



Studybeesofficial

Where Minds Pollinate

Class 12
Computer 

Chapter 3
**Programming
Fundamentals**
notes

Follow Our Socials



studybeesofficial



studybeesofficialpak



studybeesofficialpak

Programming Fundamental

```
... object to mirror
mirror_mod.mirror_object
operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
    operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True
... end -add ...
...=1
...objects.active
...+ str(modifier)
...b.select = 0
...context.selected_obj
...data.objects[one.name].select
print("please select exactly
--- OPERATOR CLASSES ---
types.Operator):
    X mirror to the selected
    object.mirror_mirror_x"
    mirror X"
... is not
```


What is Python?



- High-level, interpreted programming language



- Created by Guido van Rossum in 1991



- Emphasizes code readability and simplicity



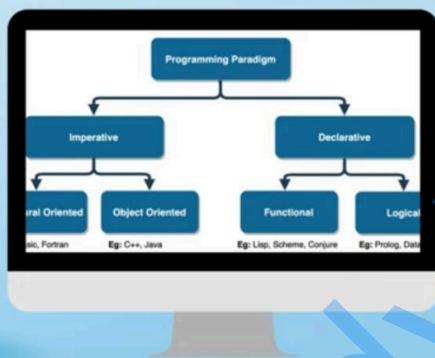
- Widely used in web development, data science, automation, AI, etc.

Python



- General purpose: build anything
- Open source! Free!
- Python packages, also for data science
 - Many applications and fields

What is a Programming Paradigm?



- A **programming paradigm** is a **way** to write and organize programs

It helps in managing:

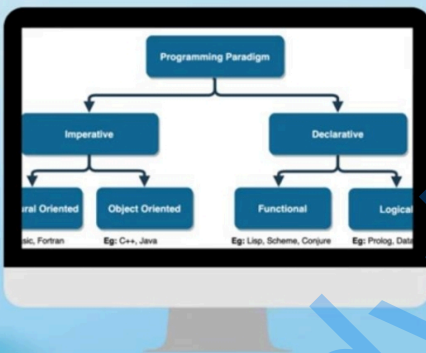
- Code structure
- Program flow
- Data handling

It tells how a programming language should be used to solve problems

Examples of programming languages:

Python, Java, SQL

Purpose of Programming Paradigms



- Helps **organize code** in a clean and simple way
- **Provides techniques** to solve problems easily
- **Helps programmers** think clearly and logically while writing code.
- Makes coding easier and more effective
- Supports good habits like **reusing and structuring code**
- **Makes teamwork easier** in software projects
- Matches real-world situations (like objects in OOP)

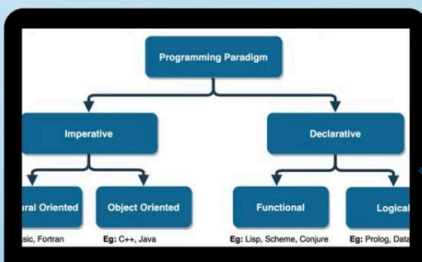
Types of Programming Paradigms

◆ Functional Programming

A programming paradigm that solves problems by dividing them into small, reusable functions.

◆ Object-Oriented Programming (OOP)

A programming paradigm that uses objects and classes to model and solve real-world problems.



What is Functional Programming?



- A way of **writing programs** using **functions**
- Focuses on **what to solve**, not step-by-step instructions
- The program is **divided** into small functions
- Each function **solves a specific part** of the problem
- Good for **small and medium applications**

Programming Paradigms

IMPERATIVE

How?

[12, 45, 9, 33, 67, 21]

```
max_num = numbers[0]
for n in numbers:
    if n > max_num:
        max_num = n
```

Maximum number is: 67

FUNCTIONAL

What?

[12, 45, 9, 33, 67, 21]

```
max(numbers)
```

Maximum number is: 67

Pros (Advantages)

1. Simpler code

Programs written in functional style are usually shorter and easier to read.

2. Code is easier to track for logic and debug

Since logic is broken into small independent functions, it's easier to find mistakes.

3. Functions are easier to test and run

Each function works on its own, so you can test it without worrying about the rest of the program.

Pros (Advantages)

4. Naturally supports parallel and concurrent programming

Multiple functions can run at the same time (parallel), which makes programs faster on modern multi-core computers.

5. Emphasizes on what to compute rather than how

You just say what result you want, not the step-by-step process. This makes the code shorter and clearer.

Cons (Disadvantages)

1. Not easy for beginners. Needs practice

New programmers may find it confusing because it's very different from normal step-by-step(imperative) coding.

2. May involve overhead with large data

Sometimes handling big lists or structures can be slower or use more memory.

3. Complex algorithms may be harder to express

For tough problems, writing code using only functions can be tricky, often requiring advanced techniques like higher-order functions.

Cons (Disadvantages)

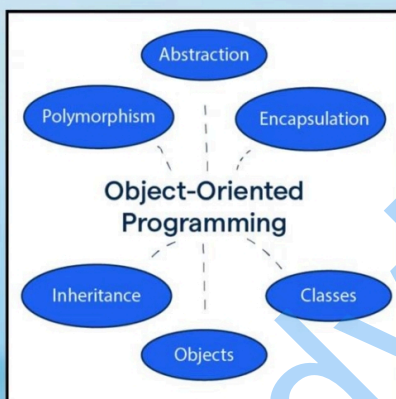
4. Requires careful handling of data sharing & synchronization

- If many functions run at the same time, making them share data safely is difficult.

5. May be less efficient in performance-critical scenarios

- In tasks where speed is extremely important (like real-time systems), functional programming may be slower than imperative style.

What is Object-Oriented Programming?

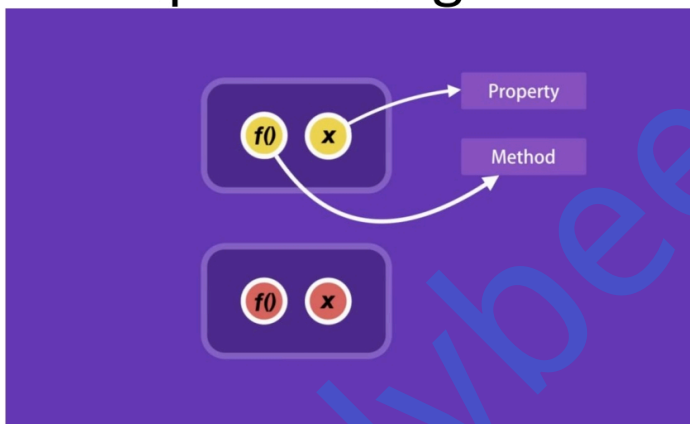


- **Based on objects**, which are like real-world items or entities
- Each **object** has data (**called attributes**)
- Each object also has **functions** (called methods)
- Focuses on data and how it behaves

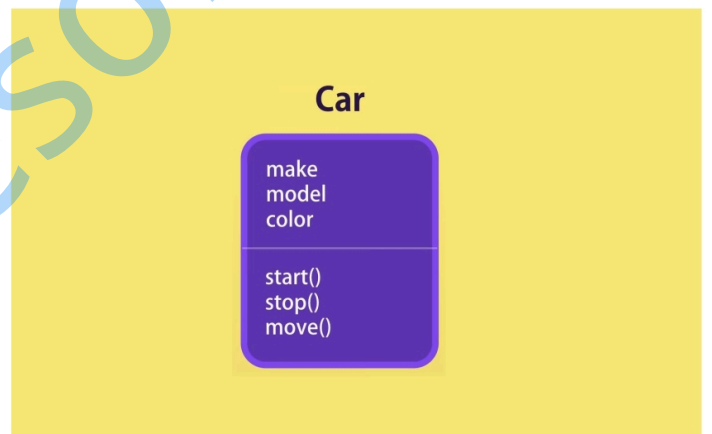
Example: A Car class may have:

- **Attributes:** color, speed, model
- **Methods:** start(), stop(), accelerate()

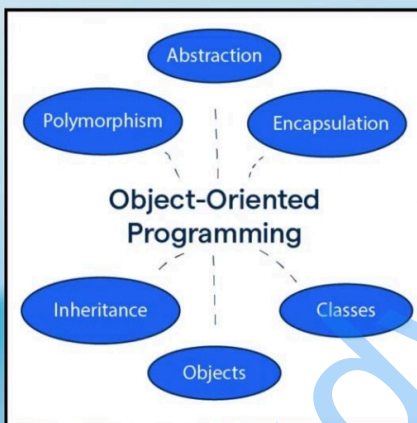
Class: Blueprint/Design



Car Example

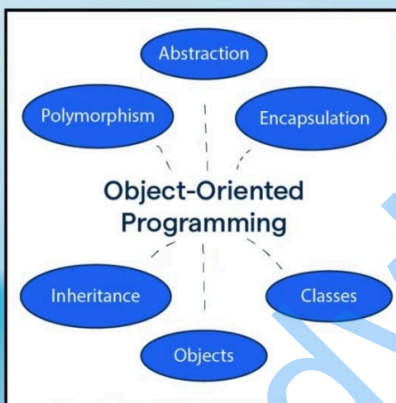


Benefits of OOP



- Helps **break down complex problems** into smaller parts
- Uses **encapsulation** to hide internal details and protect data
- Promote **code reusability** using classes and objects
- Makes it **easier to model** real-world problems in code
- Produces **safer and more reliable** programs

