Abstract Classes in Python

Introduction

As Python programs become larger, classes often start representing different types of objects that share a common idea but behave differently.

For example, a learning platform may have different types of resources:

- PDFResource
- VideoResource
- QuizResource
- CourseResource

All of them are resources, but each one may need to provide its own way of displaying or opening the resource.

This creates an important design question:

> How can we define what every child class **must provide**, without deciding exactly how it should implement it?

This is where **abstract classes** become useful.

An abstract class defines a common structure or contract for related classes.

It can say:

> “Every class that inherits from me must implement these methods.”

---

1. What Is an Abstract Class?

An **abstract class** is a class designed to be inherited from rather than directly used to create objects.

It can contain:

- normal attributes
- normal methods
- abstract methods
- constructors
- shared logic

An abstract class provides a **blueprint** for child classes.

For example:

```
    Resource
   /      \
 PDF      Video
```

`Resource` can define what every resource should be able to do.

But `PDF` and `Video` decide how that behavior actually works.

---

2. Why Do We Need Abstract Classes?

Suppose we have:

```
class Resource:
```

```
    pass
```

Then we create:

```
class PDF(Resource):
    pass


class Video(Resource):
    pass
```

There is nothing stopping us from creating:

```
resource = Resource()
```

But perhaps a generic `Resource` should never exist by itself.

A resource should always be a specific type such as:

```
PDF
Video
Quiz
Article
```

An abstract class allows us to enforce this design.

---

3. Abstract Class vs Normal Class

A normal class can usually be instantiated directly:

```
class Resource:
```

```
    pass


resource = Resource()
```

An abstract class cannot be instantiated if it contains unimplemented abstract methods.

Conceptually:

```
Normal Class
    
Can create objects directly

Abstract Class
    
Designed to be inherited
    
Child classes implement required behavior
```

---

4. Python's `abc` Module

Python provides the `abc` module for creating abstract classes.

`abc` stands for:

> Abstract Base Classes

Import:

`from abc import ABC, abstractmethod`

There are two important pieces here:

`ABC`

Used as the base class for an abstract class.

`@abstractmethod`

Used to mark a method that child classes must implement.

---

5. Creating an Abstract Class

Example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

Here:

`class Resource(ABC):`

means `Resource` is an Abstract Base Class.

And:

```
@abstractmethod
def open(self):
```

means every concrete child class must provide an implementation of `open()`.

---

6. Abstract Methods

An **abstract method** defines required behavior without providing the complete implementation.

Example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

The abstract class is saying:

> Every Resource must know how to open itself.

But it does not decide exactly how.

A PDF may open differently from a video.

---

7. Implementing an Abstract Method

Child classes implement the required method.

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass


class PDF(Resource):

    def open(self):
        print("Opening PDF")


class Video(Resource):

    def open(self):
        print("Playing video")
```

Now:

```
pdf = PDF()
video = Video()

pdf.open()
video.open()
```

Output:

```
Opening PDF
Playing video
```

The parent defines the requirement.

The child defines the implementation.

---

8. You Cannot Instantiate an Incomplete Abstract Class

Consider:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

Trying:

```
resource = Resource()
```

causes an error.

Python raises:

```
TypeError
```

because `Resource` contains an abstract method that has not been implemented.

This is intentional.

The abstract class is not meant to represent a complete object.

---

9. Why Is This Useful?

Without abstraction, developers might accidentally create incomplete objects.

For example:

```
resource = Resource()
```

What exactly should happen when:

```
resource.open()
```

is called?

There is no specific resource type.

An abstract class prevents this design problem.

Instead:

```
pdf = PDF()
```

or:

`video = Video()`

must be used.

---

10. Abstract Classes Define a Contract

One of the most important ideas is:

> An abstract class defines a contract.

Suppose:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

The contract is:

```
Any concrete Resource
        
must provide open()
```

A child class that does not follow the contract cannot be instantiated.

---

11. A Child Class Must Implement All Abstract Methods

Consider:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass

    @abstractmethod
    def close(self):
        pass
```

Now:

```
class PDF(Resource):

    def open(self):
        print("Opening PDF")
```

`PDF` is still abstract because it has not implemented:

```
close()
```

Therefore:

```
pdf = PDF()
```

will fail.

The child must implement both:

```
class PDF(Resource):

    def open(self):
        print("Opening PDF")

    def close(self):
        print("Closing PDF")
```

Now:

```
pdf = PDF()
```

works.

