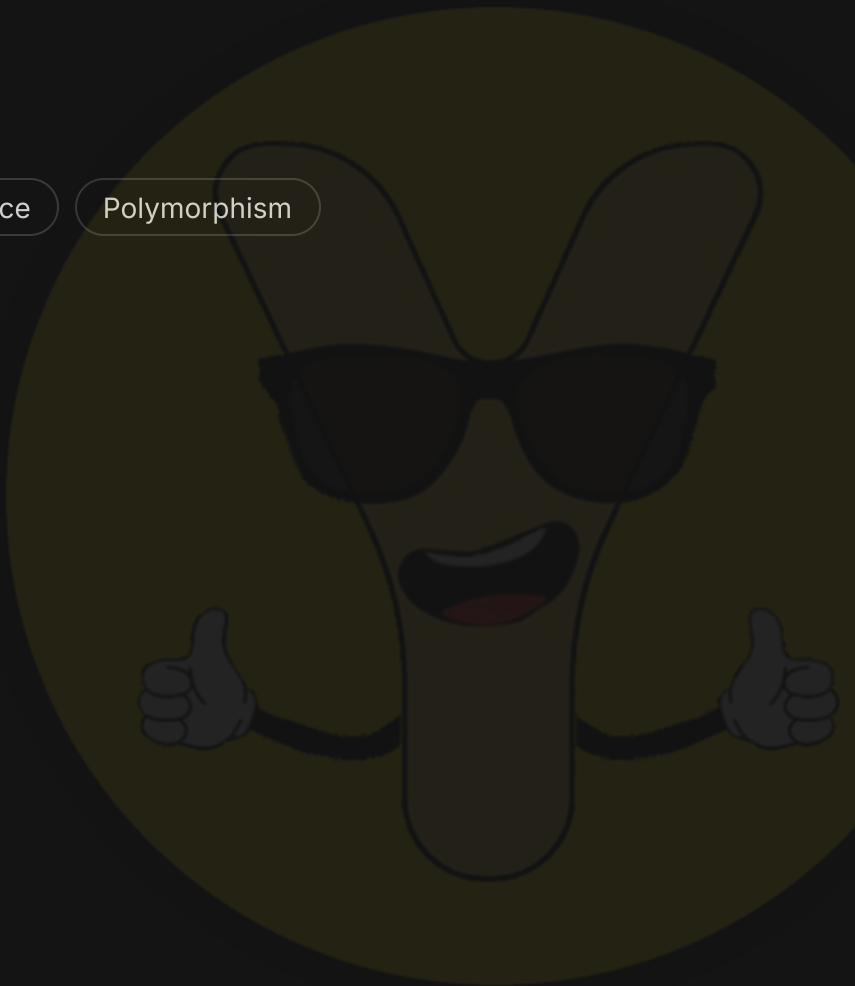
AKTU · 2ND YEAR · BCC302 / BCC402

Python Programming

Unit 3 — Object-Oriented Programming and Files

- Object-Oriented Programming concepts
- Classes and objects
- Constructors and destructors
- Inheritance
- Polymorphism
- Encapsulation and data hiding
- Operator overloading
- File I/O: opening, reading, writing files
- Exception handling: try, except, else, finally, custom exceptions



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

theycotes.com

Unit 3: Object-Oriented Programming and Files

Introduction to modular programming through classes, advanced data manipulation via file streams, and robust runtime management.

- Object-Oriented Programming principles and implementation in Python.
- Reusability features including Inheritance, Polymorphism, and Operator Overloading.
- Data security through Encapsulation and Data Hiding.
- Persistent storage management via File I/O operations.
- Runtime stability using Exception Handling constructs.

Object-Oriented Programming Concepts

OOP

A programming paradigm based on the concept of 'objects', which bundle data (attributes) and behavior (methods) together.

- Models real-world entities directly into software components.
- Promotes code reusability, modularity, and maintainability.
- Core pillars: Classes/Objects, Inheritance, Polymorphism, and Encapsulation.