Drawbacks of Object-Oriented Programming (OOP)

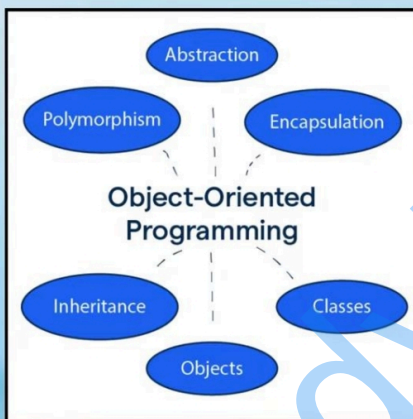


- May **increase program complexity** if not designed carefully
- Encapsulation can make debugging and error **tracing more difficult**
- Using inheritance excessively can make the code structure **rigid** and **less flexible** for future changes
- **Polymorphism** adds complexity
- **Not ideal for** high-performance computing tasks

Functional Programming vs Object-Oriented Programming)

Characteristics	Functional Programming	Object-Oriented Programming
Basic Idea	Focuses on writing programs using functions	Focuses on creating objects that combine data and methods
Approach	Describes what to do	Describes how to model real-world entities
Structure	Program is divided into small, pure functions	Program is organized using classes and objects
Ease of Understanding	May be harder for beginners	Often easier to relate to real-world concepts
Best Use Case	Good for mathematical and small/medium tasks	Good for complex systems and real-world modeling
Example	A function to calculate square of a number	A Car object with attributes like speed and methods like drive()

What is Object-Oriented Programming?

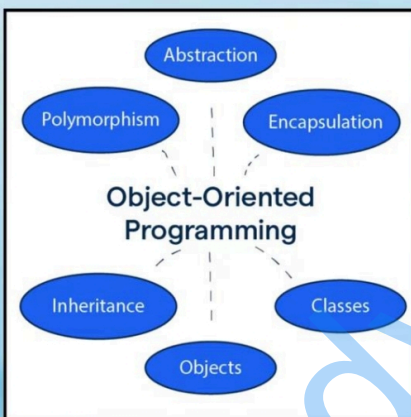


- **Object-Oriented Programming (OOP)** helps to structure complex programs.
- **Real-world entities** (e.g., person, car, bank account) are modeled as objects.

Objects contain:

- **Attributes** (data/properties)
- **Methods** (functions/behaviors)
-

Class and Object



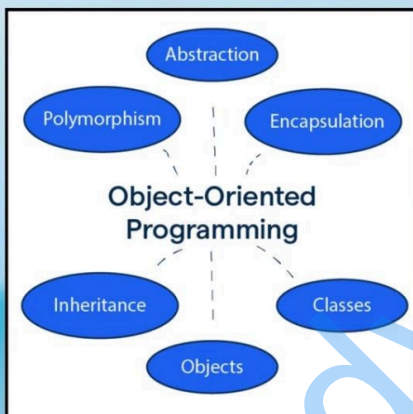
- **Class** is a **blueprint/template** for creating objects.
- An **object** is an **instance** of a **class**.

Each object:

- Shares class methods
- Has unique attribute values

One class can create many objects

Example – StationaryItem Class



Class: StationaryItem

Attributes:

- name
- price
- color
- brand

Object	Name	Price	Color	Brand
Object 1	Pen	200	Blue	XYZ
Object 2	Pencil	100	Red	ABC

Example Methods:

- get_price()
- set_price()

Enables interaction and management of object data.

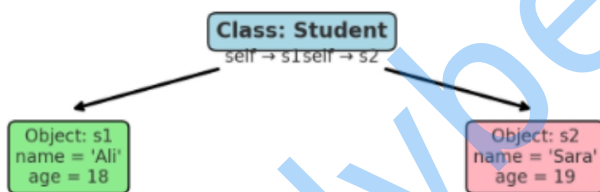
Constructor:

- A **constructor** is a **special method in Object-Oriented Programming (OOP)** that is **automatically called when an object is created** from a class.
- Its main purpose is to **initialize the object's attributes (variables)** when the object is created.

In Python, the constructor is defined using the special method:

```
python  
  
def __init__(self, ...):  
    # initialization code
```


Constructor:



- In Python, self is a **reference to the current object** that is being created or used.
- When you call a constructor (or any method inside a class), **Python automatically passes the object itself as the first argument** — by convention, we call this self.

ACTIVITY:

- Class = Student (attributes: name, roll_no, grade)
- Class = Book (attributes: title, author, year)
- Class = Mobile (attributes: brand, model, price)

Task for each group:

1. Write a Python class with an `__init__` function.
2. Create at least 2 objects from the class.
3. Print the attributes of objects.

Questions:

- What attributes did you define in your class and why?
- What role did the `__init__` function play in your program?
- If you add more objects, do you need to rewrite the class? Why/why not?

Class : Person

Attributes/Properties / Data

- **name** → Person's full name
- **age** → Age in years
- **gender** → Male, Female, Other
- **height** → in cm or ft
- **weight** → in kg
- **nationality** → Country of origin
- **blood_group** → e.g., A+, O-
- **address** → Place of residence
- **phone_number** → Contact number
- **email** → Email address

Behaviors / Methods

- **eat()** → Person eats food
- **sleep()** → Person goes to sleep
- **walk()** → Person walks
- **talk()** → Person speaks
- **work()** → Person performs work/task
- **study()** → Person studies/learns
- **introduce()** → Prints person's introduction (name, age, nationality, etc.)
- **celebrate_birthday()** → Increases age by 1

Class : Smartwatch

Attributes/Properties / Data

- **brand** → e.g., Apple, Samsung, Fitbit
- **model** → e.g., Galaxy Watch 6, Apple Watch Series 9
- **color** → e.g., black, silver, rose gold
- **batteryLife** → in hours/days (e.g., 24 hours, 7 days)
- **weight** → in grams
- **strapMaterial** → leather, silicone, metal
- **storageCapacity** → e.g., 16GB, 32GB

Behaviors / Methods

- **showTime()** → Displays current time and date
- **trackSteps()** → Counts daily steps
- **monitorHeartRate()** → Measures heart rate
- **receiveNotifications()** → Shows calls, messages, app alerts
- **makeCall()** → Makes a call via Bluetooth
- **sendMessage()** → Sends a text/voice message

Class : Marker

Attributes/Properties / Data

- **brand** → e.g., Dollar, Piano, Sharpie
- **color** → ink color (black, blue, red, green)
- **ink_level** → remaining ink percentage
- **material** → plastic, metal, etc.
- **length** → size of the marker
- **cap_present** → True/False (whether the cap is on)
- **refillable** → True/False
- **type** → permanent, whiteboard, highlighter

Behaviors / Methods

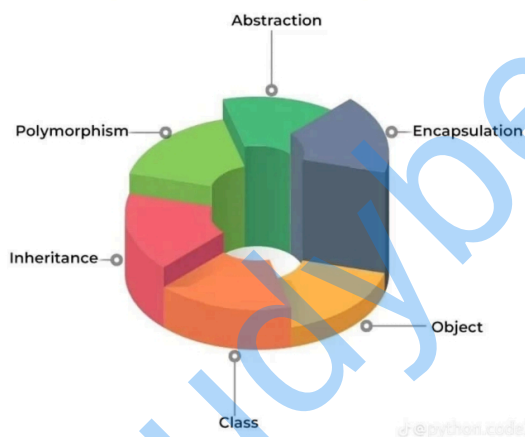
- **write(text)** → Writes given text on a surface
- **draw(shape)** → Draws a shape (circle, line, etc.)
- **open_cap()** → Removes the cap
- **close_cap()** → Puts the cap back on
- **check_ink_level()** → Shows remaining ink
- **refill(amount)** → Refills the marker with ink
- **change_color(new_color)** → Changes ink color (if refillable)

Practice Programs

1. Create a Book class with **title, author, and price**.
Add a method to display book details.
2. Create a Rectangle class with **length and width**.
Add methods to calculate **area**.
3. Create a Car class with **brand, model, and year**.
Add a method to print car details in one line.

OOPS

in real life



Encapsulation



A college can have many departments like CS Department, Accounts Department, etc. All these departments together makes a college.

@python.codes31

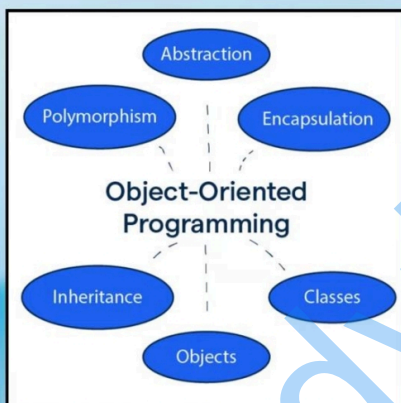
Encapsulation

Encapsulation: Hiding internal data, exposing only needed methods.

