needs to be populated with rows or records. We can add entries to a table using the `INSERT INTO` command.

There are two ways to write the `INSERT INTO` command: inserting values for specific columns and inserting values for all columns. It's generally a good practice to explicitly specify the columns for which you are providing values during an `INSERT` statement.

Explicit column specification makes the code more readable and reduces the risk of errors, especially when the table schema changes or new columns are added.

1. Inserting values for specific columns:

If you only want to insert data into specific columns and want to specify both the column names and the corresponding values, you can use this method. The syntax is as follows:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

For instance, to add an entry to the `Employee` table using this approach, you would write:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address,
PhoneNumber)
VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

This method is useful when you want to insert data into specific columns and leave other columns with default values or `NULL`. For example, as the `FirstName` and `Address` fields don't have a `NOT NULL` constraint, they don't necessarily have to be included when inserting. The `FirstName` and `Address` fields for this record will be `NULL`:

```
INSERT INTO Employee (EmployeeID, LastName, PhoneNumber)
VALUES (1321, 'Jones', '0827546787');
```

To insert multiple rows using this method, you can provide multiple sets of values within parentheses, separated by commas. Each set of values represents a row to be inserted. Here's an instance:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address,
PhoneNumber)

VALUES

(1234, 'Smith', 'John', '25 Oak Rd', '0837856767'),
(5678, 'Brown', 'Robert', '5 Pine Str', '0821116789'),
(9112, 'Davies', 'Emily', '25 Maple Ave', '0876543210');
```

2. Inserting values for all columns:

If you want to add values for all the columns in the table and the order of the values matches the exact order of the columns, you can simply not specify the column names. The syntax for this approach is as follows:

```
INSERT INTO table_name

VALUES (value1, value2, value3, ...);
```

For instance, to add an entry to the `Employee` table, you would do the following:

```
INSERT INTO Employee

VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

To insert multiple rows using this method, you can simply provide multiple sets of values within parentheses, separated by commas. Each set of values represents a row to be inserted. For example:

```
INSERT INTO Employee

VALUES

(1234, 'Smith', 'John', '25 Oak Rd', '0837856767'),
(5678, 'Brown', 'Robert', '5 Pine Str', '0821116789'),
(9112, 'Davies', 'Emily', '25 Maple Ave', '0876543210');
```

Retrieving data from a table

The `SELECT` statement is used to fetch data from a database. The data returned is stored in a result table, known as the result set. The syntax of a `SELECT` statement is as follows:

```
SELECT column1, column2, ...
FROM table_name;
```

Here, `column1`, `column2`, ... refer to the column names of the table from which you want to select data. The following example below selects the `FirstName` and `LastName` columns from the `Employee` table:

```
SELECT FirstName, LastName
FROM Employee;
```

If you want to select all the columns and rows in the `Employee` table, use the following syntax:

```
SELECT * FROM Employee;
```

The asterisk (*) indicates that we want to fetch all of the columns, without excluding any of them.

You can also order and filter the data that is returned when using the `SELECT` statement using the `ORDER BY` and `WHERE` commands.

Order by

You can use the `ORDER BY` command to sort the results returned in ascending or descending order based on one or more columns. The `ORDER BY` command sorts the records in ascending order by default. You need to use the `DESC` keyword to sort the records in descending order.

The `ORDER BY` syntax is as follows:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

The example below selects all employees in the `Employee` table and sorts them in descending order (4, 3, 2, 1 ... for numbers, and Z...A for letters), based on the values in the `FirstName` column:

```
SELECT *
FROM Employee
ORDER BY FirstName DESC;
```

Where

The `WHERE` clause allows us to filter data depending on a specific condition. The syntax of the `WHERE` clause is as follows:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

The following SQL statement selects all the employees with the first name ‘John’, in the `Employee` table:

```
SELECT *
FROM Employee
WHERE FirstName = 'John';
```

Note that SQL requires single quotes around text values; however, you do not need to enclose numeric fields in quotes. You should also note that, for conditions in SQL, we use a single ‘=’, which is equivalent to the ‘==’ used in Python, JavaScript, and Java.

You can use logical operators (`AND`, `OR`) and comparison operators (`=, <, >, <=, >=, <>`) to make `WHERE` conditions as specific as you like.