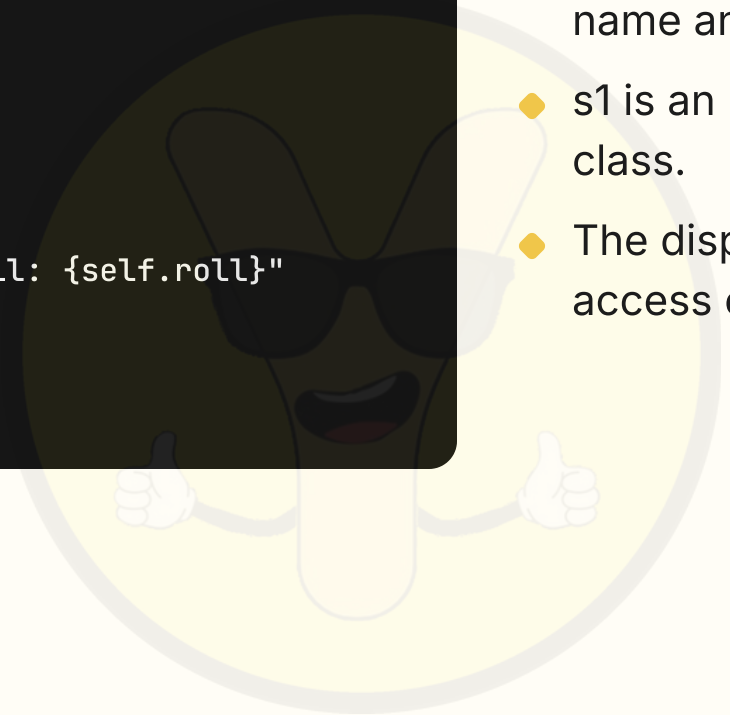
Classes and Objects

Understanding the blueprint and its physical realization in memory.

- Class
  - A user-defined prototype or blueprint from which objects are created.
  - Defines a set of attributes and methods common to all objects of one type.
- Object
  - An instance of a class containing actual state and behavior.
  - Allocates physical memory when instantiated.
- Self Parameter
  - Represents the instance of the class and must be the first parameter of any method.

Class and Object Implementation

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

    def display(self):
        return f"Name: {self.name}, Roll: {self.roll}"

s1 = Student("Alice", 101)
print(s1.display())
```

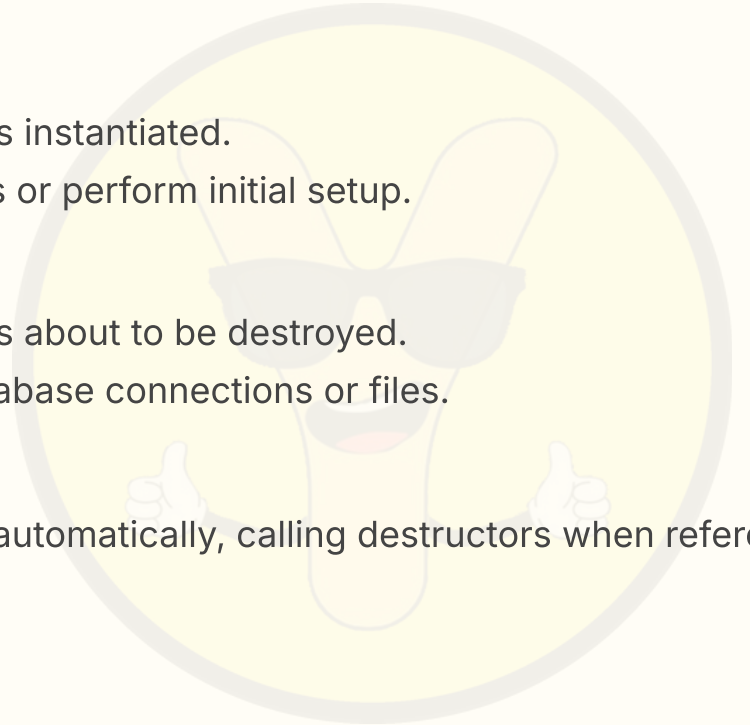


- The class Student defines state attributes name and roll.
- s1 is an instantiated object of the Student class.
- The display method uses the self reference to access object state.

Constructors and Destructors

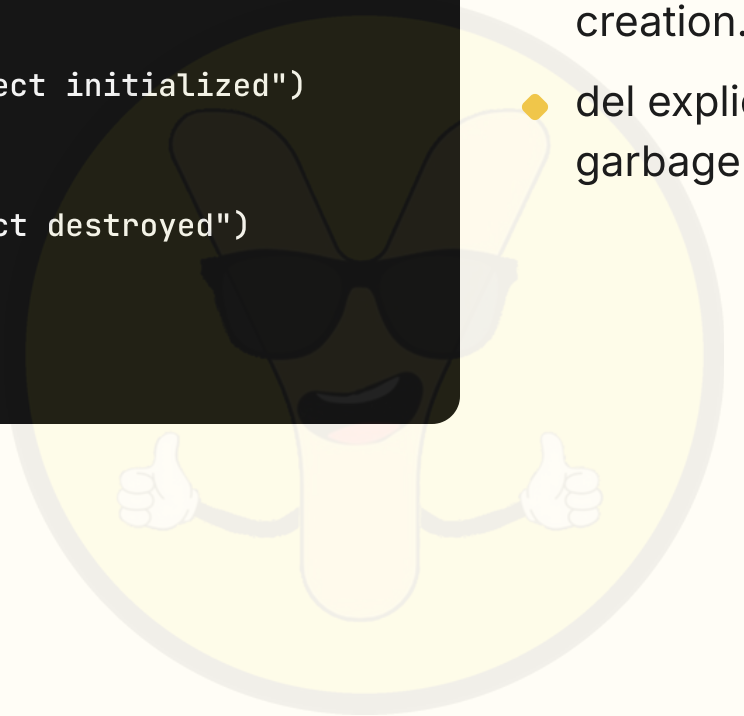
Special member methods for initialization and resource cleanup.

- __init__() Constructor
  - Invoked automatically when an object is instantiated.
  - Used to assign values to data members or perform initial setup.
- __del__() Destructor
  - Invoked automatically when an object is about to be destroyed.
  - Used for cleanup tasks like closing database connections or files.
- Garbage Collection
  - Python handles memory management automatically, calling destructors when reference count drops to zero.



Constructors and Destructors Example