- Data + methods are packed in one unit (class).
- Direct access is restricted and is provided via specific methods

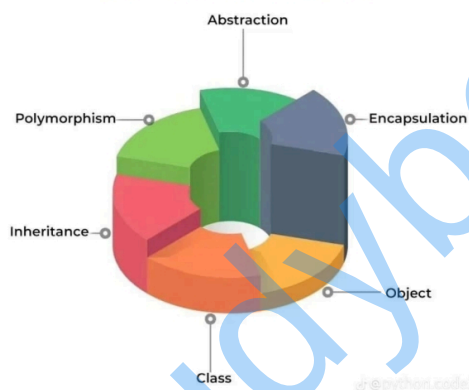
Bank Account Example:

- Use deposit(), withdraw() instead of direct balance change
- get_balance() to view balance info safely



OOPS

in real life



Abstraction



With a phone you can do many things like phone calls, messages, play games etc.

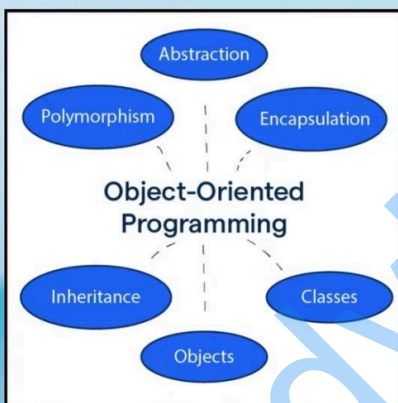
It doesn't show the inside process of how its doing the things. implementation parts are hidden.

Abstraction

Abstraction means Show only necessary details, hide complexity

Example:

- **Driving a car:** You use start(), stop(), accelerate()
- But internal working (engine, gear system) is hidden



Inheritance

Inheritance is the process of creating a new class (called a subclass or derived class) based on an existing class (called a parent class or base class).

Example:

Parent Class: Vehicle

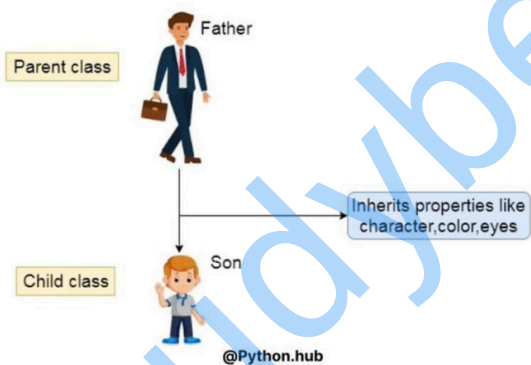
- **Attributes:**
- speed, color, wheels
- **Methods:**
- accelerate(), brake()

Derived Class: Car inherits from Vehicle

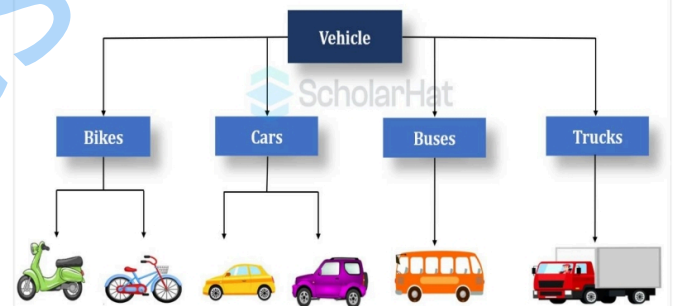
- Inherits all properties and methods from Vehicle
- Adds its own:
- + seatingCapacity
- + openSunroof()

INHERITANCE

Inheritance means it allows classes to inherit common properties from the parent class.



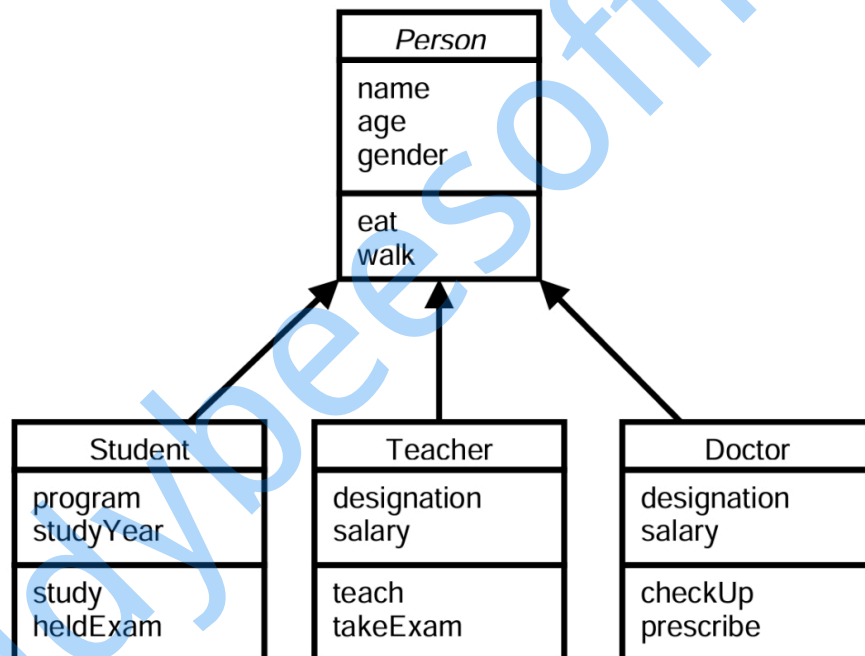
Inheritance



Student
name age gender program studyYear
study heldExam eat walk

Teacher
name age gender designation salary
teach takeExam eat walk

Doctor
name age gender designation salary
checkUp prescribe eat walk



Polymorphism

Polymorphism = **One interface, many forms**

Example: Shape Class

Common Interface:

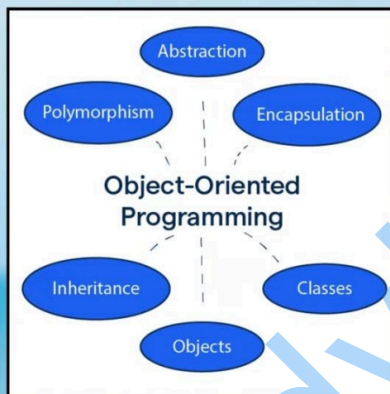
All shapes have:

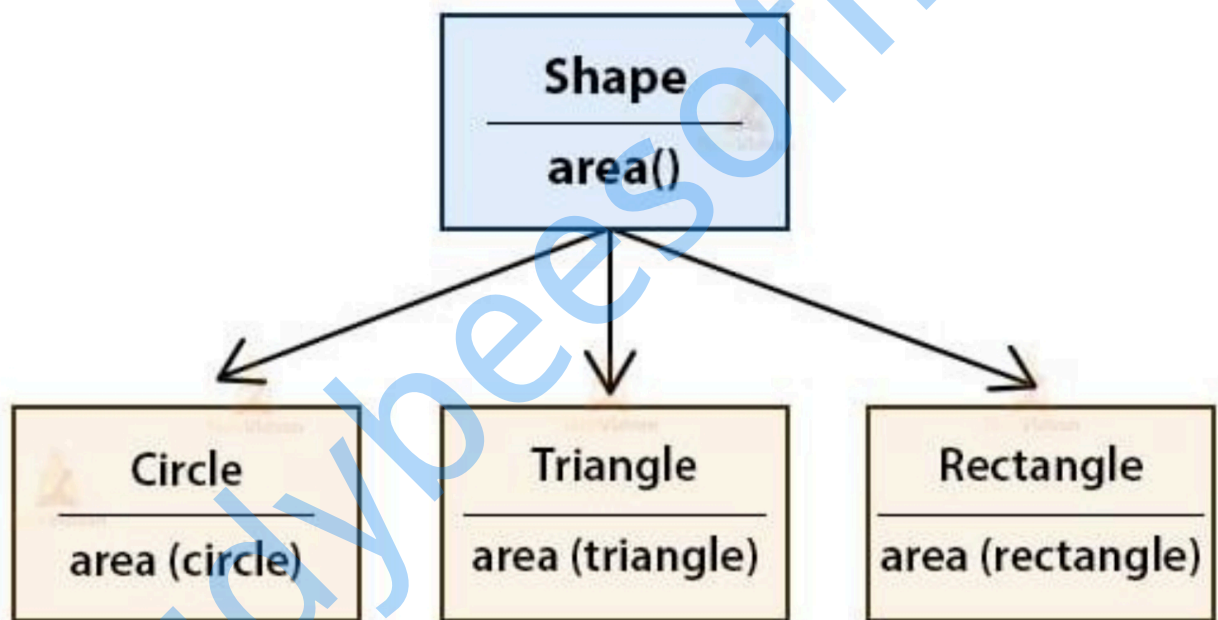
- draw()
- calculate_area()

Different Implementations:

- Triangle → calculates area using $\text{base} \times \text{height}$
- Rectangle → uses $\text{length} \times \text{width}$
- Circle → uses $\pi \times \text{radius}$

Same command → different behavior based on object type





Inheritance



Both are cars, but one has extra features.

Polymorphism



Like he is an actor, YouTuber, doctor, son, brother etc.



A man can have many roles like actor, doctor, Astronaut, son etc.

@python.codell

Same person can have different roles.

4 Pillars of OOP

1. Encapsulation (Data Hiding)

- Definition: Binding data and methods together, restricting direct access.
- **Real-life Example:** A capsule medicine hides the drug inside. Similarly, a mobile phone hides its circuit but provides buttons/apps to operate.