For instance, suppose you have the following table that contains the most sold albums of all time:

Artist	Album	Released	Genre	sales_in_millions
Michael Jackson	<i>Thriller</i>	1982	pop	70
AC/DC	<i>Back in Black</i>	1980	rock	50
Pink Floyd	<i>The Dark Side of the Moon</i>	1973	rock	45
Whitney Houston	<i>The Bodyguard</i>	1992	soul	44

You can select the records that are classified as rock and have sold under 50 million copies by simply using the AND operator as follows:

```
SELECT *
FROM albums
WHERE Genre = 'rock' AND sales_in_millions <= 50
ORDER BY Released
```

WHERE statements also support some special operators to customise queries further:

- **IN:** Compares the column to multiple possible values and returns true if it matches at least one.
- **BETWEEN:** Checks if a value is within an inclusive range.
- **LIKE:** Searches for a pattern match, which is specific character patterns within text data. For instance, using the pattern specification S% searches for all text that starts with an 'S' followed by any number of other characters (no matter which characters). The percentage sign is referred to as a wildcard, which represents any characters. An underscore represents any single character. For example, J__n can be used to search for any text that starts with a 'J' followed by two characters and ends with an 'n'.

For example, if we want to select the pop and soul albums from the table above, we can use:

```
SELECT *
FROM albums
WHERE Genre IN ('pop', 'soul');
```

Or, if we want to get all the albums released between 1975 and 1985, we can use:

```
SELECT *
FROM albums
WHERE Released BETWEEN 1975 AND 1985;
```

We can also find all albums sung by artists whose names start with an M:

```
SELECT *
FROM albums
WHERE Artist LIKE 'M%';
```

Using aggregate functions

SQL has many functions that are helpful. Some of the most regularly used ones are:

- COUNT(): Returns the number of rows.
- SUM(): Returns the total sum of a numeric column.
- AVG(): Returns the average of a set of values.
- MIN() / MAX(): Gets the minimum or maximum value from a column.
- GROUP BY: Group rows are returned by a query and divided into summary rows based on the values in one or more columns.

For example, to get the most recent year in the `album` table, we can use:

```
SELECT MAX(Released)
FROM albums;
```

Or to get the number of albums released between 1975 and 1985, we can use:

```
SELECT COUNT(*)  
FROM albums  
WHERE Released BETWEEN 1975 AND 1985;
```

The GROUP BY clause is often used in conjunction with the other functions. The following SQL command calculates the total sales by genre:

```
SELECT Genre, SUM(sales_in_millions) AS total_sales  
FROM albums  
GROUP BY Genre;
```

Retrieving data across multiple tables

In complex databases, there are often several tables connected in some way. A JOIN clause is used to combine rows from two or more tables, based on a related column between them. Look at the two tables below:

VideoGame table:

ID	Name	DeveloperID	Genre
1	<i>Super Mario Bros.</i>	2	platformer
2	<i>World of Warcraft</i>	1	MMORPG
3	<i>The Legend of Zelda</i>	2	adventure

GameDeveloper table:

ID	Name	Country
1	Blizzard	USA
2	Nintendo	Japan

The `VideoGame` table contains information about various video games, while the `GameDeveloper` table contains information about the developers of the games. The `VideoGame` table has a `DeveloperID` column that holds the game developer's ID, which represents the ID of the respective developer from the `GameDeveloper` table. `DeveloperID` points to a specific developer with a matching ID (primary key) in the `GameDeveloper` table; a column that can be used to link two tables like this is called a foreign key. A foreign key in one table **must always correspond** to a primary key in another table.

From the tables above, we can see that a developer called Blizzard from the USA developed the game entitled *World of Warcraft*. The foreign key logically links the two tables, and allows us to access and use the information stored in both of them at the same time.

If we want to create a query that returns everything we need to know about the games, we can use `INNER JOIN` to acquire the columns from both tables.

```
SELECT VideoGame.Name, VideoGame.Genre, GameDeveloper.Name, GameDeveloper.Country  
FROM VideoGame  
INNER JOIN GameDeveloper  
ON VideoGame.DeveloperID = GameDeveloper.ID;
```

Notice that in the `SELECT` statement above, we specify the name of the table and the column from which we want to retrieve information, and not just the name of the column as we have done previously. This is because we are getting information from more than just one table, and tables may have columns with the same names. In this case, both the table `VideoGame` and the table `GameDeveloper` contain columns called `Name`. In the next section, you will see how to use aliases to further address this issue.