---

12. Abstract Classes Can Have Normal Methods

An abstract class does not have to contain only abstract methods.

It can contain normal methods too.

Example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

```
    def info(self):
        print("This is a learning resource.")
```

Child class:

```
class PDF(Resource):

    def open(self):
        print("Opening PDF")
```

Now:

```
pdf = PDF()

pdf.open()
pdf.info()
```

Output:

```
Opening PDF
This is a learning resource.
```

The abstract class can therefore provide **shared behavior** while requiring child classes to provide specialized behavior.

---

13. Abstract Class With a Constructor

Abstract classes can have `__init__()` methods.

Example:

```
from abc import ABC, abstractmethod
```

```
class Resource(ABC):

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

    @abstractmethod
    def open(self):
        pass
```

Child:

```
class PDF(Resource):

    def open(self):
        print(f"Opening {self.title}")
```

Usage:

```
pdf = PDF("Python Basics")

pdf.open()
```

Output:

```
Opening Python Basics
```

The constructor belongs to the parent, so the child receives the shared initialization.

---

14. Abstract Class With Shared Logic

Abstract classes are especially useful when child classes share some behavior.

Example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

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

    def describe(self):
        print(f"Resource: {self.title}")

    @abstractmethod
    def open(self):
        pass
```

Child classes:

```
class PDF(Resource):

    def open(self):
        print(f"Opening PDF: {self.title}")


class Video(Resource):

    def open(self):
        print(f"Playing video: {self.title}")
```

The parent handles:

`describe()`

while each child handles:

`open()`

This gives us:

```
Shared behavior
    ↓
Abstract parent
    ↓
Specialized behavior
    ↓
Child classes
```

---

15. Abstract Classes and Polymorphism

Abstract classes work very well with polymorphism.

Example:

```
resources = [
    PDF("Python Basics"),
    Video("Python OOP")
]

for resource in resources:
    resource.open()
```

Each object responds to the same method:

`open()`

but behaves differently.

Output:

```
Opening PDF: Python Basics
Playing video: Python OOP
```

The code does not need to know the exact resource type.

It only relies on the contract:

`Resource must have open()`

---

16. Abstract Class as a Common Interface

Imagine:

```
Resource
  |
  +--- PDF
  |
  +--- Video
  |
  +--- Quiz
  |
  +--- Article
```

The parent can define:

`open()`

Each child implements it differently.

```
class PDF(Resource):

    def open(self):
        print("Opening PDF")


class Video(Resource):

    def open(self):
        print("Playing video")


class Quiz(Resource):

    def open(self):
        print("Starting quiz")


class Article(Resource):

    def open(self):
        print("Opening article")
```

Now other parts of the program can simply call:

```
resource.open()
```

without worrying about the exact type.

---

17. Abstract Properties

Abstract classes can also require properties.

Example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @property
    @abstractmethod
    def resource_type(self):
        pass
```

Child:

```
class PDF(Resource):

    @property
    def resource_type(self):
        return "PDF"
```

Now:

```
pdf = PDF()

print(pdf.resource_type)
```

Output:

```
PDF
```

This is useful when every child must expose a particular piece of information.

---

18. Abstract Class With Properties and Methods

A more realistic example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

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

    @property
    @abstractmethod
    def resource_type(self):
        pass

    @abstractmethod
    def open(self):
        pass

    def describe(self):
        print(f"{self.title} ({self.resource_type})")
```

Child:

```
class PDF(Resource):

    @property
    def resource_type(self):
```

```
        return "PDF"

    def open(self):
        print(f"Opening {self.title}")
```

Usage:

```
pdf = PDF("Python OOP")

pdf.describe()
pdf.open()
```

Output:

```
Python OOP (PDF)
Opening Python OOP
```

---

19. Abstract Class vs Interface

Python does not have a separate `interface` keyword like some languages.

Instead, abstract base classes can provide interface-like behavior.

For example:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

This effectively says:

`Every Resource must provide open()`

But Python's abstract classes can also contain:

- constructors
- attributes
- implemented methods
- properties
- shared logic

So an abstract class can be more than just an interface.

---

20. Abstract Class vs Concrete Class

Abstract Class	**Concrete Class**
Defines a blueprint	Represents a usable implementation
May contain abstract methods	Must implement inherited abstract methods
Usually not instantiated directly	Can be instantiated
Defines required behavior	Provides actual behavior

Often used as a parent	Often used as a child

Example:

`Resource`

`Abstract class`

`PDF`

`Concrete class`

---

21. Abstract Class vs Inheritance

Inheritance answers:

> “Is this class a type of another class?”

For example:

`PDF is a Resource`

Abstract classes add another requirement:

> “What must every Resource provide?”

For example:

`Every Resource must implement open()`

So they are related but not identical concepts.

---

22. Abstract Classes vs Composition

Composition answers:

> “What objects does this object contain or use?”

For example:

`Course HAS-A Resource`

Abstract inheritance answers:

> “What common contract do these related classes follow?”

For example:

`PDF IS-A Resource`
`Video IS-A Resource`

A real application can use both.

```
Course
  |
 +--- contains Resources
            |
            +--- PDF
            +--- Video
            +--- Quiz
```

---

23. A Real haas.dev Example

Imagine haas.dev has different learning resources:

```
Resource
  |
  +--- PDF
  +--- Quiz
  +--- Article
```

Every resource should provide:

```
open()
```

But opening each resource is different.

A PDF might open a PDF file.

A quiz might start quiz questions.

An article might display article content.

We can model this with an abstract class:

```
from abc import ABC, abstractmethod


class Resource(ABC):

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

    @abstractmethod
    def open(self):
        pass

    def describe(self):
        print(f"Resource: {self.title}")
```

PDF:

```
class PDF(Resource):

    def open(self):
        print(f"Opening PDF: {self.title}")
```

Quiz:

```
class Quiz(Resource):

    def open(self):
        print(f"Starting quiz: {self.title}")
```

Article:

```
class Article(Resource):

    def open(self):
```

```
        print(f"Opening article: {self.title}")
```

Now:

```
resources = [
    PDF("Python Basics"),
    Quiz("Python Functions Quiz"),
    Article("Understanding APIs")
]

for resource in resources:
    resource.describe()
    resource.open()
```

Output:

```
Resource: Python Basics
Opening PDF: Python Basics

Resource: Python Functions Quiz
Starting quiz: Python Functions Quiz

Resource: Understanding APIs
Opening article: Understanding APIs
```

The parent class defines the contract.

Each resource provides its own implementation.

---

24. Abstract Classes Improve Design

Abstract classes are useful because they make requirements explicit.

Without abstraction:

```
class PDF:
    pass


class Video:
    pass
```

There is no clear indication of what every resource should support.

With abstraction:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

The requirement becomes part of the program structure.

This helps developers understand:

```
Resource
   
must implement open()
```

---

25. Abstract Classes and Dependency Injection

Abstract classes can also make applications easier to extend and test.

For example:

```
class Storage(ABC):

    @abstractmethod
    def save(self, data):
        pass
```

Different implementations:

```
class FileStorage(Storage):

    def save(self, data):
        print("Saving to file")


class DatabaseStorage(Storage):

    def save(self, data):
        print("Saving to database")
```

Another class can depend on the abstraction:

```
class ResourceManager:

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

    def save_resource(self, resource):
        self.storage.save(resource)
```

Now the manager does not need to know whether data is stored in:

- a file
- a database
- cloud storage
- a test implementation

It only expects the `Storage` contract.

---

26. Abstract Classes and Testing

Suppose:

```
class Storage(ABC):

    @abstractmethod
    def save(self, data):
        pass
```

A real implementation may use a database.

For testing, we can create another implementation:

```
class TestStorage(Storage):

    def save(self, data):
        print("Saving test data")
```

Now application code can use `TestStorage` during tests.

This reduces dependency on external systems.

---

27. Multiple Abstract Methods

An abstract class can define multiple requirements.

```
from abc import ABC, abstractmethod


class PaymentProcessor(ABC):

    @abstractmethod
    def pay(self, amount):
        pass

    @abstractmethod
    def refund(self, amount):
        pass
```

A concrete class must implement both:

```
class CardPayment(PaymentProcessor):

    def pay(self, amount):
        print(f"Paid {amount}")

    def refund(self, amount):
        print(f"Refunded {amount}")
```

This gives the child class a clear contract.

---

28. Abstract Methods Can Contain Code

An abstract method can technically contain an implementation.

Example:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        print("Opening resource")
```

A child can call the parent's implementation:

```
class PDF(Resource):

    def open(self):
        super().open()
        print("Opening PDF file")
```

Output:

```
Opening resource
Opening PDF file
```

However, use this intentionally.