2. Abstraction (Hiding Complexity, Showing Essentials)

- Definition: Showing only necessary details while hiding the background processes.
- **Real-life Example:** ATM machine – user just inserts card & enters PIN, while complex banking operations are hidden.

4 Pillars of OOP

3. Inheritance (Reusability)

- Definition: One class acquires properties/methods of another class.
- **Real-life Example:** Child inherits traits from parents. In software, a “Car” class can inherit properties from a general “Vehicle” class.

4. Polymorphism (Many Forms)

- Definition: Same function or operation behaves differently based on context.
- **Real-life Example:** A person can be a student in class, a customer in a shop, and a player on a field. Same person, different roles.

Pillar	Benefits
Encapsulation	- Protects data by restricting direct access (data hiding)- Improves security of the program- Makes code easier to maintain and modify
Abstraction	- Hides system complexity from the user- Focuses on essential features only- Makes systems easier to use and understand- Helps in reducing programming errors
Inheritance	- Promotes code reusability- Reduces redundancy in code- Makes programs easier to extend- Establishes relationships between classes (parent-child)
Polymorphism	- Provides flexibility (same method can perform different tasks)- Improves code readability and organization- Makes code more scalable-

SUMMARY & CONCLUSION

- **Class & Object: Template & instance**
- **Encapsulation: Data hiding**
- **Abstraction: Simplified interface**
- **Inheritance: Reuse & extend**
- **Polymorphism: Common commands, customized behavior**



Python Collections

There are four collection data types in the Python programming language:

List is a collection which is ordered and changeable. Allows duplicate members.

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

Set is a collection which is unordered, unchangeable, and unindexed. No duplicate members.

Dictionary is a collection which is ordered and changeable. No duplicate members.

PYTHON LIST



```
[1, 2, "apple"]
```



Python Lists



Access List Item



Change List Item



Loop List



Sort, Copy



Sort, Copy

Introduction to Lists

- A list is a collection of items in a particular order.
- Lists are created using square brackets [].
- Lists are ordered, mutable collections.
- Can store items of different data types.
- Example:
 - `my_list = [10, "apple", 3.14, True]`

PYTHON LIST METHODS

PROGRAMMINGVALLEY.COM

   **.append()**     

   **.clear()**

   **.copy()**   

   **.count()**  **2**

   **.index()**  **2**

   **.insert(1,**  **)**    

   **.pop(2)** 

   **.remove()**   

   **.reverse()**   

Access List Items

- Use index to access an item (starting from 0).
- Use negative indexing for reverse order.
- Example:
 - `my_list = ["a", "b", "c"]`
 - `print(my_list[0])` # 'a'
 - `print(my_list[-1])` # 'c'

Change List Items

- Modify values using index.
- Example:
 - `my_list = [1, 2, 3]`
 - `my_list[1] = 200`
 - `print(my_list)` # [1, 200, 3]

Add List Items

- Use **append()** to add to the end.
- Use **insert()** to add at a specific position.
- Example:
 - `my_list = [1, 2, 3]`
 - `my_list.append(4)`
 - `my_list.insert(1, 100)`
 - `print(my_list)` # `[1, 100, 2, 3, 4]`

Remove List Items

- Use **remove()** to delete a value.
- Use **pop()** to remove by index.
- **del**: Delete item at index or the entire list
- **clear()**: Removes all items
- Example:
 - `my_list = ["a", "b", "c"]`
 - `my_list.remove("b")`
 - `my_list.pop(0)`
 - `del my_list[0]`
 - `print(my_list) # []`
 - `my_list.clear()`

Loop Through a List

- Use **for loop** to iterate.
- Use **while** with index.

- Example:

- for item in [1, 2, 3]:
- print(item)

- lst = ["x", "y"]
- i = 0
- while i < len(lst):
- print(lst[i])
- i += 1

Sort List

- Use **sort()** for ascending (or descending with `reverse=True`).
- Example:
 - `nums = [3, 1, 4, 2]`
 - `nums.sort()`
 - `print(nums)` # [1, 2, 3, 4]

Copy a List

- Use **copy()** to copy.
- Example:
 - `original = ["a", "b"]`
 - `copied = original.copy()`
 - `print(copied)`

Conclusion

- Lists are flexible, powerful data structures.
- Learn to manipulate them using various operations.
- Quick Quiz:
 - 1. How do you add an item at index 2?
 - 2. What does `list[-1]` return?

What is a Tuple?

- A tuple is an ordered, immutable collection in Python.
- Defined using parentheses ()
- Elements can be of mixed data types.
- Example:
 - `tuple1 = (10, "apple", 3.14, True)`
 - `print(tuple1)`

Access Tuple Items

- Access elements using index (starting from 0)
- Negative indexing is allowed
- Example:
 - `tuple1 = ("a", "b", "c")`
 - `print(tuple1[0])` # 'a'
 - `print(tuple1[-1])` # 'c'

Update Tuple

- Tuples are immutable (cannot change items directly)
- Convert to list, update, convert back to tuple
- Example:
 - `tuple1 = (1, 2, 3)`
 - `temp = list(tuple1)`
 - `temp[1] = 200`
 - `tuple1 = tuple(temp)`
 - `print(tuple1)`

Loop Through Tuple

- - Use for loop to iterate over tuple elements
- Example:
 - `tuple1 = ("red", "green", "blue")`
 - `for color in tuple1:`
 - `print(color)`

Join Tuples

- Use + operator to join two tuples
- Example:
 - tuple1 = (1, 2)
 - tuple2 = (3, 4)
 - tuple3 = tuple1 + tuple2
 - print(tuple3)

Tuple Methods

- `count(value)`: number of times value appears
- `index(value)`: first index of value
- Example:
 - `tuple1 = (1, 2, 3, 2, 4)`
 - `print(tuple1.count(2))`
 - `print(tuple1.index(3))`

Feature	List	Tuple
Mutability	Mutable: You can change, add, or remove elements.	Immutable: Once created, elements cannot be changed.
Syntax	Defined using square brackets <code>[]</code> .	Defined using parentheses <code>()</code> .
Methods	Supports methods like <code>append()</code> , <code>remove()</code> , <code>sort()</code> , etc.	Has fewer methods; no methods to modify elements.
Performance	Slower due to dynamic nature (mutable).	Faster because of immutability.
Use Case	Suitable for collections that change frequently.	Suitable for fixed collections of elements.
Example	<code>my_list = [1, 2, 3]</code>	<code>my_tuple = (1, 2, 3)</code>

Conclusion

- Tuples are like lists but immutable
- Use them when data should not be changed
- Quick Quiz:
 1. Can we modify a tuple directly?
 2. How do we access the last item in a tuple?
 3. What does count() method do?

What is a Set?

- A set is an unordered collection of unique elements.
- Defined using curly braces {}.
- No duplicate values allowed.
- Example:
 - `my_set = {1, 2, 3, 4}`
 - `print(my_set)`

Access Set Items

- Sets are unordered, so items cannot be accessed by index.
- Use loops to access elements.
- Example:
 - `my_set = {'apple', 'banana', 'cherry'}`
 - `for item in my_set:`
 `print(item)`

Add Set Items

- Use `add()` to add one item.
- Use `update()` to add multiple items.
- Example:
- `my_set = {'apple', 'banana'}`
- `my_set.add('cherry')`
- `my_set.update(['orange', 'mango'])`
- `print(my_set)`

Remove Set Items

- Use `remove()` or `discard()` to remove an item.
- Use `pop()` to remove a random item.
- Use `clear()` to empty the set.
- Example:
 - `my_set = {'apple', 'banana', 'cherry'}`
 - `my_set.remove('banana')`
 - `print(my_set)`

Loop Through a Set

- Use a for loop to iterate through a set.
 - Example:
 - `fruits = {'apple', 'banana', 'cherry'}`
 - `for fruit in fruits:`
 - `print(fruit)`
-

Join Sets

- Use `union()` or `|` to join sets.
 - Use `update()` to add elements from one set to another.
 - Example:
 - `set1 = {1, 2, 3}`
 - `set2 = {3, 4, 5}`
 - `set3 = set1.union(set2)`
 - `print(set3)`
-

Set Methods

- `union()`: Combine two sets.
 - `intersection()`: Items common to both sets '&'.
 - `difference()`: Items in one set but not the other.
 - Example:
 - `A = {1, 2, 3}`
 - `B = {3, 4, 5}`
 - `print(A.intersection(B))`
 - `print(A.difference(B))`
-

Practice Programs

1. Write a program to remove duplicate elements by using set.

```
lst = [1, 2, 2, 3, 4, 3, 5]
```

2. Write a program to find common elements in two lists by using set.

```
a = [1, 2, 3, 4]
```

```
b = [3, 4, 5, 6]
```