```
class Demo:
    def __init__(self):
        print("Constructor called: Object initialized")

    def __del__(self):
        print("Destructor called: Object destroyed")

d = Demo()
del d
```

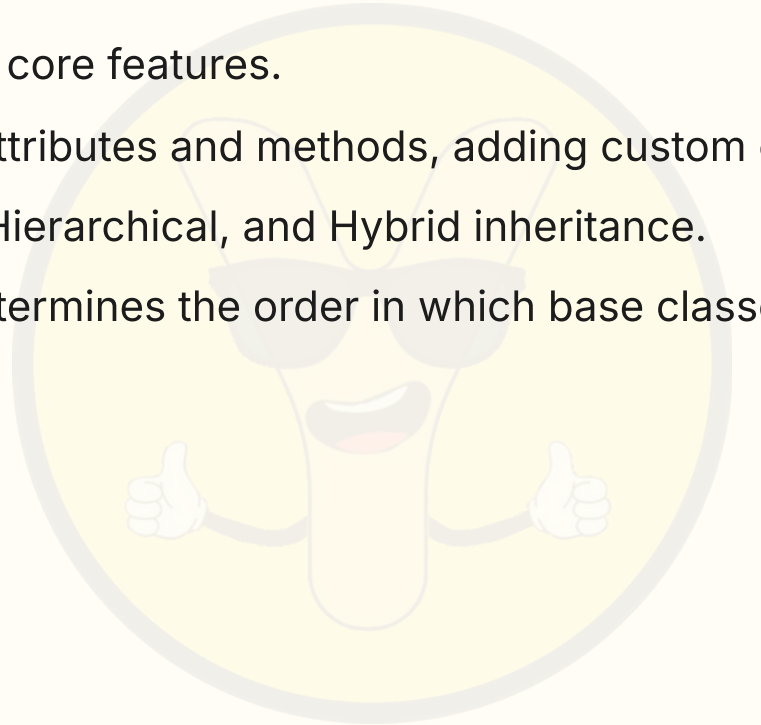


- __init__ executes instantly upon object creation.
- del explicitly triggers the destructor before garbage collection.

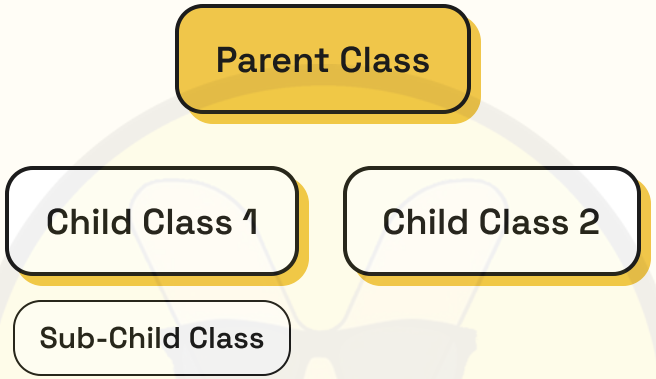
Inheritance

Mechanism of deriving a new class from an existing class to promote code reuse.

- Base class (Parent/Super) provides core features.
- Derived class (Child/Sub) inherits attributes and methods, adding custom extensions.
- Types: Single, Multiple, Multilevel, Hierarchical, and Hybrid inheritance.
- Method Resolution Order (MRO) determines the order in which base classes are searched.



Types of Inheritance Hierarchy

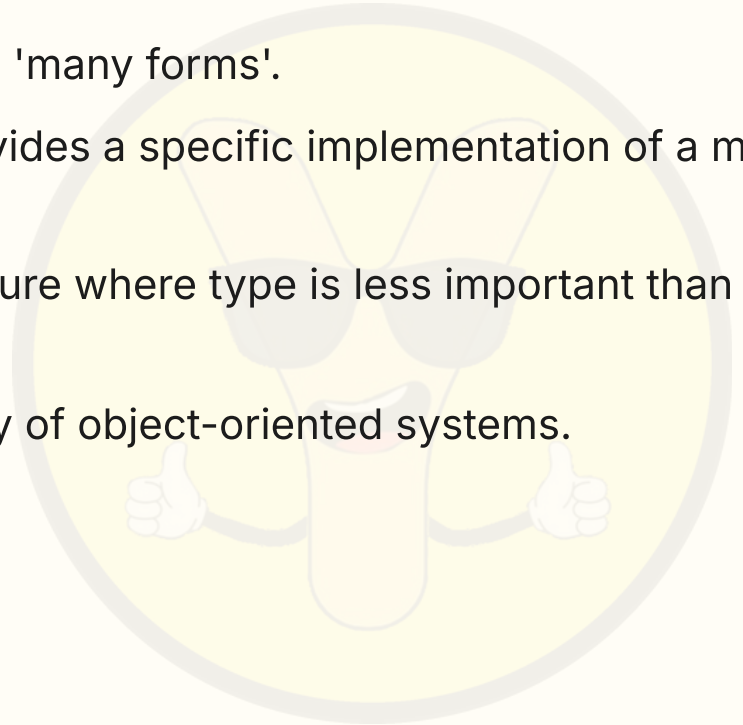


Tree hierarchy showing single, multilevel, and hierarchical inheritance structures.

Polymorphism

Ability of different classes to respond to the same message or method call in different ways.

- Derived from Greek words meaning 'many forms'.