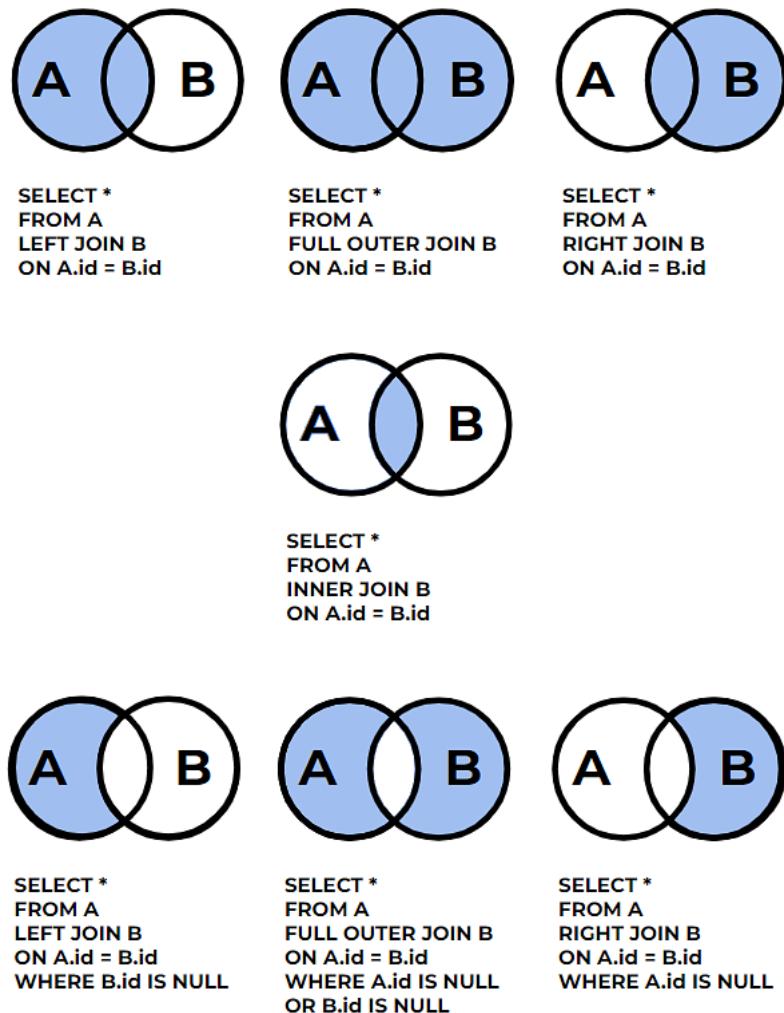
Also, notice that we use the `ON` clause to specify how we link the foreign key in one table to the corresponding primary key in the other table. The query above would result in the following dataset being returned:

<code>VideoGame.Name</code>	<code>VideoGame.Genre</code>	<code>GameDeveloper.Name</code>	<code>GameDeveloper.Country</code>
<i>Super Mario Bros.</i>	platformer	Nintendo	Japan
<i>World of Warcraft</i>	MMORPG	Blizzard	USA
<i>The Legend of Zelda</i>	adventure	Nintendo	Japan

The `INNER JOIN` is the simplest and most common type of `JOIN`. However, there are many other different types of joins in SQL. Let's take a look at these and what they do.

- `INNER JOIN`: Returns records that have matching values in both tables.
- `LEFT JOIN`: Returns all records from the left table, and the matched records from the right table.
- `RIGHT JOIN`: Returns all records from the right table, and the matched records from the left table.
- `FULL JOIN`: Returns all records when there is a match in either the left or right table.

The figure below provides a graphical representation of the above joins, plus some extras that are less common. Look carefully at the Venn diagrams and ensure you understand exactly which records would be returned from tables A and B if the given SQL code was executed.



Graphical representation of common SQL joins

Using aliases

Notice that in the `VideoGame` and `GameDeveloper` tables, there are two columns called `Name`. This can become confusing. Aliases are used to give a table or column a temporary name. An alias only exists for the duration of the query and is often used to make column names more readable.

The alias column syntax is:

```
SELECT column_name AS alias_name  
FROM table_name;
```

The alias table syntax is:

```
SELECT column_name(s)  
FROM table_name AS alias_name;
```

The following SQL statement creates an alias for the `Name` column from the `GameDeveloper` table:

```
SELECT Name AS Developer  
FROM GameDeveloper;
```

See how aliases have been used below:

```
SELECT games.Name, games.Genre, devs.Name AS Developer, devs.Country  
FROM VideoGame AS games  
INNER JOIN GameDeveloper AS devs  
ON games.DeveloperID = devs.ID;
```

As you can see, this starts to get quite difficult to follow and read, as the query statements grow. In SQL, the `WITH` clause, also known as a Common Table Expression (CTE), allows you to define a temporary result set that you can reference within the context of a larger SQL query. It enhances the readability and maintainability of complex queries by breaking them down into smaller, named, and reusable components.

The `WITH` clause example is focused solely on retrieving developer names from the `GameDeveloper` table:

```
WITH GameInfo AS (
    SELECT games.ID, games.Name AS GameName, games.Genre, devs.Name AS Developer
    FROM VideoGame AS games
    INNER JOIN GameDeveloper AS devs ON games.DeveloperID = devs.ID
)
```

The `GameInfo` CTE includes information about the video games and their developers by joining the `VideoGame` and `GameDeveloper` tables. The main query then selects distinct developer names from the CTE:

```
SELECT DISTINCT Developer
FROM GameInfo;
```

This approach allows you to create a CTE that encompasses the logic of joining the tables and then reuse the CTE in subsequent queries.

Updating data

The `UPDATE` statement is used to modify the existing rows in a table.