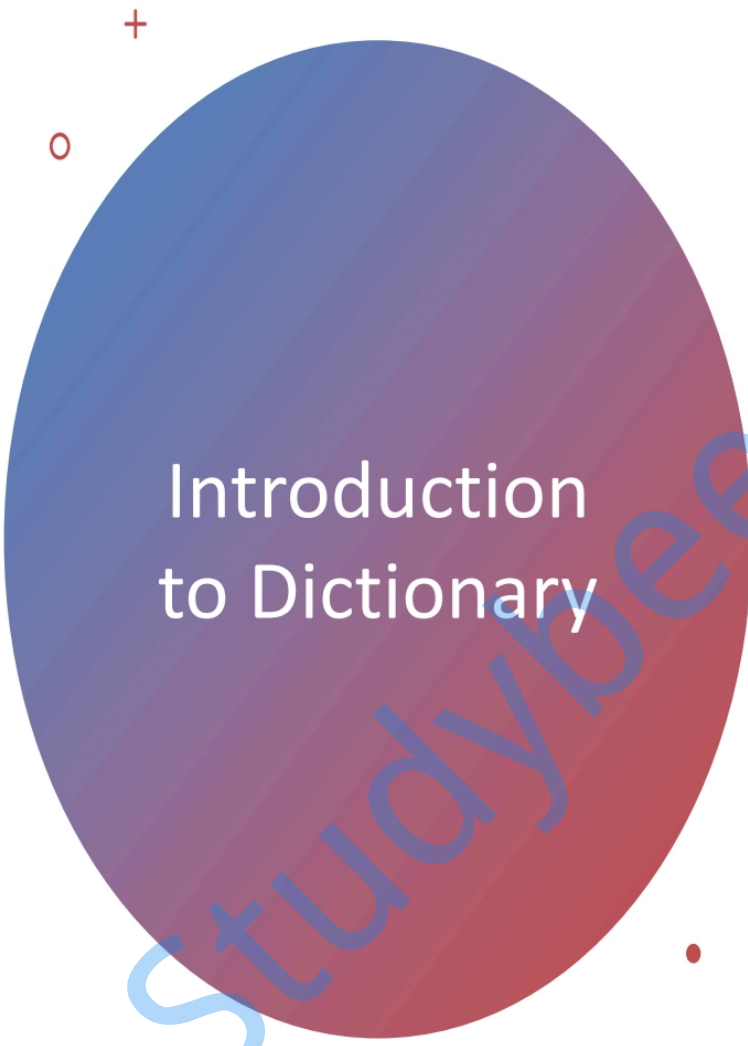
Conclusion

- Sets are useful for storing unique items.
 - Useful operations: union, intersection, difference, etc.
 - Quick Quiz:
 1. Can sets contain duplicate values?
 2. How do you add multiple items to a set?
 3. What does the intersection() method do?
-

Feature	List	Tuple	Set
Ordered	✓ Yes	✓ Yes	✗ No
Mutable	✓ Yes	✗ No	✓ Yes
Duplicates	✓ Allowed	✓ Allowed	✗ Not Allowed
Indexed Access	✓ Yes	✓ Yes	✗ No Indexing
Performance	Normal	✓ Faster	✓ Fast for lookup

Python Dictionary

Concepts, Methods, and Operations



Introduction to Dictionary

- A dictionary is a collection of key-value pairs.
 - Keys must be unique and immutable.
 - Values can be of any data type.
 - Defined using curly braces { }.
- 

Real-World Examples of Dictionary

1. Student Records

```
student = {'name': 'Ali', 'age': 20, 'roll_no': 'CS101', 'grade': 'A'}
```

2. Phonebook / Contacts

```
contacts = {'Ahmed': '03001234567', 'Sara': '03119876543'}
```

3. Word Meaning

```
dictionary = {'apple': 'a fruit', 'python': 'a programming language'}
```

4. E-commerce Product Data

```
product = {'id': 101, 'name': 'Laptop', 'price': 85000, 'stock': 25}
```

5. Employee Database

```
employee = {'emp_id': 501, 'name': 'Amir', 'designation': 'Teacher'}
```

Common Dictionary Methods

- `dict.keys()` – returns all keys
- `dict.values()` – returns all values
- `dict.items()` – returns key-value pairs
- `dict.get(key)` – returns value for key
- `dict.update(other_dict)` – updates dictionary
- `dict.pop(key)` – removes key-value pair
- `dict.popitem()` – removes last item
- `dict.clear()` – removes all items
- `dict.copy()` – returns shallow copy

Example Dictionary

```
student = {  
    'name': 'Ali',  
    'age': 20,  
    'course': 'CS'  
}
```

Accessing Values

- `print(student['name'])` # Ali
- `print(student.get('age'))` # 20

Adding Items

- `student['grade'] = 'A'`
- `print(student)`

Changing Items

- `student['age'] = 21`
- `print(student)`

Removing Items

- `student.pop('course')`
- `student.popitem()`
- `print(student)`

Looping through Dictionary

- for key in student:
 print(key, student[key])
- for key, value in student.items():
 print(key, value)

Copying Dictionary

- `student_copy = student.copy()`
- `print(student_copy)`

Nested Lists in Python

UNDERSTANDING MATRIX REPRESENTATION WITH
EXAMPLES

What is a Nested List?

- A nested list is a list inside another list.
- Used to represent matrices, tables, or 2D data.
- Example: `matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

Visual Matrix Representation

- 1 2 3
- 4 5 6
- 7 8 9
- This is a 3x3 matrix (3 rows $\times$ 3 columns).
- Each row is a list; the matrix is a list of lists.

Accessing Elements

- `matrix[0][0]` → 1
- `matrix[1][2]` → 6
- `matrix[row][column]` format is used.
- Indexing starts from 0.

Example - 2x3 Matrix

- `matrix = [[10, 20, 30], [40, 50, 60]]`
- Access: `matrix[1][1] → 50`
- Access: `matrix[0][2] → 30`

Looping Through a Matrix

- for row in matrix:
- for item in row:
- print(item, end=" ")
- print()
- This prints the matrix row by row.

Applications

- Storing matrices in math
- Image pixels (2D array)
- Game boards (like Tic-Tac-Toe)
- Tables and spreadsheets

Practice Activity

- Write a 3x2 matrix
- Access the last element
- Change value at `matrix[0][1]`

Summary

- A nested list is a list of lists.
- Used to represent 2D data.
- Access elements using `matrix[row][col]`.
- Useful in loops, conditions, and real-world data.

File Handling

- File should be handled as **binary or text mode**:
- **"t" - Text** - Default value. Text mode
- **"b" - Binary** - Binary mode (e.g. images)

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

File Handling

- Python File Open

```
✓ with open ("abc.txt","r") as file:  
✓     for char in file.read():  
|         print(char , end="")  
print("\nFile read character by character successfully ")
```

- Python Delete File

```
import os  
os.remove("demofile.txt")
```


Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

`"a"` - Append - will append to the end of the file

`"w"` - Write - will overwrite any existing content

```
with open ("abc.txt","w") as file:  
    file.write("The quick brown fox jumps over the lazy dog!!!!")  
print("File created and successfully ")
```

```
with open("abc.txt", "a") as f:  
    f.write("\nNow the file has more content!")  
print("Content Added!")
```


Creating GUIs with Python Tkinter

(Building User-Friendly Desktop Applications)

What is a GUI?

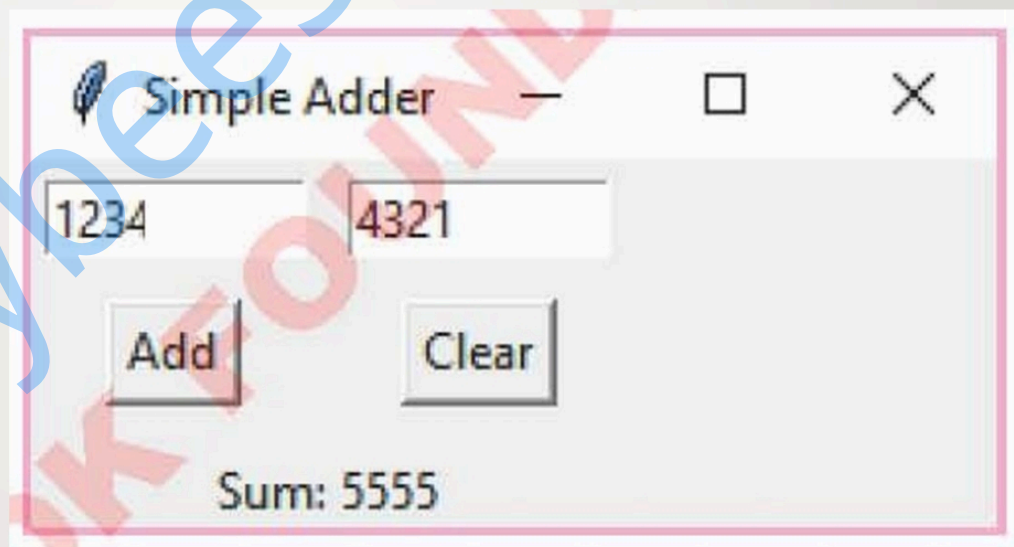
- **GUI (Graphical User Interface):** Uses visual components like windows, buttons, and menus.
- **vs. CLI (Command-Line Interface):** Requires text-based commands.
- **Advantages:** Improved usability, accessibility, and a smoother user experience.
- **Python's Built-in Solution:** The **Tkinter** library.