- Method Overriding: Child class provides a specific implementation of a method already defined in its parent class.
- Duck Typing: Python's dynamic nature where type is less important than the presence of methods and properties.
- Enhances flexibility and extensibility of object-oriented systems.



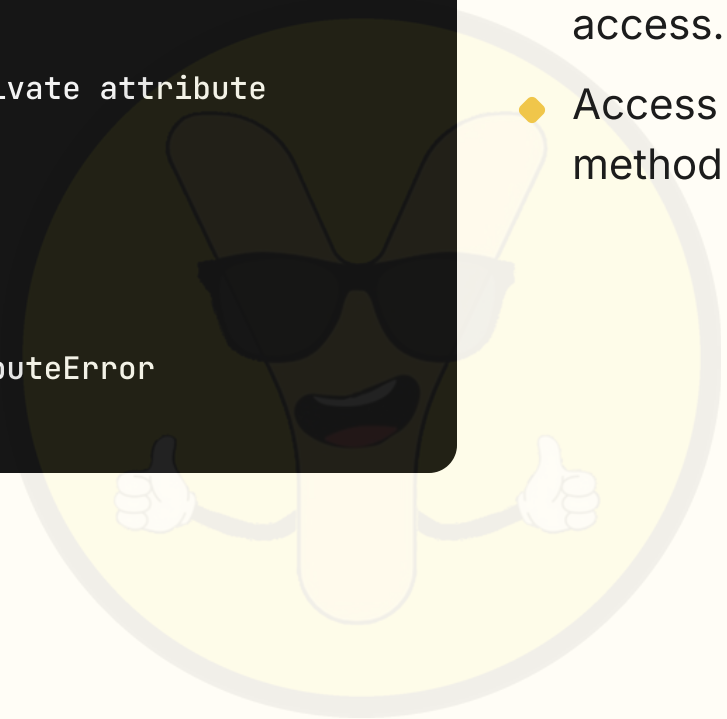
Encapsulation and Data Hiding

Bundling data and methods into a single unit while restricting direct access to components.

- Encapsulation
  - Protects internal state by preventing external code from directly modifying object attributes.
- Public Members
  - Accessible from anywhere outside the class (standard naming).
- Private Members
  - Prefixed with double underscore (__), restricting access to inside the class via name mangling.
- Getters and Setters
  - Controlled interfaces to read and modify private attributes safely.

Data Hiding Implementation

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

    def get_balance(self):
        return self.__balance

acc = Account(5000)
# print(acc.__balance)  # Raises AttributeError
print(acc.get_balance())
```