To use the `UPDATE` statement, you:

- Choose the table with the row you want to change.
- Set the new value(s) for the desired column(s).
- Select which of the rows you want to update using the `WHERE` statement. If you omit this, all rows in the table will change.

The syntax for the update statement is:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Take a look at the following `Customer` table:

CustomerID	CustomerName	Address	City
1	Maria Anderson	23 York St	New York
2	Jackson Peters	124 River Rd	Berlin
3	Thomas Hardy	455 Hanover Sq	London
4	Kelly Martins	55 Loop St	Cape Town

The following SQL statement updates the first customer (`CustomerID = 1`) with a new address and a new city:

```
UPDATE Customer
SET Address = '78 Oak St', City = 'Los Angeles'
WHERE CustomerID = 1;
```

Deleting rows

Deleting a row is a simple process. All you need to do is select the right table and row that you want to remove. The `DELETE` statement is used to delete existing rows in a table.

The `DELETE` statement syntax is as follows:

```
DELETE FROM table_name
WHERE condition;
```

The following statement deletes the customer `Jackson Peters` from the `Customer` table:

```
DELETE FROM Customer
WHERE CustomerName = 'Jackson Peters';
```

You can also delete all the data inside a table, i.e. all rows, without deleting the table:

```
DELETE FROM table_name;
```

When dealing with foreign keys, it's important to understand the concept of referential integrity. Referential integrity ensures that relationships between tables remain valid, meaning that a foreign key in one table must correspond to a primary key in another table. When you want to delete records from a table that is referenced by foreign keys in other tables, you need to be careful to maintain referential integrity.

Therefore, deletion is a last resort because it can lead to data inconsistency and integrity issues if not handled carefully. Always ensure that deleting records won't violate referential integrity and consider alternative approaches to ensure that the database remains well-structured and consistent.

Deleting tables

The `DROP TABLE` statement is used to remove every trace of a table from a database. The syntax is as follows:

```
DROP TABLE table_name;
```

For instance, if we want to delete the table `Customer`, we do the following:

```
DROP TABLE Customer;
```

If you want to delete the data inside a table, but not the table itself, you can use the `TRUNCATE TABLE` statement:

```
TRUNCATE TABLE table_name;
```

While the `TRUNCATE TABLE` and `DELETE FROM` statements perform similarly in terms of removing data from a table, there are a few important differences to consider when deciding which statement to use:

<code>TRUNCATE TABLE</code>	<code>DELETE FROM table_name</code>
A data definition language command (DDL) command that's automatically committed. This means data cannot be recovered, as it is a non-transactional operation (it cannot be rolled back once executed) and does not generate any undo logs.	A data manipulation language (DML) command that is not automatically committed. This means data can be recovered, as it supports transactional operations and logs row-level details.
Removes all data rows and resets certain attributes associated with the table's structure, such as the value of the identity columns, to their initial values.	Removes all data rows while retaining attributes associated with the table's structure.
Faster speed and performance, as it removes all rows as a single operation.	Slower speed and performance, as it removes each row one by one.
Use case: If you have a transactional database where you want to do a bulk purge of all the transactional data accumulated over time and reclaim storage space.	Use case: If you want to delete specific data, based on certain conditions, with the added flexibility to roll back if needed.



Take note

The behaviour of `TRUNCATE TABLE` statement can vary across different database providers. While some database systems support rollback, others may not e.g. in the Oracle DB. Therefore, it's important to consult the documentation specific to your database to understand whether rollback is supported. In SQLite, the `TRUNCATE` statement is not supported. Instead, you can use the `DELETE FROM` statement without a `WHERE` clause to achieve similar results.