Introduction to Tkinter

- **Tkinter** is Python's standard library for creating Graphical User Interfaces.
- **Key Features:**
 - **Widgets:** Basic building blocks (buttons, labels, text boxes, checkboxes).
 - **Event Handling:** Responds to user actions (clicks, key presses).
 - **Customizable:** Widgets can be organized and modified to create desired layouts.
- **Use Cases:** Desktop apps (calculators, text editors), simple 2D games, and utilities.

Building a Simple App: The Number Adder (Logic)

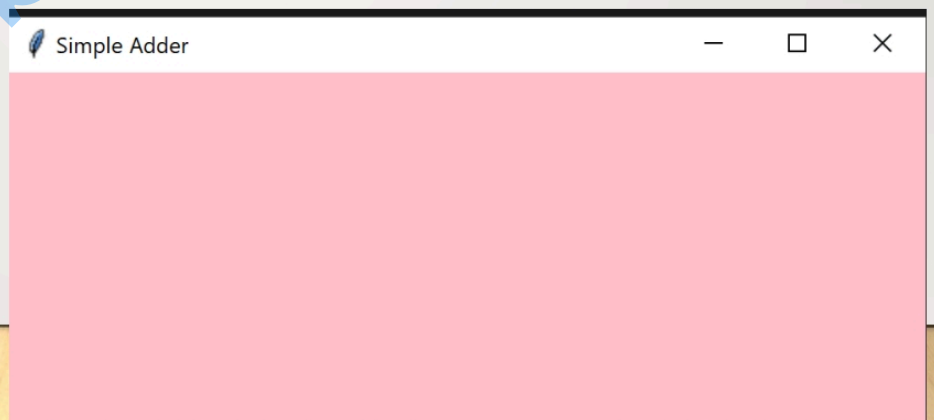
- **Goal:** An app that takes two numbers, adds them, and displays the result.



Import tkinter : Main Window

```
1 from tkinter import *
2 root = Tk()
3 root.title("Simple Adder")
4 root.minsize(width="500", height="300")
5 root.config(bg="pink")
6
7 root.mainloop()
```

Output:



Entry Box:

```
20 entry1 = Entry(root, width=10)
21 entry1.grid(row=0, column=0, padx=5, pady=5)
22
23 entry2 = Entry(root, width=10)
24 entry2.grid(row=0, column=1, padx=5, pady=5)
```

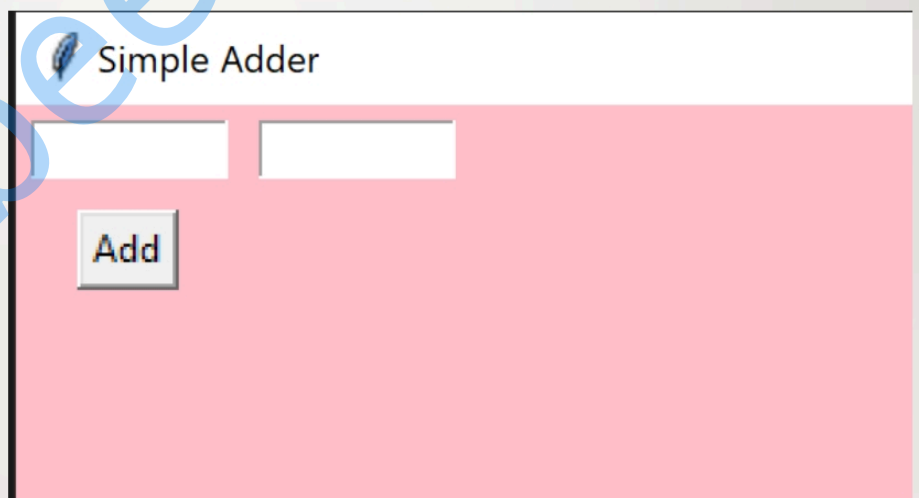
Output:



Button:

```
button1 = Button(root, text="Add", command=add_number)
button1.grid(row=1, column=0, padx=5, pady=5)
```

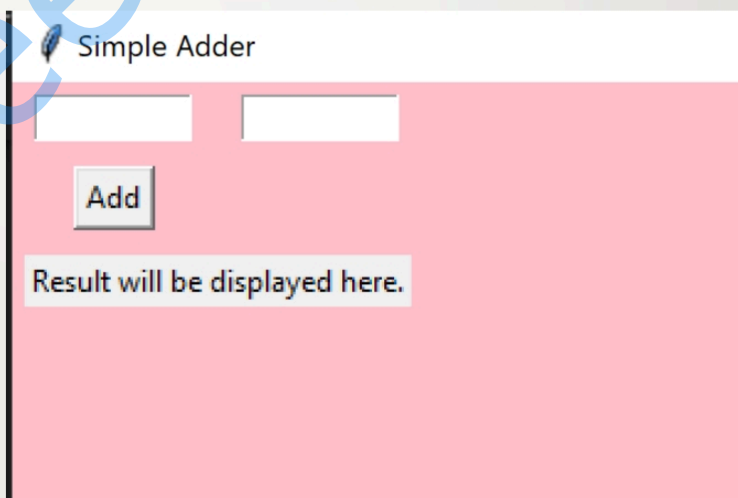
Output:



Label:

```
label1 = Label(root, text="Result will be displayed here.")  
label1.grid(row=2, column=0, columnspan=2, padx=5, pady=5)
```

Output:



A screenshot of a Tkinter window titled "Simple Adder". The window has a pink background. At the top, there is a white title bar with the text "Simple Adder". Below the title bar, there are two white input fields side-by-side. Below the input fields, there is a button labeled "Add". Below the button, there is a label displaying the text "Result will be displayed here.".

FILE MANAGEMENT



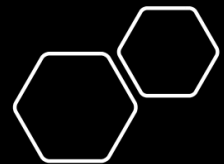
- Repetition of data
- Slow searching
- Difficult to update
- Limited access

DATABASE



RollNo	Name	Class
101	Khan	12

- No duplication
- Faster search
- Easy updates
- Concurrent access



What is a Database?

A **database** is a **systematic collection** of data, usually stored in a computer.

It allows for:

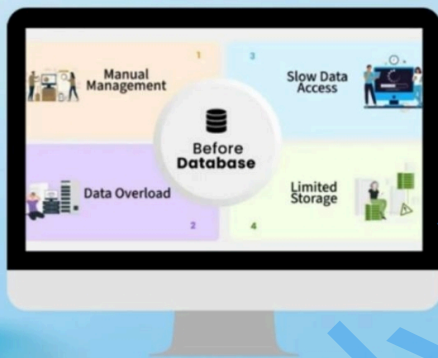
- Storing
- Retrieving
- Managing data for multiple users

Examples of use:

- Logging in
- Online shopping
- Transactions



Importance of Databases



- Removes **redundancies** and **inconsistencies**
- Organizes data for **accuracy and reliability**
- Allows **concurrent access** (multi-user access)
- Includes **security measures** to protect sensitive info.
- Enables **efficient information retrieval & data analysis**.

Types of Databases

Relational Databases:

- Data organized in **tables** (rows & columns)
- Each table represents an **entity** (e.g., Customers, Orders)

NoSQL Databases:

- Flexible structure
- Can manage **unstructured** or **semi-structured** data
- No fixed schema or table format



RELATIONAL DATABASE



University System

Student	Course	Enrollment
RollNo	Course	CAL
Name	Class	12



Banking System

Customer	Account	
CustomerID	Balance	Customer
Name	Address	

NOSQL DATABASE



Social Media Apps

- Text
- Image
- Video



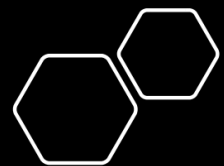
E-Commerce Product Catalogs

- Feature 1
- Feature 2
- Feature 3

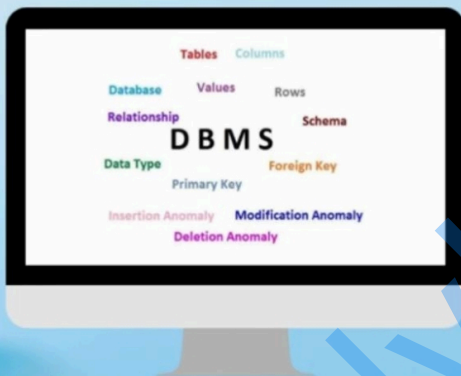


Recommendation Systems

- Netflix
- Spotify
- YouTube

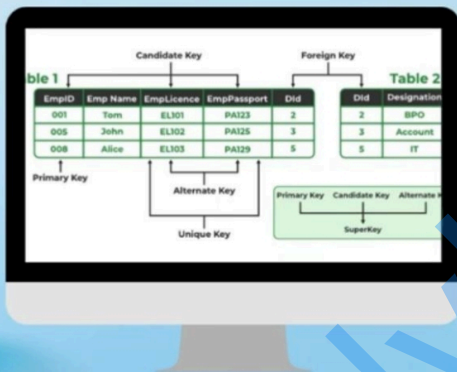


Database Key Concepts/Terms



- **Table:** Core structure made of rows (records) and columns (fields)
- **Record:** A row with complete info about one item (e.g., a customer)
- **Field:** A column that holds specific type of data (e.g., Name, Age)
- **Data Integrity:** Ensures data is accurate, consistent, and reliable
- **Keys:** Unique identifiers used to link and organize records across tables

