Encapsulation in Python

Introduction

In Object Oriented Programming, a class combines **data** and **behavior**.

But simply putting data inside a class is not enough.

A well designed class should also control:

- which data can be accessed directly
- which data should be changed through methods
- what values are considered valid
- how the object's internal state can be modified

This idea is called **encapsulation**.

A simple definition is:

> **Encapsulation means keeping an object's data and the logic that controls that data together, while controlling how that internal state is accessed or changed.**

For example, a bank account should not allow every part of the program to freely change its balance without rules.

Instead of:

```
account.balance = -50000
```

we can provide controlled behavior:

```
account.withdraw(500)
```

The method can check whether the withdrawal is valid before changing the balance.

---

1. Why Encapsulation Matters

Imagine a bank account.

The account has:

```
balance
account_number
owner
```

Some operations are allowed:

```
deposit money
withdraw money
check balance
```

But we do not want arbitrary code changing the balance without validation.

Without encapsulation:

```
account.balance = -100000
```

The object could enter an invalid state.

With encapsulation:

```
account.withdraw(500)
```

The class controls how the balance changes.

This makes the object safer and easier to maintain.

---

2. Encapsulation Has Two Main Ideas

When learning encapsulation, think about two related ideas:

1. Bundling

Keep related data and behavior together.

```
BankAccount
□
□□□ balance
□□□ deposit()
□□□ withdraw()
□□□ get_balance()
```

2. Controlled access

Do not allow every piece of code to manipulate internal state without rules.

```
Outside code
    □
Public interface
    □
Internal state
```

The class decides how its state should be used.

---

3. A Simple Example Without Encapsulation

Consider:

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

Now:

```
account = BankAccount(1000)

account.balance = -500
```

Nothing prevents this.

The object now contains:

```
-500
```

That may be an invalid state for a bank account.

---

4. Adding Controlled Behavior

We can move the rules into methods:

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

```
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
```

Now the class defines how money should be added and removed.

Example:

```
account = BankAccount(1000)

account.deposit(500)
account.withdraw(200)

print(account.balance)
```

Output:

```
1300
```

The methods contain the rules for modifying the state.

---

5. Encapsulation Does Not Mean "Nothing Can Be Accessed"

A common misunderstanding is:

> Encapsulation means making everything completely private.

That is not quite correct.

Encapsulation is about **controlling and organizing access to internal state**.

A class can intentionally expose some operations while keeping implementation details internal.

For example:

```
account.deposit(500)
account.withdraw(200)
account.get_balance()
```

These methods form part of the object's **public interface**.

The caller does not need to know exactly how the balance is stored or updated internally.

---

6. Public Attributes

Python attributes are generally public by default.

Example:

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

The attribute can be accessed directly:

```
user = User("Hafsa")
```

```
print(user.name)
```

Output:

```
Hafsa
```

This is perfectly normal Python.

Not every attribute needs to be hidden.

---

7. Naming Conventions for Internal Data

Python provides naming conventions to communicate how an attribute should be used.

The first level is a single underscore:

```
self._balance
```

A leading underscore generally means:

> This is intended for internal use.

Example:

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

The underscore is a **convention**, not a strict access restriction.

Python does not prevent this:

```
account._balance = 500
```

It simply communicates that code outside the class should generally avoid manipulating it directly unless there is a good reason.

---

8. Public vs Internal Attributes

Compare:

```
self.name
```

and:

```
self._balance
```

The convention suggests:

```
name
 
Public interface

_balance
 
Internal implementation detail
```

This distinction helps developers understand how a class is intended to be used.

---

9. Double Underscore and Name Mangling

Python also supports a double leading underscore:

```
self.__balance
```

This triggers **name mangling**.

Example:

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

Now:

```
account = BankAccount(1000)

print(account.__balance)
```

This raises an `AttributeError`.

Python internally transforms the name approximately into:

```
_BankAccount__balance
```

So:

```
print(account._BankAccount__balance)
```

can access it.

This shows an important point:

> Python's double underscore is not absolute security or true private memory.

It is mainly a mechanism for **name mangling** and avoiding accidental name conflicts, especially with inheritance.

---

10. Single Underscore vs Double Underscore

Syntax	Meaning
`name`	Public
`_name`	Internal-use convention
`__name`	Name mangling
`__name__`	Special Python/dunder name

For example:

```
self.name
self._balance
self.__secret
```

mean different things.

---

11. Why Use __?

Double underscore can help prevent accidental collisions between a parent class and subclass.