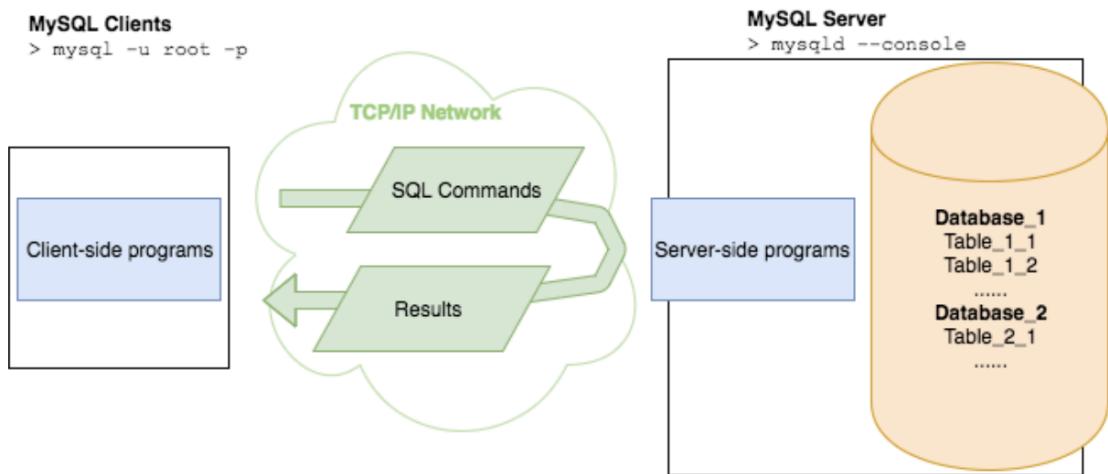
Extra resource

If you'd like to know more about SQL, we highly recommend reading the book [**Database Design – 2nd Edition**](#) by Adrienne Watt. Chapters 15 and 16, and Appendix C of this book provide more detail regarding what has been covered in this task.

SQLite features

Important features to note about SQLite are that it is self-contained, serverless, transactional, and requires zero configuration to run. Let's look more closely at these properties.

- **Self-contained:** This means that SQLite does not need much support from the operating system or external libraries. This makes it suitable for use in embedded devices like mobile phones, iPods, and game devices that lack the infrastructure provided by a regular computer. In Python, you can access SQLite databases using the `sqlite3` module, which is part of Python's standard library. This means you do not need to manage SQLite source code files like `sqlite3.c` or `sqlite3.h` yourself. Instead, simply import the `sqlite3` module to interact with SQLite databases in your Python projects.
- **Serverless:** In most cases, RDBMSs require a separate server to receive and respond to requests sent from the client, as shown in the diagram below.



SQLite as a serverless RDBMS (SQLite, n.d.)

Such systems include MySQL, MariaDB, and the Java Database Connectivity (JDBC). These clients have to use the TCP/IP protocol to send and receive responses. This is referred to as the client/server architecture. SQLite does not make use of a separate server and, therefore, does not utilise client/server architecture. Instead, the entire SQLite database is embedded into the application that needs to access the database.

- **Transactional:** All transactions in SQLite are atomic, consistent, isolated, and durable (**ACID**-compliant). In other words, if a transaction occurs that attempts to make changes to the databases, the changes will either be made in all the appropriate places (in all linked tables and affected rows) or not at all. This is to ensure data integrity (i.e., to avoid conflicting records in different places due to some being updated and others not).
- **Zero configuration required:** You don't need to install SQLite prior to using it in an application or system. This is because of the previously described serverless characteristic.

Python's SQLite module

It is easy to create and manipulate databases with Python. To enable the use of SQLite with Python, the Python standard library includes a module called `sqlite3`. To use this module, we need to add an `import` statement to our Python script:

```
import sqlite3
```

We can then use the function `sqlite3.connect()` to connect to the database. We pass the name of the database file to this function to open or create the database.

```
# Creates or opens a file called student_db with an SQLite3 DB
db = sqlite3.connect("student_db.db")
```

Creating and deleting tables

To make any changes to the database, we need a **cursor object**, which is an object that is used to execute SQL statements. Next, we use `.commit()` to save changes to the database. It is important to remember to commit changes since this ensures the atomicity of the database. If you close the connection using `close()` or the connection to the file is lost, changes that have not been committed will be lost.

Below we create a student table with `id`, `name`, and `grade` columns:

```
cursor = db.cursor() # Get a cursor object

# Execute a SQL command to create the student table
cursor.execute('''
    CREATE TABLE student(
        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
''')

# Commit the changes to the database to ensure they are saved
db.commit()
```

In the above code snippet, a cursor object is obtained from the database connection (`db`). Subsequently, an SQL query is executed using the cursor to create a new table named `student` with columns named `id` (as the primary key), `name`, and `grade`. The `db.commit()` statement is used to commit the transaction, finalising the table creation in the database.

Using IF NOT EXISTS

When working with databases and creating tables, it is often necessary to ensure that a table does not already exist before attempting to create the table. The `IF NOT EXISTS` clause can be used to create a table only if it does not already exist, which helps prevent errors that might occur if the table is already present in the database.

Let's have a look at an example demonstrating how the `CREATE TABLE` statement can be modified to include the `IF NOT EXISTS` clause, ensuring the table is created only if it does not already exist.

```
# Get the cursor object
cursor = db.cursor()

# Create the student table if it does not exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS student (
        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
''')

# Commit the changes to the database
db.commit()
```

Always remember that the `commit()` function is invoked on the `db` object, not the `cursor` object. If we type `cursor.commit()`, we will get the following error message:

```
AttributeError: 'sqlite3.Cursor' object has no attribute 'commit'
```

Inserting into the database

To insert data into a database we use prepared statements. This is a secure method for executing SQL queries with Python. Prepared statements involve using placeholders, such as "?", in your SQL query instead of directly including data. This approach ensures that your data is handled safely and efficiently. Avoid using string operations or concatenation to construct SQL queries, as these methods can be less secure.