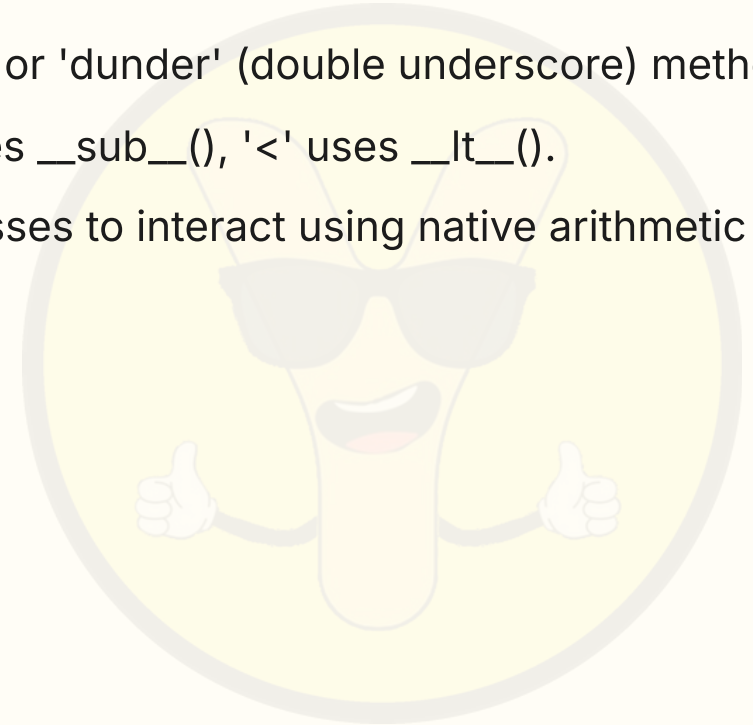


- __balance is hidden from direct external access.
- Access is safely granted through the getter method get_balance.

Operator Overloading

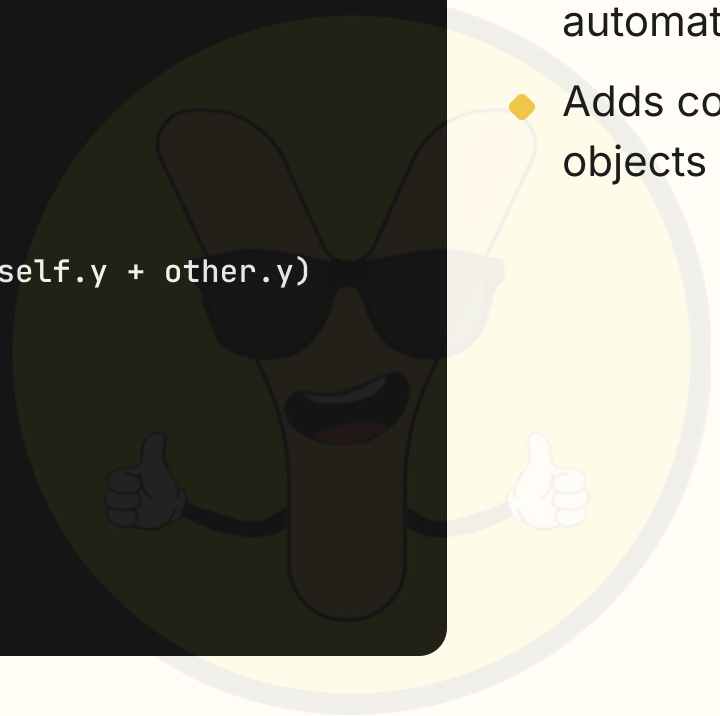
Giving standard operators custom meaning when applied to user-defined class instances.