An abstract method is primarily communicating:

> “Concrete classes must provide this behavior.”

---

29. Partial Implementations

An abstract class can be inherited by another abstract class.

Example:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass

    @abstractmethod
    def close(self):
        pass
```

Intermediate class:

```
class DigitalResource(Resource):

    def close(self):
        print("Closing digital resource")
```

`DigitalResource` is still abstract because it has not implemented:

```
open()
```

A final child can complete it:

```
class PDF(DigitalResource):

    def open(self):
```

```
        print("Opening PDF")
```

Now `PDF` can be instantiated.

---

30. Common Mistake: Forgetting `ABC`

Incorrect:

```
from abc import abstractmethod


class Resource:

    @abstractmethod
    def open(self):
        pass
```

The class does not inherit from `ABC`.

Correct:

```
from abc import ABC, abstractmethod


class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

Use:

`class Resource(ABC):`

when creating an abstract base class.

---

31. Common Mistake: Forgetting the Decorator

Incorrect:

```
class Resource(ABC):

    def open(self):
        pass
```

This is simply a normal method.

Correct:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

The decorator tells Python that the method is required.

---

32. Common Mistake: Incomplete Child Class

Suppose:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass

    @abstractmethod
    def close(self):
        pass
```

And:

```
class PDF(Resource):

    def open(self):
        print("Opening PDF")
```

`PDF` is still abstract.

You must implement:

```
def close(self):
```

too.

---

33. Common Mistake: Making Everything Abstract

Not every method needs to be abstract.

Bad design:

```
class Resource(ABC):

    @abstractmethod
    def describe(self):
        pass

    @abstractmethod
    def validate_title(self):
        pass

    @abstractmethod
    def format_title(self):
        pass
```

If all child classes would use the same logic, the parent can implement it once.

Better:

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass

    def describe(self):
        print("This is a resource.")
```

Use abstraction where behavior genuinely needs to vary.

---

34. Common Mistake: Using Abstract Classes Just Because They Exist

Do not automatically create an abstract class for every inheritance hierarchy.

Ask:

```
Do these classes share a meaningful concept?
        ↓
Do they need a common contract?
        ↓
Do different implementations need to follow that contract?
```

If yes, an abstract class may be useful.

Otherwise, a normal class, composition, or another design may be simpler.

---

35. A Simple Mental Model

Think of an abstract class as a **rulebook**.

```
Abstract Class
      ↓
Defines the rules
      ↓
Child Classes
      ↓
Follow the rules
      ↓
Provide actual implementation
```

Example:

```
Resource
  Rule:
  "You must implement open()"
```

```
    □
```

```
PDF
 open() □ Open PDF

Video
 open() □ Play Video

Quiz
 open() □ Start Quiz
```

---

36. When Should You Use Abstract Classes?

Consider an abstract class when:

- multiple classes share a common concept
- they should follow a common contract
- some behavior must be implemented differently
- you want to prevent incomplete objects
- you want polymorphic code
- you want a clear architecture
- different implementations may be swapped
- you want to make dependencies easier to test

---

37. When Should You Avoid Them?

Avoid unnecessary abstraction when:

- there is only one implementation
- there is no meaningful shared contract
- inheritance is being used only for code reuse
- a simple function would solve the problem
- composition would represent the relationship better
- the abstraction adds more complexity than value

Good design is not about using more OOP features.

It is about choosing the right abstraction for the problem.

---

38. Abstract Classes and Real Software Architecture

Abstract classes become particularly useful in larger applications.

For example:

```
Application
    
Service
    
Repository
    
Storage
```

A storage abstraction could define:

```
class Storage(ABC):

    @abstractmethod
```

```
    def save(self, data):
        pass

    @abstractmethod
    def load(self):
        pass
```

Implementations could include:

```
FileStorage
DatabaseStorage
CloudStorage
TestStorage
```

The rest of the application can work with `Storage` rather than depending directly on one implementation.

This is one reason abstraction appears frequently in professional software architecture.

---

39. Problem Solving Framework

Before creating an abstract class, ask these questions:

Step 1: Identify the common concept

What do these classes represent?

```
PDF
Video
Quiz
```

Common concept:

Resource

Step 2: Identify common requirements

What must every resource provide?

open()

Step 3: Identify variable behavior

How does each class perform that action?