In this example, we are going to insert two students into the database, whose information is stored in Python variables.

```
name1 = 'Andres'
grade1 = 60

name2 = 'John'
grade2 = 90

# Insert student 1
cursor.execute('''
    INSERT INTO student(name, grade)
    VALUES (?, ?)
''', (name1, grade1))

print('First user inserted')

# Insert student 2
cursor.execute('''
    INSERT INTO student(name, grade)
    VALUES (?, ?)
''', (name2, grade2))

print('Second user inserted')

db.commit()
```

In the example above, the values of the Python variables are passed inside a [tuple](#). You could also use a dictionary with the named style placeholder:

```
name3 = 'Sheila'
grade3 = 40

# Insert student 3 using named parameters
cursor.execute('''
    INSERT INTO student (name, grade)
    VALUES (:name, :grade)
''', {'name': name3, 'grade': grade3})

print('Third user inserted')
```

If you need to insert several users, use `executemany` and a list with the tuples:

```
students_grades = [(name1, grade1), (name2, grade2), (name3, grade3)]  
  
cursor.executemany(  
    '''INSERT INTO student(name, grade) VALUES(?, ?)''', students_grades  
)  
  
db.commit()
```

Each record inserted into the table gets a unique `id` value starting at one and ascending in increments of one for each new record. If you need to get the `id` of the row you just inserted, use `lastrowid`:

```
# Get the ID of the last inserted row  
last_row_id = cursor.lastrowid  
print(f'Last row ID: {last_row_id}')
```

Use `rollback()` to roll back any change to the database since the last call to commit:

```
cursor.execute('''UPDATE student SET grade = ? WHERE id = ?''', (65, 2))  
  
db.rollback()
```

Retrieving data

To retrieve data, execute a `SELECT` SQL statement against the `cursor` object, and then use `fetchone()` to retrieve a single row or `fetchall()` to retrieve all the rows.

Here is an example using `fetchone()` to retrieve the first student record that matches the specified ID:

```
# Define the ID of the student we want to retrieve  
id = 3  
  
# Execute a query to select the name and grade of the student with the  
# specified ID  
cursor.execute('''SELECT name, grade FROM student WHERE id = ?''', (id,))  
  
# Fetch the first row that matches the query  
student = cursor.fetchone()  
  
# Print the retrieved student's name and grade
```

```
print(student)
```

Note: When using a single variable in a query with placeholders, ensure you include a comma to create a single-element tuple, for example “(id,)”. This is necessary for the query to work correctly.

If you use `fetchone()` and there are multiple rows that match the criteria, only the first one will be retrieved. If you expect multiple rows, rather use `fetchall()`.

To retrieve all student records where the grade is below a specified threshold, we can use `fetchall()` as shown below:

```
# Define the grade threshold
grade_threshold = 80

# Execute a query to select all students with grades less than the specified
# threshold
cursor.execute('SELECT name, grade FROM student WHERE grade < ?',
(grade_threshold,))

# Fetch all rows that match the query
students = cursor.fetchall()

# Print each student's name and grade
print(f'Students with a grade less than {grade_threshold}:')
for student in students:
    print(f'{student[0]} : {student[1]}')
```

In the above example, the code selects all student names and grades from the table where each student's grade is less than a specified threshold. This query returns a result set, which is a list of tuples, with each tuple containing a pair of student names and their corresponding grades. By looping through this list, you can use indexing (`student[0]` to access the name and `student[1]` to access the grade) to retrieve and display the name and grade for each student.

Alternatively, you can use the `cursor` object, which works as an iterator, invoking `fetchall()` automatically:

```
cursor.execute('''SELECT name, grade FROM student''')

# Iterate over the result set returned by the query
for row in cursor:
    # Each 'row' is a tuple where row[0] is the student's name and row[1]
    # is their grade.
    # Print the student's name and grade in a formatted string
```

```
print(f'{row[0]} : {row[1]}' )
```

Updating and deleting data

Updating or deleting data is similar to inserting data:

```
# Update user with id 1
grade = 100
user_id = 1
cursor.execute('''UPDATE student SET grade = ? WHERE id = ?''', (grade,
user_id))

# Delete user with id 2
user_id = 2
cursor.execute('''DELETE FROM student WHERE id = ?''', (user_id,))

# Drop the student table
cursor.execute('''DROP TABLE student''')

# Commit the changes
db.commit()
```

When we are done working with the db, we need to close the connection. Failing to close the connection could result in issues such as incomplete transactions, data corruption, and resource leaks.

```
db.close()
```

SQLite database exceptions

It is very common for exceptions to occur when working with databases, so it is important to handle these exceptions in your code.

In the example below, we use a `try/except/finally` clause to catch any exception in the code. We put in the code that we would like to execute, but that may throw an exception (or cause an error) in the `try` block.