Database Keys Overview

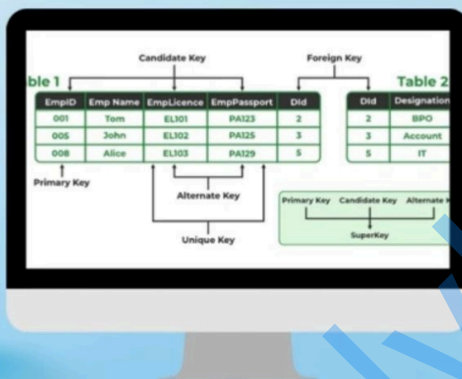


- Help uniquely identify records
- Allow relationships between tables

Types of keys:

- Primary Key
- Secondary Key
- Candidate Key
- Foreign Key
- Composite Key

Primary & Candidate Keys



Primary Key:

- Unique identifier for a record
- Can be single or composite

Candidate Key:

- Any field(s) that can uniquely identify record
- One is chosen as primary, others secondary

Secondary Key:

A field used for data access, but not unique

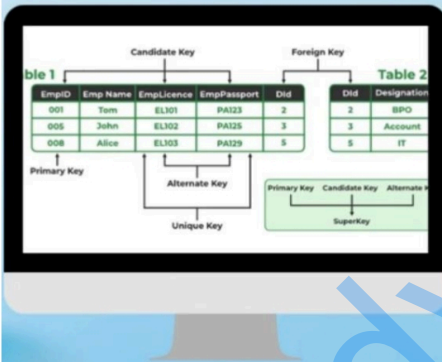
Foreign & Composite Keys

Foreign Key:

- Field in **one table** that refers to **primary key** in another
- Creates **relationships** between tables

Composite Key:

- **Combination of two or more attributes** to uniquely identify a record



Student Management System Example

1. Student Table

RollNo (PK)	Name	Email	CNIC	Phone	DeptID (FK)
101	Ali Khan	ali101@gmail.com ↗	12345-6789	0300123456	CS
102	Sara	sara102@gmail.com ↗	98765-4321	0300654321	IT
103	Ahmed	ahmed103@gmail.com ↗	45678-1234	0300765432	CS

2. Department Table

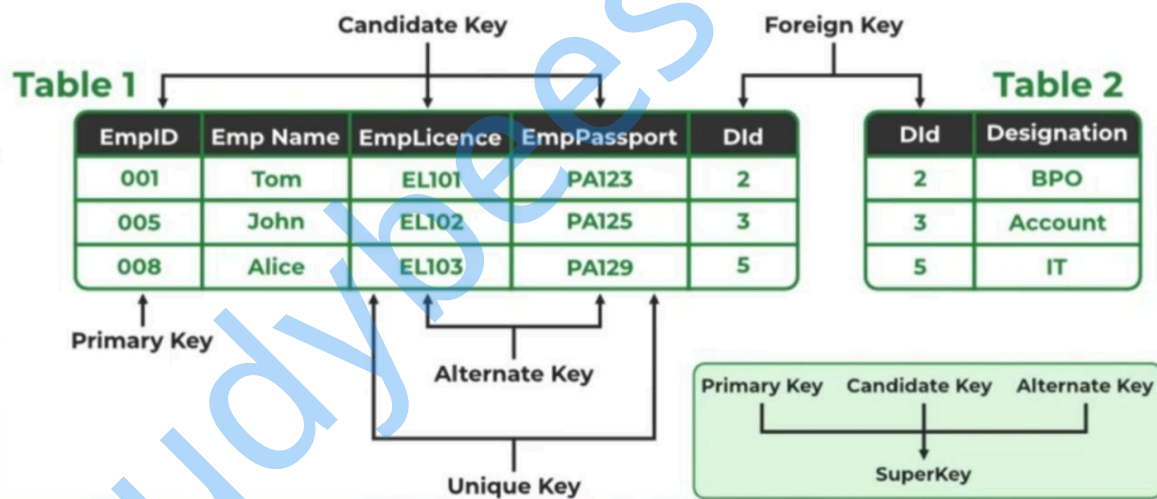
DeptID (PK)	DepartmentName
CS	Computer Science
IT	Information Tech

Enrollment

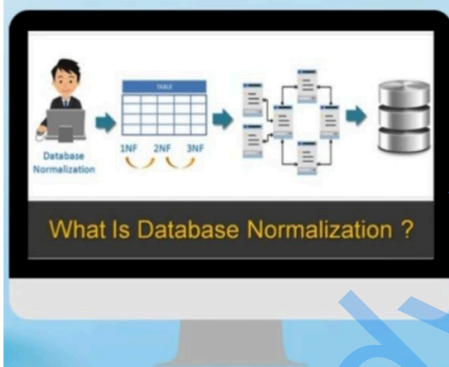
RollNo	CourseID	Grade
101	CS101	A
101	CS102	B
102	CS101	A

(Here RollNo + CourseID is the Composite Key).

KEYS



What is Data Normalization?



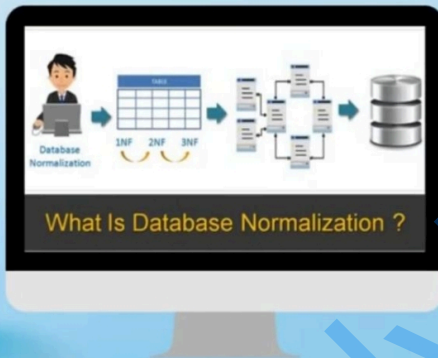
- Process of minimizing **duplicate data**
- Divides contents into **smaller interlinked tables**
- **Ensures:**
 - Efficient storage
 - Fast data access
 - Data consistency

Why Normalize Data?

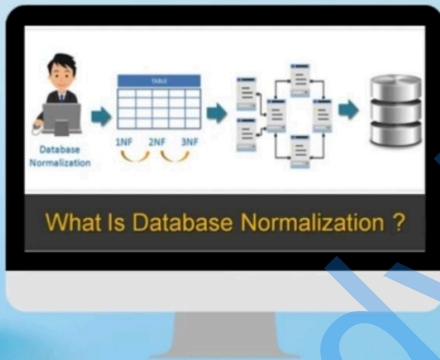
- Removes redundancy and avoids update errors
- Ensures consistency across records

Example:

- A student enrolled in multiple courses
- **Without normalization:** same student info repeated
- **With normalization:** student data stored once, linked by Student ID



What is Data Normalization?



- Process of organizing data in a database to **reduce redundancy**.
- Helps **improve data integrity**.
- Achieved through a series of **steps called Normal Forms (NFs)**.

Steps of Normalization

First Normal Form (1NF):

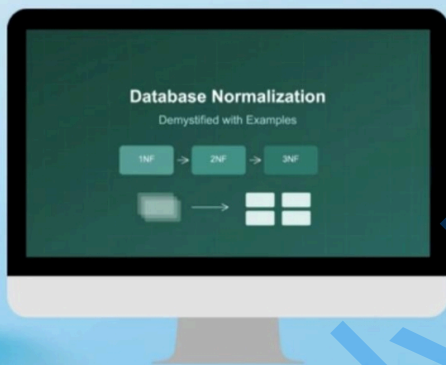
- Eliminate repeating groups/multi-valued attributes.
- Use atomic values only

Second Normal Form (2NF):

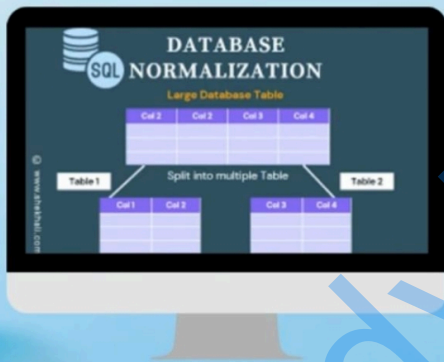
- Remove partial dependencies.

Third Normal Form (3NF):

- Remove transitive dependencies.



Laptop Shop Database Example



Entities and Attributes:

- **Customer:** C_ID, C_Name, C_Address, C_Phone
- **Laptop:** L_ID, L_Name, L_Price, L_Brand
- **Order:** OrderID, OrderDate, C_ID, L_ID, Quantity

Unnormalized Table (Sample)

<u>OrderID</u>	OrderDate	C_ID	C_Name	C_Address	C_Phone	L_ID	L_Name	L_Price	L_Brand	Quantity
1	20 Nov 2024	C1	Qamar	House:123	123-4567	L1	Dell XPS 13	100000	Dell	1
2	21 Nov 2024	C2	Mahtab	House:456	987-6543	L2	MacBook Air	90000	Apple	2
3	22 Nov 2024	C1	Qamar	House:123	123-4567	L3	HP Spectre x360	120000	HP	1

1NF – Remove Repeating Groups

Ensure **atomic values** and **no repeating fields**.

To normalize to **1NF**, we need to find **repeating groups**, which are:

- **Customer information** is repeated for each order.
- **Laptop information** is repeated for each order.

Solution:

- Split the unnormalized data into three separate tables.

2NF – Eliminate Partial Dependencies

A **partial dependency** occurs when a **non-key attribute** is dependent on **part of a composite primary key**, not the **whole key**.

