Python Dunder (Magic) Methods

Table of contents

Common dunder methods 1
`__repr__` vs. `__str__` 2
Type-hinting a class inside its own body 2
Overloadable operators 3
Instance variables vs. class variables 3
Worked example: `Vector2D` 4

Dunder ("double underscore") or **magic** methods are special methods, named `__like_this__`, that Python calls automatically to implement built-in behaviour for a type — instantiation, printing, equality, arithmetic operators, and more. You don't call them directly; Python invokes them on your behalf when the corresponding syntax/built-in is used.

See 2025-09-16-dunder-methods[1] for the lecture note, and python-classes-and-objects[2] for classes generally.

Common dunder methods

Method	Called by	Purpose
`__init__(self, ...)`	`ClassName(...)`	Initializes a new instance. Returns `None` — it fills in an object that already exists rather than producing one.
`__str__(self)`	`print(obj)`, `str(obj)`	User-friendly display string.

[1] courses/csse1001/2025-09-16-dunder-methods.pdf
[2] courses/csse1001/python-classes-and-objects.pdf

Method	Called by	Purpose
`__repr__(self)`	the REPL, `repr(obj)`	Unambiguous, developer-facing string — ideally a valid Python expression that recreates the object via `eval()`.
`__eq__(self, other)`	`obj == other`	Custom equality (instead of identity).
`__add__(self, other)`	`obj + other`	Addition.
`__sub__(self, other)`	`obj - other`	Subtraction.
`__neg__(self)`	`-obj`	Unary negation.
`__len__(self)`	`len(obj)`	Length.

Every binary method takes the left operand as `self` and the right one as `other`.

`__repr__` vs. `__str__`

- `__repr__` is for the programmer: unambiguous, meant for debugging/logging, ideally `eval(repr(obj))` reconstructs an equal object. The round trip is `obj` → `repr()` → source text → `eval()` → a second object, whose `repr` matches the first's.
- `__str__` is for the user: a friendly, readable string, used by `print()`.
- If only `__repr__` is defined, `print()` falls back to it. If neither is defined, printing an object shows its default `<... object at 0x...>` representation.

A consequence that catches people out: at the interactive prompt, `>>> obj` and `>>> print(obj)` can produce completely different text from the same object, because they go through different methods.

Type-hinting a class inside its own body

Methods like `__add__` want to say they return the class being defined, but the class object doesn't exist yet while the body is executing:

```
from __future__ import annotations   # makes annotations lazy (strings)

class Fraction():
    def __add__(self, other) -> Fraction:   # legal because of the import
        ...
```

Without that import, `-> Fraction` is evaluated immediately and raises `NameError`; the alternative is to quote it, `-> "Fraction"`.

Overloadable operators

Binary	Method	Unary	Method	Comparison	Method
`+`	`__add__`	`-`	`__neg__`	`<`	`__lt__`
`-`	`__sub__`	`abs`	`__abs__`	`<=`	`__le__`
`*`	`__mul__`	`~`	`__invert__`	`==`	`__eq__`
`**`	`__pow__`			`!=`	`__ne__`
`//`	`__floordiv__`			`>`	`__gt__`
`/`	`__truediv__`			`>=`	`__ge__`

Operators can be defined in terms of each other rather than from scratch — `Fraction.__sub__` is just `return self + -other`, i.e. `__add__` composed with `__neg__`.

Instance variables vs. class variables

An **instance variable** (`self.x = ...`) belongs to one specific object — each instance has its own copy. A **class variable** (declared directly in the class body) is shared by *all* instances of the class, and can be accessed either via an instance (`obj.class_var`) or via the class itself (`ClassName.class_var`), without needing any instance at all.

```
class Clicker():
    _all_clicks = 0   # class variable -- shared by every instance

    def __init__(self) -> None:
        self._clicks = 0   # instance variable -- unique per object

    def click(self) -> None:
        self._clicks += 1
        Clicker._all_clicks += 1
```

Why `Clicker._all_clicks += 1` and not `self._all_clicks += 1`. `self._all_clicks += 1` expands to `self._all_clicks = self._all_clicks + 1`. The read on the right-hand side finds the *class* attribute (attribute lookup falls back to the class when the instance has no such name), but the assignment on the left always writes an *instance* attribute. So the first click on each object creates a private `_all_clicks` on that object that shadows the shared one forever, and the class-level total stays at 0. Naming the class explicitly is what makes the update land on the shared cell.

Reading through an instance is fine — `c._all_clicks` returns the class's value, which is why the transcript in 2025-09-16-dunder-methods[3] shows `(6, 6, 6)`. It is only *assignment* through an instance that breaks the sharing.

Worked example: `Vector2D`

A fuller worked example combining several dunder methods together. The length is Euclidean: for `(2, 3)` it is `sqrt(13)`, `3.605551275463989`.

```
import math

class Vector2D():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def length(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __repr__(self):
        return f"Vector2D(x={self.x}, y={self.y})"

    def __str__(self):
        return f"2D Vector: ({self.x}, {self.y}) --- length: {self.length()}"

    def __eq__(self, other):
        return isinstance(other, Vector2D) and self.x == other.x and self.y == other.y

    def __add__(self, other):
        if not isinstance(other, Vector2D):
            return NotImplemented
        return Vector2D(self.x + other.x, self.y + other.y)

    def __len__(self):
        return 2
```

```
>>> u = Vector2D(2, 3)
>>> v = Vector2D(4, 5)
>>> print(u + v)
2D Vector: (6, 8) --- length: 10.0
```

[3] courses/csse1001/2025-09-16-dunder-methods.pdf

```
>>> print(u)          # calls __str__
2D Vector: (2, 3) --- length: 3.605551275463989
>>> repr(u)            # calls __repr__
'Vector2D(x=2, y=3)'
>>> u == v              # calls __eq__
False
>>> len(u)               # calls __len__
2
```

`print(u + v)` uses two magic methods in one expression: `__add__` builds the new vector, then `print` displays it through `__str__`.

Returning `NotImplemented` (rather than raising or returning `False`) from a method like `__add__` is the standard way to signal "I don't know how to combine these two types" — it lets Python fall back to the other object's reflected method, or raise a clean `TypeError` if nothing handles it.