Within the `except` block, we write the code that will be executed if an exception does occur. If no exception is thrown, the `except` block will be ignored. The `finally` clause will always be executed, whether an exception was thrown or not. When working with databases, the `finally` clause is very important, because it always closes the database connection correctly. View this resource to find out more [about exceptions](#).

```
import sqlite3

try:
    # Creates or opens a file called student_db with an SQLite3 DB
    db = sqlite3.connect('student_db.db')
    # Get a cursor object
    cursor = db.cursor()
    # Checks if the table "users" exists and if not creates it
    cursor.execute(
        '''
        CREATE TABLE IF NOT EXISTS
        users(id INTEGER PRIMARY KEY, name TEXT, grade INTEGER)
        '''
    )
    # Commit the change
    db.commit()
# Catch the exception
except Exception as e:
    # Roll back any change if something goes wrong
    db.rollback()
    raise e
finally:
    # Close the db connection
    db.close()
```

Notice that the `except` block of our `try/except/finally` clause in the example above will be executed if any type of error occurs:

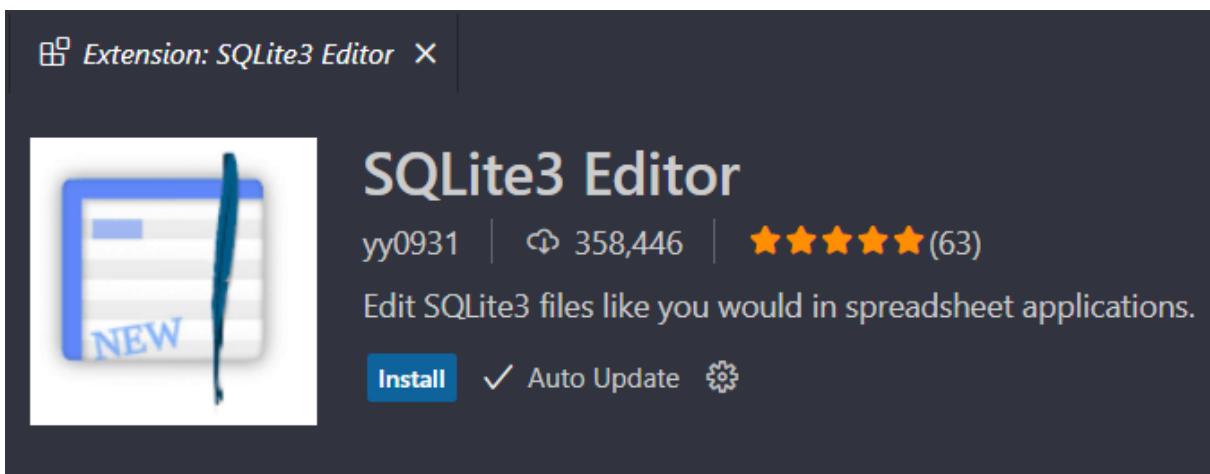
```
# Catch the exception
except Exception as e:
    raise e
```

This is called a catch-all clause. In a real application, you should catch a specific exception. To see what type of exceptions could occur, see here for [database API v2.0 exceptions](#).

SQLite3 Editor

One effective way to explore and manage your SQLite database is by using the [**SQLite3 Editor**](#) extension in Visual Studio Code. This tool allows you to manage databases by viewing and editing data, modifying table structures, running SQL queries, and visualising table relationships. To install and use this tool, follow these steps:

1. Open Visual Studio Code and go to the Extensions view.
2. Search for **SQLite3 Editor** in the Extensions Marketplace, select it from the results, and click Install to add it to VS Code. Once the installation is complete, you may be prompted to reload VS Code to apply the changes. If prompted, click Reload, and follow any additional on-screen instructions to finalise the setup.



3. Navigate to the folder containing your `student_db.db` database file using the Explorer panel in VS Code.
4. Click on the database file to open it. You should now be able to view and manage its contents, including running queries and modifying table structures.

The first time you use the extension, you may be presented with a guided walkthrough or on-screen tips to help you navigate its features.

The screenshot shows the SQLite3 Editor extension interface in Visual Studio Code. At the top, there's a toolbar with buttons for 'Schema' (selected), 'Edit', 'Comment', 'Query Editor', and 'Auto Reload'. Below the toolbar, the 'Schema' tab is active, displaying the SQL code for creating a 'student' table:

```
CREATE TABLE student (
    id INTEGER PRIMARY KEY,
    name TEXT,
    grade INTEGER
)
```

Below the schema, there are buttons for 'Index' (+ Add) and 'Trigger' (+ Add). A table view shows the data for the 'student' table:

	<i>id INTEGER PRIMARY KEY</i>	<i>name TEXT</i>	<i>grade INTEGER</i>	+
1	1	Andres	60	
2	2	John	90	
3	3	Sarah	75	

With the **SQLite3 Editor** extension, you can easily interact with your database without leaving VS Code. This tool allows you to execute SQL queries, modify tables, insert or update records, and visualise database relationships. To explore additional features and functionalities of the **SQLite3 Editor** extension, visit the official [Visual Studio Marketplace page](#).