PDF open PDF
Video play video
Quiz start quiz

Step 4: Create the contract

```
class Resource(ABC):

    @abstractmethod
    def open(self):
        pass
```

Step 5: Implement each variation

```
class PDF(Resource):
    ...

class Video(Resource):
    ...
```

Step 6: Use polymorphism

```
for resource in resources:
    resource.open()
```

---

40. Mini Project: Resource System

Build a small resource system for haas.dev.

Requirements:

Create an abstract class:

`Resource`

It should have:

- title
- abstract `open()`
- `describe()`

Create three concrete classes:

`PDF`
`Video`
`Quiz`

Each must implement:

`open()`

Example output:

```
Python Functions
Type: PDF
Opening PDF
```

```
Python OOP
Type: Video
Playing video

Python Quiz
Type: Quiz
Starting quiz
```

Possible Solution

```
from abc import ABC, abstractmethod


class Resource(ABC):

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

    @property
    @abstractmethod
    def resource_type(self):
        pass

    @abstractmethod
    def open(self):
        pass

    def describe(self):
        print(f"{self.title}")
        print(f"Type: {self.resource_type}")


class PDF(Resource):
```

```
    @property
    def resource_type(self):
        return "PDF"

    def open(self):
        print("Opening PDF")


class Video(Resource):

    @property
    def resource_type(self):
        return "Video"

    def open(self):
        print("Playing video")


class Quiz(Resource):

    @property
    def resource_type(self):
        return "Quiz"

    def open(self):
        print("Starting quiz")


resources = [
    PDF("Python Functions"),
    Video("Python OOP"),
    Quiz("Python Quiz")
]
```

```
for resource in resources:
    resource.describe()
    resource.open()
    print()
```

---

41. Five Minute Challenge

Create:

`class Notification(ABC):`

It should require:

`send(message)`

Create:

`EmailNotification`
`SMSNotification`
`PushNotification`

Each class should implement `send()` differently.

Expected idea:

```
Email  Sending email
SMS  Sending SMS
Push  Sending push notification
```

Do not create an object directly from `Notification`.

---

42. Exercises

Exercise 1

Create an abstract class:

`Shape`

with:

`area()`

Create:

`Circle`
`Rectangle`
`Triangle`

Each should calculate its own area.

Exercise 2

Create:

`class Employee(ABC)`

Require:

`calculate_salary()`

Create:

`FullTimeEmployee`

PartTimeEmployee
Freelancer

Exercise 3

Create:

class PaymentMethod(ABC)

Require:

pay(amount)

Implement:

CardPayment
CashPayment
OnlinePayment

Exercise 4

Create an abstract:

Storage

with:

save(data)
load()

Implement:

FileStorage
MemoryStorage

---

43. Mini Quiz

Question 1

What is the main purpose of an abstract class?

A. To create database tables
B. To define a common contract for child classes
C. To make every method private
D. To avoid using objects

Answer: B

Question 2

Which module provides Abstract Base Classes?

A. os
B. sys
C. abc
D. abstract

Answer: C

Question 3

Which decorator marks a method as abstract?

A. `@abstract`
B. `@abstractmethod`
C. `@method`
D. `@required`

Answer: B

Question 4

Can an abstract class contain normal methods?

A. Yes
B. No

Answer: A

Question 5

Can an incomplete concrete child class be instantiated?

A. Yes
B. No

Answer: B

---

44. Interview Questions

Beginner

1. What is an abstract class in Python?

A class designed to provide a common blueprint or contract for subclasses.

2. What is the `abc` module?

Python's module for creating and working with Abstract Base Classes.

3. What is `ABC`?

A base class provided by the `abc` module that can be inherited to create an Abstract Base Class.

4. What is `@abstractmethod`?

A decorator that marks a method as required for concrete subclasses.

Intermediate

5. Can abstract classes contain implemented methods?

Yes. They can contain both abstract methods and normal methods.

6. Can an abstract class have `__init__()`?

Yes.

7. Can an abstract class have properties?

Yes. Properties can also be marked as abstract.

8. What happens if a child class does not implement every abstract method?

The child class remains abstract and cannot be instantiated.

Advanced

9. How do abstract classes support polymorphism?

They define a common interface that different concrete classes implement differently, allowing code to operate on objects through the shared abstraction.