- Implemented using special 'magic' or 'dunder' (double underscore) methods.
- Example: '+' uses __add__(), '-' uses __sub__(), '<' uses __lt__().
- Allows objects of user-defined classes to interact using native arithmetic and relational operators.



Operator Overloading Example

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

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

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

p1 = Point(1, 2)
p2 = Point(3, 4)
print(p1 + p2)
```



- The + operator invokes the __add__ method automatically.
- Adds corresponding coordinates of two Point objects and returns a new Point.

File I/O: Opening, Reading, and Writing

Persistent data storage mechanism interacting with secondary memory systems.

- Opening Files
  - Syntax: file_obj = open('filename', 'mode').
  - Modes: 'r' (read), 'w' (write), 'a' (append), 'b' (binary).
- Reading Files
  - Methods: read(), readline(), readlines() to fetch text content.
- Writing Files
  - Methods: write() and writelines() to store output data.
- Context Manager (with)
  - Ensures files are closed automatically after block execution completes.

File Handling Operations

```
with open("test.txt", "w") as f:
    f.write("Hello AKTU students\nLearning Python File I/O")

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

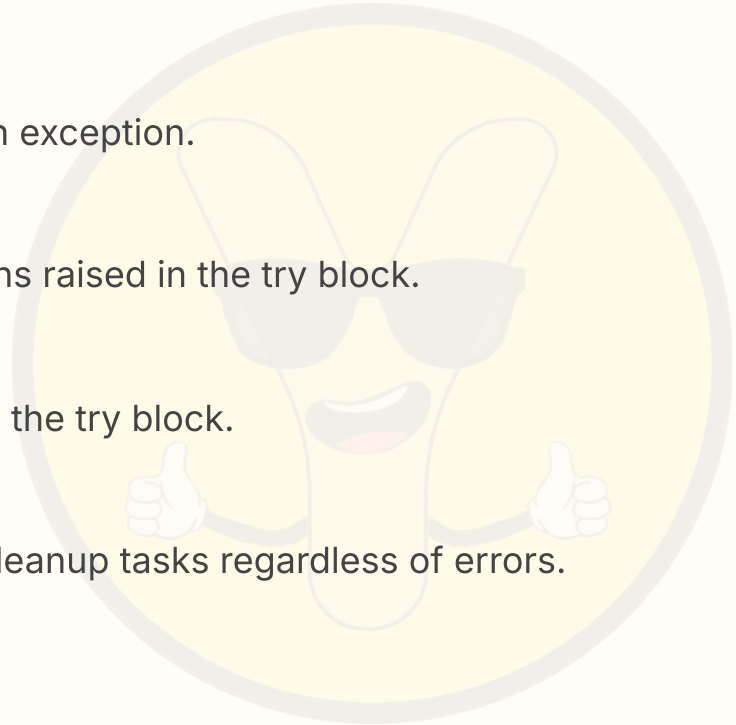


- The 'with' statement safely handles resource allocation and cleanup.
- Data is written to test.txt and subsequently read back onto the console.