Example:

```
class Parent:
    def __init__(self):
        self.__value = 10
```

Python mangles this approximately as:

```
_Parent__value
```

A subclass can have:

```
self.__value = 20
```

which becomes approximately:

```
_Subclass__value
```

These are different names.

This is one reason Python provides name mangling.

---

12. Encapsulation Through Methods

A common design is:

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

    def deposit(self, amount):
        if amount <= 0:
            return False

        self._balance += amount
        return True

    def withdraw(self, amount):
        if amount <= 0:
            return False

        if amount > self._balance:
            return False

        self._balance -= amount
        return True

    def get_balance(self):
        return self._balance
```

Now the class controls the important operations.

Usage:

```
account = BankAccount(1000)

account.deposit(500)
account.withdraw(200)

print(account.get_balance())
```

Output:

```
1300
```

---

13. Getters

A method that retrieves internal data is traditionally called a **getter**.

Example:

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

    def get_balance(self):
        return self._balance
```

Usage:

```
print(account.get_balance())
```

The caller doesn't need to know how the balance is stored.

---

14. Setters

A method that changes internal data is traditionally called a **setter**.

Example:

```
class User:
    def __init__(self, age):
        self._age = age

    def set_age(self, age):
        if age >= 0:
            self._age = age
```

Usage:

```
user.set_age(25)
```

The setter can validate the value before changing the state.

---

15. Why Validation Belongs Inside the Class

Suppose we have:

```
class Product:
    def __init__(self, price):
        self._price = price

    def set_price(self, price):
        if price >= 0:
            self._price = price
```

The class owns the rule:

```
price cannot be negative
```

Instead of repeating this validation everywhere:

```
if price >= 0:
    ...
```

the rule lives inside the class.

This reduces duplication and keeps the object's state valid.

---

16. Properties: Python's Cleaner Approach

Python provides `@property` for controlled attribute access.

Instead of:

```
account.get_balance()
```

you can design:

```
account.balance
```

while still controlling how the value is retrieved.

Example:

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

    @property
    def balance(self):
        return self._balance
```

Now:

```
account = BankAccount(1000)

print(account.balance)
```

Output:

```
1000
```

It looks like normal attribute access, but Python is actually calling the property method.

Properties will be covered in greater depth separately.

---

17. Property Setter

A property can also control assignment.

```
class Product:
    def __init__(self, price):
        self.price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")
```

```
        self._price = value
```

Now:

```
product = Product(100)

product.price = 150
```

works.

But:

```
product.price = -50
```

raises an error.

The class controls the state while providing a natural interface.

---

18. Encapsulation Through a Public Interface

Think about a class as a machine.

The user interacts with:

```
Public Interface
    
deposit()
withdraw()
balance
    
Internal Implementation
```

□
_balance
validation
calculations
rules

The caller only needs to understand the public interface.

The internal implementation can change without necessarily changing how the caller uses the class.

---

19. Why This Is Useful

Suppose your class initially stores balance as:

self._balance

Later, you might change the implementation to:

database value
cached value
calculated value
external service

If callers interact through a stable interface such as:

account.withdraw(500)
account.balance

the internal implementation can change without requiring every caller to understand the change.

This is one of the major benefits of encapsulation.

---

20. Encapsulation Reduces Coupling

Coupling describes how strongly different parts of a program depend on each other's internal details.

Bad design:

```
account._balance = account._balance - 500
```

Other code now depends directly on the internal representation.

Better:

```
account.withdraw(500)
```

The caller depends on the behavior rather than the implementation.

This creates a cleaner boundary between parts of the program.

---

21. Encapsulation and Abstraction

Encapsulation and abstraction are related, but they are not the same.

Encapsulation

Focuses on:

> **How data and behavior are bundled and controlled.**

Abstraction

Focuses on:

> **What details should be exposed and what implementation details can be hidden.**

Example:

```
account.withdraw(500)
```

The method provides a simple operation.

The internal details:

```
checking balance
validation
updating state
recording transaction
```

do not need to be handled by the caller.

---

22. Encapsulation and Data Validation

Consider a `Student`:

```
class Student:
    def __init__(self, marks):
        self._marks = marks
```

```
    def set_marks(self, marks):
        if 0 <= marks <= 100:
            self._marks = marks
```

Now:

```
student = Student(80)

student.set_marks(95)
```

is valid.

But:

```
student.set_marks(150)
```

is rejected.

The class protects its own rules.

---

23. Encapsulation Example: User Account