10. What is the difference between an abstract class and a concrete class?

An abstract class defines incomplete or required behavior and cannot be instantiated while abstract requirements remain. A concrete class provides complete implementations and can be instantiated.

11. Why are abstract classes useful in large applications?

They establish clear contracts between components and allow different implementations to be substituted while keeping dependent code focused on the abstraction.

12. Are abstract classes always necessary for polymorphism?

No. Python's duck typing allows polymorphism without inheritance or abstract classes. Abstract classes are useful when an explicit contract and structure are valuable.

---

45. Abstract Classes and Duck Typing

Python is dynamically typed and often uses **duck typing**.

The idea is:

> If an object provides the required behavior, it can be used.

For example:

```
def open_resource(resource):
    resource.open()
```

This function does not necessarily require:

`Resource`

as a parent class.

It only needs an object that provides:

`open()`

This means Python programs can use polymorphism without abstract classes.

Abstract classes become useful when you want to make the contract explicit and enforce implementation requirements.

---

46. Abstract Classes vs Duck Typing

Abstract Classes	Duck Typing
Explicit contract	Implicit contract
Uses inheritance	Does not require inheritance
Can prevent incomplete implementations	Relies on actual behavior
Useful for structured architectures	Very natural in Python
Can improve discoverability	Often simpler

Neither approach should automatically be considered better.

Choose based on the problem.

---

47. Key Takeaways

- An abstract class provides a blueprint for related classes.
- Python provides abstract classes through the `abc` module.
- `ABC` is commonly used as the base class.
- `@abstractmethod` marks required methods.
- An abstract class cannot be instantiated while abstract requirements remain.
- Child classes must implement inherited abstract methods before they can be instantiated.
- Abstract classes can contain normal methods.

- Abstract classes can have constructors and properties.
- They are useful for defining explicit contracts.
- They work naturally with polymorphism.
- They can improve architecture and testability.
- They should not be used when a simpler design is enough.
- Python also supports polymorphism through duck typing.

---

48. Cheat Sheet

Import

```
from abc import ABC, abstractmethod
```

Abstract class

```
class Resource(ABC):
    pass
```

Abstract method

```
@abstractmethod
def open(self):
    pass
```

Complete example

```
from abc import ABC, abstractmethod


class Resource(ABC):

    def __init__(self, title):
```

```
        self.title = title

    @abstractmethod
    def open(self):
        pass

    def describe(self):
        print(self.title)


class PDF(Resource):

    def open(self):
        print("Opening PDF")
```

Cannot instantiate incomplete class

```
resource = Resource()
```

Instantiate completed child

```
pdf = PDF("Python Basics")
pdf.open()
```

Multiple implementations

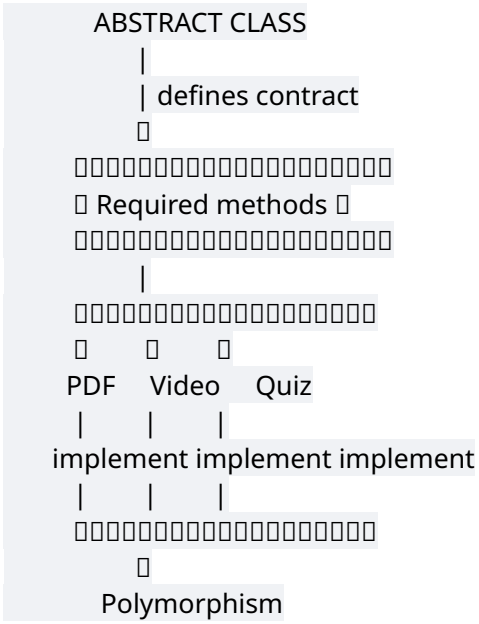
```
resources = [
    PDF("Python"),
    Video("OOP")
]

for resource in resources:
    resource.open()
```

---

49. Final Mental Model

Remember abstract classes using this simple structure:



The key idea is:

> **An abstract class defines what a family of classes must be able to do, while concrete classes decide how they do it.**

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

Key Takeaways

Abstract classes are not primarily about preventing object creation.

Their real purpose is **designing a clear contract between related classes**.

When several implementations must follow the same structure, an abstract class can make that requirement explicit, enforce it, and allow the rest of the application to work with the common abstraction rather than individual implementations.