OrderDate depends only on OrderID, not on full key (OrderID, C_ID).
So, create a new table, **OrderDetails Table**.

OrderID	OrderDate
1	20 Nov 2024
2	21 Nov 2024
3	22 Nov 2024

2NF – Eliminate Partial Dependencies

Order Table (After 2NF):

<u>OrderID</u>	<u>C_ID</u>	L_ID	Quantity
1	C1	L1	1
2	C2	L2	2
3	C1	L3	1

3NF – Eliminate Transitive Dependencies

A **transitive dependency** occurs when a **non-key attribute depends on another non-key attribute**, which is also depends on the **primary key**.

Order Table:

<u>OrderID</u>	C_ID	L_ID	L_Name	L_Price	L_Brand	Quantity
1	C1	L1	Dell XPS 13	100000	Dell	1
2	C2	L2	MacBook Air	90000	Apple	2

Here:

OrderID → L_ID

L_ID → L_Name, L_Price, L_Brand

3NF – Eliminate Transitive Dependencies

After 2NF, we have already split into four tables:

1. **Customer**(C_ID, C_Name, C_Address, C_Phone)
2. **Laptop**(L_ID, L_Name, L_Price, L_Brand)
3. **OrderDetails**(OrderID, OrderDate)
4. **Order**(OrderID, C_ID, L_ID, Quantity)

Now:

- All **non-key attributes** in each table depend only on their **own primary key**.
- There are no transitive dependencies.

Normalization

- 1st Normal Form

- 2nd Normal Form

- 3rd Normal Form

1st Normal Form:

A relation is in first normal form (1NF) if it does not contain a **repeating group**.

Accountant Number	Skill Number	Skill Category	Name
21	113	System	Ali
35	113	System	Daud
	179	Tax	
50	204	Audit	Chohan
	179	Tax	

Normalization

- 1st Normal Form

- 2nd Normal Form

- 3rd Normal Form

1st Normal Form:

A relation is in first normal form (1NF) if it does not contain a repeating group.

Accountant Number	Skill Number	Skill Category	Name
21	113	System	Ali
35	113	System	Daud
	179	Tax	
	204	Audit	
50	179	Tax	Chohan

Accountant Number	Skill Number	Skill Category	Name
21	113	System	Ali
35	113	System	Daud
35	179	Tax	Daud
35	204	Audit	Daud
50	179	Tax	Chohan

2nd Normal Form

A relation is in 2nd normal form if it is in 1st normal form and there is no partial dependency.

PK

Accountant Number	Skill Number	Skill Category	Name	Group Number	Proficiency
21	113	System	Ali	52	3
35	113	System	Daud	44	5
35	179	Tax	Daud	44	1
35	204	Audit	Daud	44	6
50	179	Tax	Chohan	44	2

PK

Accountant Number	Name	Group Number
21	Ali	52
35	Daud	44
50	Chohan	44

Partial Dependency:

In a DBMS, partial dependency refers to a situation where a non-key attribute depends on only a portion of the primary key, rather than the entire primary key.

PK

Skill Number	Skill Category
113	System
179	Tax
204	Audit

PK

Accountant Number	Skill Number	Proficiency
21	112	3
35	113	5
35	179	1
35	204	6
50	179	2

3rd Normal Form

A relation is in 3rd normal form if it is in 2nd normal form and there is no transitive dependency.

PK

Accountant Number	Name	Group Number
21	Ali	52
35	Daud	44
50	Chohan	44

PK

Skill Number	Skill Category
113	System
179	Tax
204	Audit

PK

Accountant Number	Skill Number	Proficiency
21	112	3
35	113	5
35	179	1
35	204	6
50	179	2

Transitive Dependency:

A transitive dependency occurs when the value of one non-key attribute is determined by another non-key attribute through a chain of dependencies, rather than directly by the primary key.

3rd Normal Form

A relation is in 3rd normal form if it is in 2nd normal form and there is no transitive dependency.

PK			
BOOKID	BOOK Description	Category ID	Category Description
1001	BOOK A	F1	History
1002	BOOK B	F2	Science
1003	BOOK C	F3	Geography
1004	BOOK D	F4	Math

BookID → Category_ID
Category_ID → Category_Description

Transitive Dependency:

A transitive dependency occurs when the value of one non-key attribute is determined by another non-key attribute through a chain of dependencies, rather than directly by the primary key.

PK		FK
BOOKID	BOOK Description	Category ID
1001	BOOK A	F1
1002	BOOK B	F2
1003	BOOK C	F3
1004	BOOK D	F4

PK	
Category ID	Category Description
F1	History
F2	Science
F3	Geography
F4	Math

Database Tools

MS Access

- A database tool made by Microsoft.
- Very easy to use — no coding needed.
- Can create tables and forms by drag-and-drop.

Best for small projects, like:

- Managing inventory
- Tracking student records
- Planning event

Database Tools

SQLite

- A lightweight database that works without a server.
- Saves everything in just one file.

Commonly used in:

- Mobile apps
- Small desktop programs
- Simple and easy to set up – no installation required.

Advantages of SQLite



- No need to install or manage a server.
- Works on mobile and desktop apps easily.
- Very lightweight, uses very little space.
- All data is stored in one small file.
- Great choice for apps that need fast and local storage.

Creating a TABLE



```
CREATE TABLE "Laptop"(  
    L_id TEXT PRIMARY KEY,  
    L_name TEXT NOT NULL,  
    L_brand TEXT NOT NULL,  
    L_price INTEGER NOT NULL  
);
```

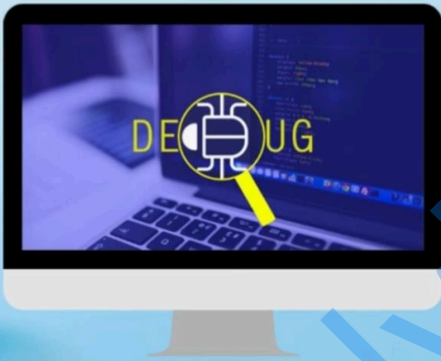
Adding Record

- INSERT INTO Laptop VALUES ("L1","Dell 13 pro","Dell", 100000);
- **Adding multiple record**
INSERT INTO Laptop
VALUES
("L2","HP series i5","HP", 150000),
("L3", "Lenovo series 13", "Lenovo",
120000);

Querying the record

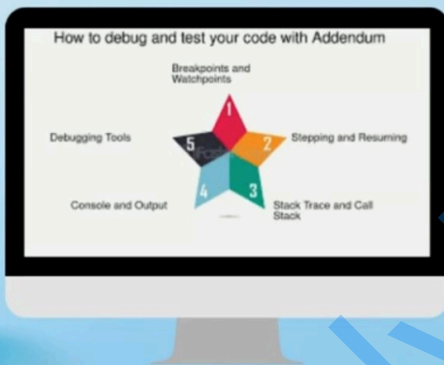
- **SELECT** * FROM Laptop
- WHERE L_id = "L3";
- **Updating Record**
UPDATE Laptop SET L_price = 130000
WHERE L_id = "L1";
- SELECT * FROM Laptop
ORDER by L_price DESC;
- **DELETE** FROM Laptop
WHERE L_id = "L3";

Introduction to Debugging



- Process of **analyzing** and **locating or identifying errors** (bugs) in code
- Helps **ensure** the **program runs smoothly** and gives the correct result
- **Basic technique:** Using print statements to check if the code behaves as expected
- Print critical values, expressions, or formula results
- Not needed for every line, just key checkpoints in the code.
- **Goal** is to track how variables change over time and verify if they match expectations

Advanced Debugging Tools

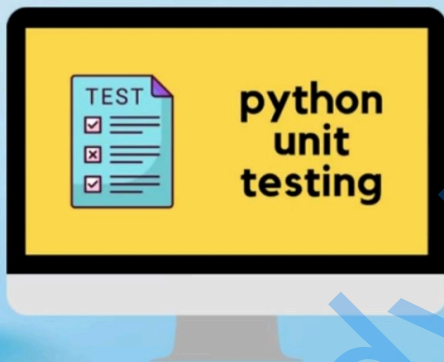


- **Breakpoints:** Pause the program at specific lines
- **Watchpoints:** Monitor variables and expressions

Benefits:

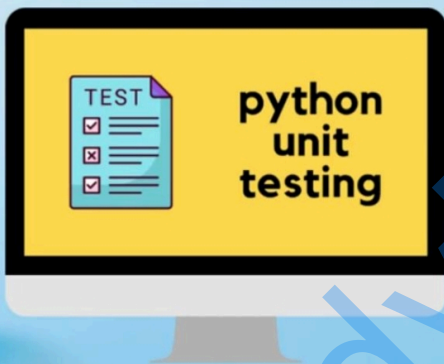
- Visual tracking of code execution
- View the values of variables live
- Easily identify incorrect logic or unexpected values

What is Python unittest?



- **unittest** is a built-in framework in Python.
- It helps **automate** testing of your code.
- You can **create small test functions** to validate your logic.
- Helps **detect bugs**, especially in larger, complex programs.

Why Use unittest?



- Manual testing becomes difficult as code grows.

unittest allows:

- Organized, repeatable tests
- Clear pass/fail feedback
- Debugging support with breakpoints