```
class User:
    def __init__(self, username, password):
        self.username = username
        self.__password = password

    def check_password(self, password):
        return self.__password == password

    def change_password(self, old_password, new_password):
```

```
        if self.__password != old_password:
            return False

        if len(new_password) < 8:
            return False

        self.__password = new_password
        return True
```

Usage:

```
user = User("hafsa", "oldpassword")

print(user.check_password("oldpassword"))
```

Output:

```
True
```

Change the password:

```
user.change_password(
    "oldpassword",
    "newpassword123"
)
```

The caller does not directly manipulate:

```
__password
```

Instead, the class controls password-related behavior.

Important security note

This example demonstrates encapsulation only. Storing real passwords as plain strings is not secure. Real applications should use proper password hashing and authentication systems.

---

24. Encapsulation Example: Shopping Cart

```
class ShoppingCart:
    def __init__(self):
        self._items = []

    def add_item(self, item):
        self._items.append(item)

    def remove_item(self, item):
        if item in self._items:
            self._items.remove(item)

    def total_items(self):
        return len(self._items)
```

Usage:

```
cart = ShoppingCart()

cart.add_item("Keyboard")
cart.add_item("Mouse")

print(cart.total_items())
```

Output:

```
2
```

The cart manages its own internal collection.

---

25. Returning Mutable Internal Data Can Break Encapsulation

Consider:

```
class ShoppingCart:
    def __init__(self):
        self._items = []

    def get_items(self):
        return self._items
```

Now:

```
items = cart.get_items()

items.clear()
```

The caller has modified the cart's internal list.

This can weaken encapsulation.

A safer option may be:

```
def get_items(self):
    return self._items.copy()
```

Now the caller receives a copy.

Another option is:

`return tuple(self._items)`

if the caller only needs read-only-style access.

The correct approach depends on the class's API requirements.

---

26. Encapsulation Is About Invariants

An **invariant** is a condition that should remain true for an object.

For a bank account:

`balance >= 0`

For a product:

`price >= 0`

For a student:

`0 <= marks <= 100`

Encapsulation helps the class maintain these rules.

Example:

```
class Product:
    def __init__(self, price):
        if price < 0:
            raise ValueError("Price cannot be negative")
```

```
        self._price = price

    def update_price(self, price):
        if price < 0:
            raise ValueError("Price cannot be negative")

        self._price = price
```

The object tries to remain valid throughout its lifetime.

---

27. Encapsulation and Object Validity

A useful design principle is:

> Try to prevent invalid objects from existing.

Instead of:

```
product = Product(-100)
```

and hoping the rest of the program handles it correctly, validate when the state enters the object.

This makes the rest of the application easier to reason about.

---

28. Encapsulation Does Not Require Getters and Setters Everywhere

A common beginner mistake is creating:

`get_name()`
`set_name()`
`get_email()`
`set_email()`
`get_age()`
`set_age()`

for every attribute automatically.

Python does not require this style.

If an attribute is simple and safe to expose:

`user.name`

may be perfectly appropriate.

Use methods or properties when you actually need:

- validation
- transformation
- controlled modification
- computed values
- compatibility with changing implementation
- meaningful domain behavior

Good encapsulation is about **useful boundaries**, not hiding everything.

---

29. Prefer Behavior Over Direct State Manipulation

Suppose you have:

```
account.balance -= 500
```

This exposes the implementation.

A behavior-focused API is:

```
account.withdraw(500)
```

Why is the second approach often better?

Because `withdraw()` can enforce rules such as:

```
Is amount positive?
Is there enough balance?
Should a transaction be recorded?
Should a notification be sent?
Should a withdrawal limit be checked?
```

The object owns the business logic.

---

30. Encapsulation in a haas.dev Resource

Imagine a resource object:

```
class Resource:
    def __init__(self, title):
        self.title = title
        self._views = 0
```

```
    def open(self):
        self._views += 1

    @property
    def views(self):
        return self._views
```

Usage:

```
resource = Resource("Python OOP")

resource.open()
resource.open()

print(resource.views)
```

Output:

```
2
```

The application can ask:

```
resource.views
```

but the class controls how views are updated.

Instead of allowing arbitrary code to do:

```
resource._views = -500
```

the intended behavior is:

```
resource.open()
```

This creates a clear boundary around the resource's state.

---

31. A Stronger Resource Example

```
class Resource:
    def __init__(self, title, category):
        self.title = title
        self.category = category
        self._views = 0

    def open(self):
        self._views += 1

    @property
    def views(self):
        return self._views

    def is_popular(self):
        return self._views >= 100
```

Now the object owns its behavior:

```
resource.open()

if resource.is_popular():
    print("Popular resource")
```

The caller does not need to know how popularity is calculated.

---

32. Encapsulation Design Pattern

A useful pattern is:

```
Internal State
    ↑
Validation / Rules
    ↑
Public Methods or Properties
    ↑
Outside Code
```

For example:

```
_balance
  ↑
deposit()
withdraw()
  ↑
application code
```

The internal state stays behind a meaningful interface.

---

33. Common Mistake: Thinking `_name` Is Truly Private

This:

```
self._balance
```

does not make the attribute inaccessible.

Python allows:

`account._balance`

The underscore is primarily a convention.

It tells other developers:

> This is an implementation detail; use the public interface instead.

---

34. Common Mistake: Thinking `__name` Is Secure

This:

`self.__password`

uses name mangling.

It does not provide cryptographic security or absolute privacy.

The value can still be accessed through its mangled name.

Therefore:

> Python's encapsulation mechanisms are primarily about software design and preventing accidental access, not security boundaries.

---

35. Common Mistake: Exposing Internal Mutable Data

Avoid returning an internal list directly when callers should not modify it.

Instead of:

```
return self._items
```

consider:

```
return self._items.copy()
```

or:

```
return tuple(self._items)
```

depending on the desired interface.

---

36. Common Mistake: Putting All Logic Outside the Class

This:

```
if amount > 0 and amount <= account.balance:
    account.balance -= amount
```

puts bank-account rules into external code.

A more object-oriented design is:

```
account.withdraw(amount)
```

and let the class decide whether the operation is valid.

This keeps related behavior close to the state it controls.

---

37. Common Mistake: Overusing Private Attributes

You do not need:

```
self.__name
self.__email
self.__age
self.__country
self.__city
```

for every attribute.

If there is no meaningful reason to restrict or control access, a normal public attribute may be simpler.

Use encapsulation where it solves a real design problem.

---

38. Encapsulation vs Data Hiding

These terms are related but not identical.

Encapsulation

Bundling:

```
data + behavior
```

and controlling their interaction.

Data hiding

Reducing direct access to implementation details.

Encapsulation can involve data hiding, but encapsulation is the broader design concept.

---

39. Real World Analogy

Think about an ATM.

You can:

```
insert card
enter PIN
withdraw money
check balance
```

You cannot directly manipulate:

```
bank database
transaction records
account balance storage
security system
```

You interact through a controlled interface.

A class can work similarly:

```
Public Interface
    
```

withdraw()
deposit()
balance
Internal State

You use the object's interface rather than manually changing its internal implementation.

---

40. Encapsulation Checklist

When designing a class, ask:

Data

- What state does this object own?
- Which state should be public?
- Which state is implementation detail?

Behavior

- Which operations should modify the state?
- Where should validation happen?
- Which rules must always remain true?

Interface

- What should outside code be allowed to do?
- Can callers use meaningful methods instead of changing state directly?
- Should a property be used?

Safety

- Could a mutable internal object be exposed?

- Could invalid state be created?
- Are internal implementation details unnecessarily exposed?

---

41. Mini Project: Bank Account

Build a `BankAccount` class with:

Internal state

`_balance`

Public behavior

`deposit()`
`withdraw()`
`get_balance()`

Rules:

- Deposit must be greater than `0`.
- Withdrawal must be greater than `0`.
- Withdrawal cannot exceed the balance.
- Invalid operations should not change the balance.

Starter:

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

    def deposit(self, amount):
        pass
```

```
    def withdraw(self, amount):
        pass

    def get_balance(self):
        pass
```

Test:

```
account = BankAccount(1000)

account.deposit(500)
account.withdraw(200)

print(account.get_balance())
```

Expected:

```
1300
```

---

42. 5 Minute Challenge

Create a `Temperature` class.

It should store temperature internally:

`self._celsius`

Create:

`get_celsius()`

set_celsius()
to_fahrenheit()

Rules:

- Temperature cannot be below absolute zero.
- to_fahrenheit() should return the converted value.
- Invalid temperatures should be rejected.

Formula:

F = (C × 9/5) + 32

Example:

```
temperature = Temperature(25)

print(temperature.get_celsius())
print(temperature.to_fahrenheit())
```

Expected:

```
25
77.0
```

---

43. Exercises

Exercise 1: Student

Create:

```
class Student:
```

Store marks internally.

Rules:

`0 <= marks <= 100`

Add:

`get_marks()`
`set_marks()`
`has_passed()`

Exercise 2: Product

Create:

`class Product:`

Store price internally.