Exception Handling Architecture

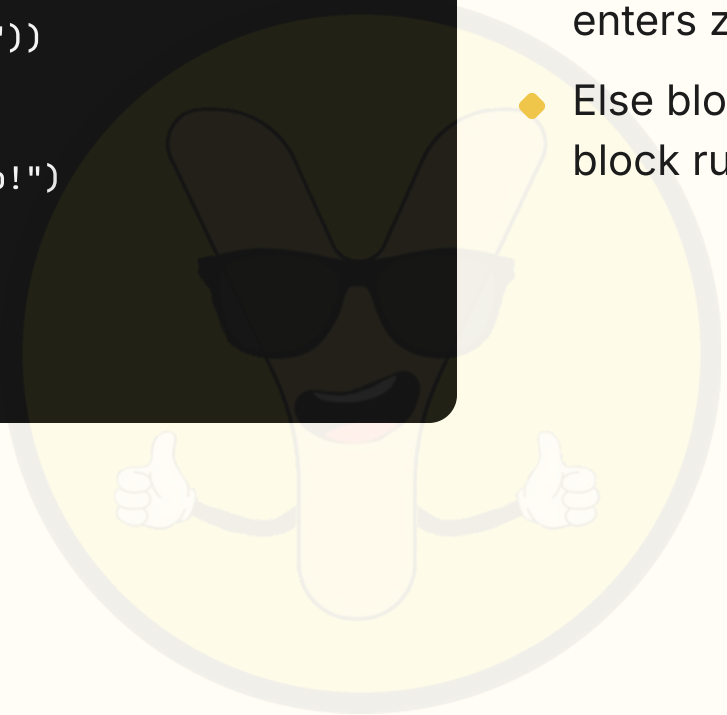
Handling runtime errors gracefully to prevent abrupt program termination.

- try block
  - Contains statements that might raise an exception.
- except block
  - Catches and handles specific exceptions raised in the try block.
- else block
  - Executes only if no exceptions occur in the try block.
- finally block
  - Guaranteed execution block used for cleanup tasks regardless of errors.



Exception Handling Structure

```
try:
    num = int(input("Enter numerator: "))
    res = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
else:
    print(f"Result: {res}")
finally:
    print("Execution complete.")
```

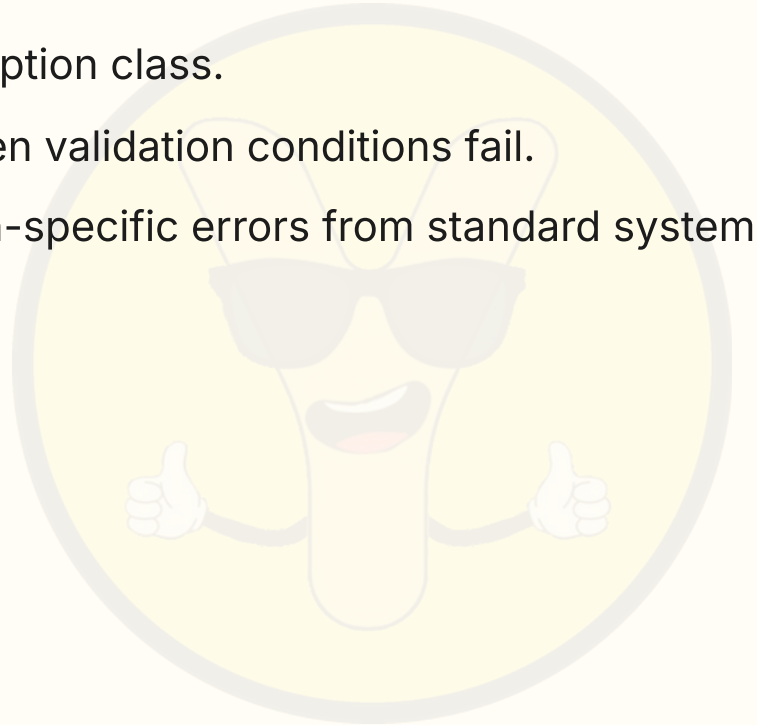


- Captures ZeroDivisionError safely if user enters zero.
- Else block runs if division succeeds; finally block runs unconditionally.

Custom Exceptions

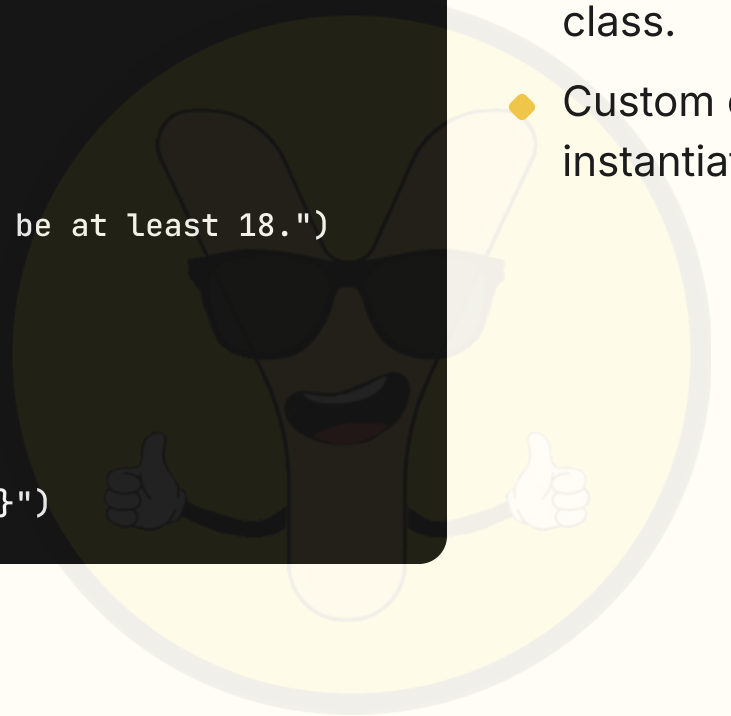
Creating user-defined exception classes to address specific application business logic errors.

