Defining a Class

Classes are blueprints for objects. Use PascalCase.

```
class SolarPanel:
    def __init__(self, brand, watts, price):
        self.brand = brand
        self.watts = watts
        self.price = price

    def value_ratio(self):
        return self.watts / self.price

    def display(self):
        ratio = self.value_ratio()
        print(f"{self.brand}: {ratio:.2f} W/$")
```

Creating Objects (Instances)

```
panel = SolarPanel("SunPower", 440, 450)
print(panel.brand)        # SunPower
print(panel.value_ratio())  # 0.978
panel.display()
```

Each object has its own attribute values.

Understanding `self`

`self` refers to **the current object**. Every method's first parameter is `self`.

- `self.brand` – access this object's brand
- `self.value_ratio()` – call another method on this object

```
class Counter:
    def __init__(self):
        self.count = 0
    def increment(self):
        self.count += 1
```

Two objects have **independent** state:

```
c1 = Counter(); c2 = Counter()
c1.increment(); c1.increment()
print(c1.count)  # 2
print(c2.count)  # 0
```

OOP Terminology

Term	Meaning
Class	Blueprint/template for objects
Object	Instance created from a class
Attribute	Data stored in an object (`self.x`)
Method	Function defined inside a class
`__init__`	Constructor — runs when object is created
`self`	Reference to the current object

Encapsulation

Keep data **private** and control access through methods.
Convention: prefix with _ (protected) or __ (private).

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

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return True
        return False

    def get_balance(self):
        return self.__balance

acc = SolarAccount(100)
acc.deposit(50)
print(acc.get_balance())  # 150
# acc.__balance  # AttributeError!
```

Methods **validate** input before modifying state.

Classes Working Together

Objects can contain other objects:

```
class KnowledgeBase:
    def __init__(self):
        self.faqs = {"rebate": "Up to $1400"}
    def search(self, keyword):
        return self.faqs.get(keyword)

class Chatbot:
    def __init__(self):
        self.kb = KnowledgeBase()  # has-a
    def respond(self, question):
        answer = self.kb.search(question)
        return answer if answer else "Unknown"
```

Four OOP Principles

Principle	Meaning
Abstraction	Hide complexity behind simple interface
Encapsulation	Bundle data + methods; control access
Generalisation	Design general classes for reuse
Inheritance	Create specialised classes from a parent

Common Errors

Forgetting `self` – first parameter of every method must be `self`.
`def display(self):` not `def display():`.

`self.x` vs `x` – `self.x` is an attribute; `x` alone is local.
Forgetting `self.` means the value is lost after the method ends.

Calling method without `()` – `panel.display` returns the method object.
Use `panel.display()` to actually run it.

Modifying attributes directly – breaks encapsulation.
Use methods like `deposit()` instead of `account._balance = 999`.