Rules:

`price >= 0`

Add:

`update_price()`
`get_price()`

Exercise 3: Shopping Cart

Create:

`class ShoppingCart:`

Keep the item list internally.

Add:

`add_item()`
`remove_item()`
`get_items()`

Make sure `get_items()` does not expose the original mutable list directly.

Exercise 4: Resource

Create:

`class Resource:`

Store:

`title`
`category`
`_views`

Add:

`open()`
`views`
`is_popular()`

Use a property for `views`.

---

44. Mini Quiz

1. What is encapsulation?

A. Creating loops inside classes
B. Bundling data and behavior while controlling access to internal state
C. Using only private variables
D. Creating multiple objects

Answer: B

---

2. What does `_balance` generally communicate in Python?

A. It is encrypted
B. It cannot be accessed
C. It is intended for internal use
D. It is a class attribute

Answer: C

---

3. What does `__balance` trigger?

A. Inheritance
B. Name mangling

C. A class method
D. A static method

Answer: B

---

4. Does `__balance` make data completely secure?

Answer: No. Python's name mangling is not a security mechanism.

---

5. Why use methods such as `withdraw()` instead of directly changing `balance`?

Answer: The method can enforce validation and business rules while keeping the state changes controlled.

---

6. Can encapsulation help reduce coupling?

Answer: Yes. Other parts of the program can depend on a public interface instead of internal implementation details.

---

7. Why can returning an internal list directly weaken encapsulation?

Answer: The caller may modify the original list and therefore change the object's internal state unexpectedly.

---

45. Interview Questions

Q1. What is encapsulation in OOP?

Encapsulation is the practice of bundling related data and behavior into an object while controlling how the object's internal state is accessed or modified.

Q2. Does Python have private variables?

Python does not enforce private variables in the same way as some languages. It uses conventions such as `_name` and name mangling with `__name`.

Q3. What does a single underscore mean?

A leading underscore communicates that an attribute or method is intended for internal use.

Q4. What is name mangling?

Python transforms names beginning with two underscores, such as `__value`, into a class-specific form to reduce accidental name conflicts.

Q5. Is name mangling a security feature?

No. It is a language mechanism for avoiding accidental access and naming conflicts.

Q6. How does a property support encapsulation?

A property allows attribute-like access while giving the class control over how the value is calculated, retrieved, or assigned.

Q7. Why is validation often placed inside a class?

Because the class owns the state and its rules. Keeping validation there helps prevent invalid states and avoids duplicating business rules throughout the application.

Q8. What is an invariant?

An invariant is a condition that should remain true for an object's valid state.

For example:

`balance >= 0`

Q9. Is encapsulation the same as abstraction?

No.

Encapsulation focuses on bundling and controlling state and behavior. Abstraction focuses on exposing useful concepts while hiding unnecessary implementation details.

Q10. Should every attribute have a getter and setter?

No. Python encourages simple, readable interfaces. Getters, setters, and properties should be used when they provide meaningful control or behavior.

---

46. Encapsulation Cheat Sheet

Public attribute

`self.name`

Internal-use convention

`self._balance`

Name mangling

`self.__balance`

Controlled modification

`def withdraw(self, amount):`

```
    if amount <= self._balance:
        self._balance -= amount
```

Getter

```
def get_balance(self):
    return self._balance
```

Property

```
@property
def balance(self):
    return self._balance
```

Property setter

```
@balance.setter
def balance(self, value):
    self._balance = value
```

Core idea

```
Internal State
      
Rules + Validation
      
Public Interface
      
Outside Code
```

---

47. Key Takeaways

1. **Encapsulation** keeps related data and behavior together.
2. It helps control how an object's internal state is accessed and modified.

3. Python attributes are public by default.
4. `_name` is a convention for internal-use attributes.
5. `__name` triggers name mangling.
6. Name mangling is not a security mechanism.
7. Methods can provide controlled ways to change object state.
8. Validation inside a class helps maintain valid object state.
9. Properties provide attribute-like access with controlled behavior.
10. Returning internal mutable objects directly can weaken encapsulation.
11. Encapsulation can reduce coupling between different parts of a program.
12. Not every attribute needs to be hidden.
13. Not every attribute needs a getter and setter.
14. Good encapsulation creates a clear boundary between **public behavior** and **internal implementation**.

The central idea is:

```
Don't make outside code manage
the object's internal rules.

Let the object manage its own state.
```

Visit haas.dev for more resources and guides.

haas.dev

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