Python - Day - 17 (Intermediate)

Topic - OOP (Application & Structure)

(*) Why Day 17 Exists (Exact purpose)

Day 17 is where (Angela) locks in OOP as a SKILL not a Concept.

We already know:
- what a class is
- what an object is

Now you learn:
- how to design classes
- how __init__ actually structures objects
- how attributes are created and stored
- how methods modify internal state
- how objects interact cleanly

(*) Class Design Philosophy

A good class should:
(1) Represent one real world entity
(2) Hold state (attributes)
(3) Expose behavior (methods)
(4) Hide implementation details

class ≠ random container
class = well-defined responsibility

(*) Constructor (__init__)

What __init__ Really is

- Special method
- Automatically executed when object is created
- Responsible for setting up object state

```
Class user:
    def __init__(self):
        pass
```

This runs when

```
user = User()
```

Why Constructors matter

Without __init__ :

- Object have no guaranteed attributes
- Code becomes fragile

With __init__ :

- Every Object starts in a valid state

(*) Parameters (vs) Attributes (Critical Distinction)

```
Class User:
    def __init__(self, user_id, username):
        self.id = user_id
        self.username = username
```

What is happening here?

Term	Meaning
User_id	Temporary parameter
Self.id	permanent attribute
Username	Temporary parameter
Self.username	permanent attribute

(*) parameters die after constructors
(*) Attributes live as long as object lives

*) How Python Creates Objects (Internal Flow)

user = User("001", "mohit")

Python does this:
(1) Creates empty object
(2) Calls __init__(user, "001", "mohit")
(3) Assigns attributes to that object
(4) Returns fully initialized object

This is object birth

*) Attribute Creation Rules
Attribute are created only when assigned to self

self.followers = 0 # creates attribute
followers = 0 # local variable discarded

(*) if it's not attached to self it does not belong to the object

(*) Default Attributes (Object State design)

```
Class User:
    def __init__(self, username):
        self.username = username
        self.followers = 0
        self.following = 0
```

why default matter:
- Predictable Object behaviour
- Prevents runtime errors
- Enforces Consistency

(*) Methods Modify Object State

```
class User:
    def follow(self):
        self.followers += 1
```

This method:
- Reads current state
- updates state
- Save state back into object

Objects are Stateful, not Static

(*) Methods with parameters (Real Behavior)

```
def follow(self, user):
    user.followers += 1
    self.following += 1
```

Page No.
Date

Key idea:
- Objects can receive other objects
- Objects can change other objects
- Clean interaction without globals

This is real-world modeling.

(*) Encapsulation in practice (Not Theory)

I am not saying "encapsulation" loudly but uses it.

Encapsulation here means:
- Data is owned by object
- Methods control how data changes

You do:
↳ user.follow(other_user)

you do NOT:
↳ other_user.followers += 1 # bad design

(*) Class vs Instance Attributes (IMPORTANT)

Instance attribute
~ ~ ~ ~

self.followers = 0

Each object has its own copy

Class attribute:

```
class User:
    platform = "Instagram"
```

Shared by all objects.
{ Angela hints this concept }

(*) Adding Methods AFTER Attributes

Order inside class:
(1) __init__
(2) Methods

This is convention, not rule - but professional standard

(*) Using External classes (APPLICATION)
Example:

```
from turtle import Turtle
tim = Turtle()
```

you don't touch __init__
you TRUST it

This reinforces:
- Constructors define object state
- Users interact via methods

Page No. _____
Date

(*) PrettyTable - OOP APPLICATION

```
from prettytable import PrettyTable
table = prettytable()
```

you:
- Created object
- Used methods
- Modified attributes

```
table.align = "1"
```
This is Controlled access

(*) Dot Notation = Object Contract

Dot notation means:
"This class promises this behavior"
you rely on that promise

(*) Common Design Mistakes
(x) Creating attributes otside init
(x) Forgetting self
(x) Treating objects like dictionaries
(x) Overloading one class with many roles
(x) Directly modifying other object data

(*) The REAL learning Outcome of Day 17

By the end of Day 17, you should be able to:

Page No.
Date

- Design clean classes
- Use Constructors properly
- Predict object behavior
- Read & use third-party OOP code
- Build Systems, not Scripts

(*) One-look final Revision (Exam Gold)

- __init__ builds object state
- Attributes belong to self
- Methods change state
- Objects interact via methods
- Defaults prevent bugs
- OOP models real-world logic

Final Note (Important)

Day 17 is where you stop being a Student and start thinking like a Software engineer

if you understand this page you don't know OOP — you Use OOP