Python Classes and Objects

Table of contents

An **object** bundles data (**attributes**, comprising its state) with **methods** (functions that act on that data) — the object has a notion of *self*.

Defining a class

```
class ClassName():
    def __init__(self, ...):
        self.attribute = ...

    def some_method(self, ...):
        return ...
```

- Class names use `CamelCaps` by convention.
- `__init__` is the **initializer**, run automatically when an instance is created.
- Every method's first parameter is `self` — Python supplies it automatically; never pass an argument for it explicitly.

Instantiation and attributes

```
>>> p = ClassName(...)      # creates an instance, passing args to __init__
>>> p.attribute              # dot notation accesses instance variables
```

Equality vs. aliasing

- Two separately-constructed instances with identical attributes are **not** `==` by default (equality compares identity unless a class defines otherwise).
- Assigning `q = p` makes `q` an **alias** for the same object as `p` — mutating one is visible through the other, and `p == q` holds because they're literally the same object.

Private variables, getters, and setters

A leading underscore (`self._value`) signals that an attribute is intended for internal use only — a *convention*, not an enforced restriction; nothing stops external code from reading or writing it directly (which is bad practice).

Prefer exposing controlled access via methods:

```
def get_value(self) -> int:            # getter
    return self._value

def set_value(self, x: int) -> None:   # setter
    if x < 0:
        raise ValueError
    self._value = x
```

Python's `@property`/`@<name>.setter` decorators let a getter/setter pair be used with ordinary attribute syntax (`obj.name` / `obj.name = value`) instead of explicit method calls.

Encapsulation

Encapsulation means keeping an object's data safe inside the class, exposed only through its attributes and methods.