Python Classes and Objects

Table of contents

Defining a class 1
Instantiation and attributes 2
Equality vs. aliasing 2
Private variables, getters, and setters 2

An **object** bundles data (**fields**/**attributes**, comprising its state) with **methods** (class-scoped functions that act on that data). An object can reference and change its own state, and so has a notion of *self*.

A **class** is the blueprint; an **object** is an instance of that class. Keeping the data inside the class and reaching it only through the class's own attributes and methods is **encapsulation**.

Introduced in 2025-09-09-object-oriented-programming[1] (Lecture 7C).

Defining a class

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

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

- Class names use `CamelCaps` by convention (`ClassNamesAreLikeThis`).
- `__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.

[1] courses/csse1001/2025-09-09-object-oriented-programming.pdf

- Think of the class body as a blueprint with placeholders: `self` has an `x`, `self` has a `y`. On **instantiation**, `self` becomes the particular object being built and each placeholder becomes a real attribute stored in it, so many instances can share one blueprint while each holds its own values.

Instantiation and attributes

```
>>> p = ClassName(...)       # creates an instance, passing args to __init__
>>> p.attribute              # dot notation accesses instance variables
>>> p
<__main__.ClassName object at 0x10a9e0dd8>
>>> type(p)
<class '__main__.ClassName'>
```

A method is called the same way: `<object_variable>.<method>(<parameters>)`. Calling it as a bare name is a `NameError` — it only exists on the object.

Equality vs. aliasing

- Two separately-constructed instances with identical attributes are **not** `==` by default: the default comparison is identity, so equal-looking objects at two different addresses compare `False`. Defining what equality *means* for a class is the job of `__eq__` — see python-dunder-methods[2].
- 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:
```

[2] courses/csse1001/python-dunder-methods.pdf

```
        raise ValueError
    self._value = x
```

The `raise` is the reason the setter exists: it is the one place the class can reject a bad value before it becomes state (see python-exceptions[3]).

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, so validation still runs while call sites stay readable.

[3] courses/csse1001/python-exceptions.pdf