Example

```
"""
This script performs various SQLite operations using Python.
It connects to or creates a database file, creates a table,
inserts multiple records, retrieves and updates data, deletes
a record, and finally drops the table and closes the connection.
"""

import sqlite3

# Connect to or create a SQLite database file
db = sqlite3.connect('student_db.db')

# Get a cursor object to interact with the database
cursor = db.cursor()

# Create the student table if it does not exist
cursor.execute(
    '''
    CREATE TABLE IF NOT EXISTS student (

```

```

        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
    ...
)

# Commit the changes to save the table creation
db.commit()

# Define data for students
students_data = [
    ('Andres', 60),
    ('John', 90),
    ('Sarah', 75)
]

# Insert multiple students into the table
cursor.executemany(
    """
    INSERT INTO student(name, grade)
    VALUES(?, ?)
    """,
    students_data
)
print('Multiple users inserted.\n')

# Commit the changes to save the inserted data
db.commit()

# Get the ID of the last inserted row
last_row_id = cursor.lastrowid
print(f'Last row ID: {last_row_id}\n')

# Retrieve and display data for the student with a specific ID
id = 3

# Execute query to fetch student with specific ID
cursor.execute('''SELECT name, grade FROM student WHERE id = ?''', (id,))

# Fetch the result of the query
student = cursor.fetchone()

# Print the student's details if found,
# otherwise output that no student was found
if student:
    print(f'Student with ID {id}: {student}\n')
else:

```

```

print(f'No student found with ID {id}.\n')

# Retrieve all student records below a specific grade threshold
grade_threshold = 80

# Execute query to fetch students with grades below the threshold
cursor.execute(
    '''
    SELECT name, grade FROM student WHERE grade < ?
    ''', (grade_threshold,))
)

# Fetch all matching records
students = cursor.fetchall()

# Print each student's details if records are found,
# otherwise output that no students met the criteria.
if students:
    print(f'Students with a grade less than {grade_threshold}:')
    for student in students:
        print(f'{student[0]} : {student[1]}')
else:
    print(f'No students found with a grade less than {grade_threshold}.\n')

# Update the grade of the student with ID=1
grade = 100
id = 1

# Execute UPDATE statement to modify the student's grade
cursor.execute(
    '''
    UPDATE student SET grade = ? WHERE id = ?
    ''', (grade, id))
)

# Commit the changes to save the updated data
db.commit()

print(f'\nStudent with ID {id} updated with new grade {grade}.\n')

# Retrieve and display all students' names and grades
print('Updated student data:')

# Execute query to fetch all student records
cursor.execute('''SELECT name, grade FROM student''')

# Fetch all records
students = cursor.fetchall()

```

```
# Print each student's details if records are found, otherwise
# output that no data is available
if students:
    for row in students:
        print(f'{row[0]} : {row[1]}')
else:
    print('No student data available.\n')

# Delete the student with ID=2
id = 2

# Execute DELETE statement to remove student with specific ID
cursor.execute('''DELETE FROM student WHERE id = ?''', (id,))

# Commit the changes to save the deletion
db.commit()

print(f'\nStudent with ID {id} deleted.\n')

# Drop the 'student' table
cursor.execute('''DROP TABLE student''')
print('Student table deleted!\n')

# Commit the changes to save the table deletion
db.commit()

# Close the database connection
db.close()
print('Connection to database closed.')
```



Share your thoughts

Please take some time to complete this short feedback [form](#) to help us ensure we provide you with the best possible learning experience.

References

- Phillips, D. (2021). Python 3 object-oriented programming: Build robust and maintainable software with object-oriented design patterns in Python 3.8 (3rd ed.). Packt Publishing.
- Ramalho, L. (2022). Fluent Python: Clear, concise, and effective programming (2nd ed.). O'Reilly Media.
- Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.
- Slatkin, B. (2019). Effective Python: 90 specific ways to write better Python (2nd ed.). Addison-Wesley.
- Barry, P. (2016). Head First Python: A brain-friendly guide (2nd ed.). O'Reilly Media.
- Beazley, D., & Jones, B. K. (2013). Python cookbook: Recipes for mastering Python 3 (3rd ed.). O'Reilly Media.
- Budd, T. (2001). An introduction to object-oriented programming (3rd ed.). Addison-Wesley.
- Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
- Rob, P., & Coronel, C. (2009). *Database Systems: Design, Implementation, and Management*, (8th ed.). Boston, Massachusetts: Course Technology.