- Derived from Python's built-in Exception class.
- Raised using the raise keyword when validation conditions fail.
- Enables clean separation of domain-specific errors from standard system errors.



Custom Exception Implementation

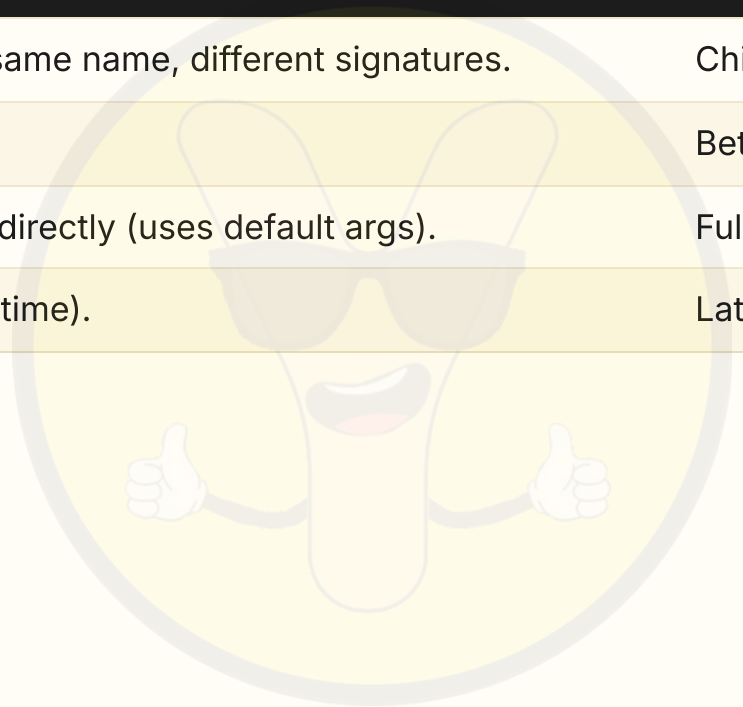
```
class InvalidAgeError(Exception):
    pass

def verify_age(age):
    if age < 18:
        raise InvalidAgeError("Age must be at least 18.")
    return "Access Granted"

try:
    print(verify_age(15))
except InvalidAgeError as e:
    print(f"Custom Exception Caught: {e}")
```



- InvalidAgeError inherits from Exception base class.
- Custom error message is passed during instantiation and caught in the except block.

Comparison: Method Overloading vs Method Overriding

Feature	Method Overloading	Method Overriding
Definition	Multiple methods with same name, different signatures.	Child class defines a parent method again.
Scope	Within the same class.	Between parent and child classes.
Python Support	Not natively supported directly (uses default args).	Fully supported via inheritance.
Binding	Early binding (Compile-time).	Late binding (Run-time).



Exam corner

- **Q1** Explain the core pillars of Object-Oriented Programming with suitable Python examples.
- **Q2** What is method overriding? Differentiate between method overloading and method overriding.
- **Q3** Explain operator overloading in Python. Write a program to overload the '+' operator for a custom Vector class.
- **Q4** Discuss data hiding and encapsulation. How does Python implement private attributes using name mangling?
- **Q5** Explain various file opening modes in Python. Write a program to copy contents from one file to another.
- **Q6** Describe the role of try, except, else, and finally blocks in exception handling with a clear code example.
- **Q7** What are custom exceptions? Write a Python script to create a user-defined exception for bank balance shortfall.

Unit 3 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 4 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

