Dunder (Magic) Methods

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 8A's technical content. See python-dunder-methods[2] for the full reference on `__init__`, `__str__`, `__repr__`, and operator overloading.

Today's outline

- Recap: class vs. object
- Underscores: anonymous variables, private variables, and dunder names
- Magic methods: `__init__`, `__str__`, `__repr__`, `__eq__`, `__add__`, `__sub__`
- Overloadable operator tables (binary, unary, comparison)
- Instance variables vs. class variables

Class vs. object

Aspect	Class	Object
Meaning	Blueprint/template for creating objects	Instance of a class with real data

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

Aspect	Class	Object
Represents	General concept or idea	Concrete entity based on a class
Defined by	`class` keyword	Instantiating a class
Example	`class Animal:`	`my_animal = Animal()`
Memory usage	No direct memory for data	Allocates memory for attributes
Purpose	Describes structure and behaviour	Performs actions and stores data
Analogy	House blueprint	Actual built house

Underscores

Python overloads the underscore with several distinct meanings (not an exhaustive list):

1. As **anonymous variables**, e.g. `for _ in [1, 2, 3]:` or `x, _, z = (1, 2, 3)`.
2. For giving special meaning to functions and names:
    - `_private` variables — a leading single underscore (convention only).
    - `__names__` — reserved for Python's **magic/dunder** methods, like `__init__()`.

III: Magic (overloading built-in functions)

We can create a new type, `Fraction`, with:

- **Data attributes**: numerator, denominator.
- **Methods**: arithmetic operations (`add`, `eq`, `sub`) to work with `+`, `==`, `-`; and a print-friendly representation.

Initialiser

Runs when the object is *instantiated* (created):

```
class Fraction():
    def __init__(self, numer: int, denom: int) -> None:
        self._numer = numer
        self._denom = denom
```

String representation (`__str__`)

Says what to display when *printing* the object:

```
>>> p = Fraction(2, 3)
>>> print(p)
<__main__.Fraction object at 0x7f95c625e9d0>
```

Without a `__str__`, printing an object just shows its default memory-address representation. Defining one fixes this:

```
>>> class Fraction():
...     def __str__(self) -> str:
...         return f"A fraction: {self._numer} / {self._denom}"   # must return a string
>>> p = Fraction(2, 3)
>>> print(p)
A fraction: 2 / 3
```

Representation (`__repr__`)

The **representation** of an object is what Python displays for it in the console, and should be enough information to re-instantiate the object:

```
>>> class Fraction():
...     def __repr__(self) -> str:
...         return f"{self._numer} / {self._denom}"
>>> p = Fraction(2, 3)
>>> p
2 / 3
```

This is equivalent to calling `print(repr(p))` or directly invoking `print(p.__repr__())` — but we don't manually invoke these methods; Python does.

`__repr__` vs. `__str__`

- `__repr__` is meant to be used by the **programmer**:
  - Unambiguous — it should clearly describe the object.
  - Meant for debugging, logging, and development, not end-users.

  - Its output should, if possible, be a valid Python expression that could recreate the object when passed to `eval()`:

```
>>> u
Vector2D(x=2, y=3)
>>> print(u)
2D Vector: (2, 3) --- length: 3.605551275463989
>>> u_copy = eval(repr(u))
>>> u_copy
Vector2D(x=2, y=3)
```

- `__str__` is meant for pretty prints (a user-friendly string, printed for the user).

Equality (`__eq__`)

We can specify that objects are equal for reasons other than sharing a memory location:

```
>>> class Fraction():
...     def __eq__(self, other) -> bool:   # note the use of 'other'
...         a, b = self._numer, self._denom
...         c, d = other._numer, other._denom
...         return a*d == b*c
>>> p = Fraction(4, 6)
>>> q = Fraction(2, 3)
>>> p == q
True
```

Addition (`__add__`)

Instructs Python on how to add two objects together:

```
>>> from __future__ import annotations   # for the class' own type hint
>>> class Fraction():
...     def __add__(self, other) -> Fraction:
...         a, b = self._numer, self._denom
...         c, d = other._numer, other._denom
...         return Fraction(a*d + c*b, b*d)
>>> p = Fraction(2, 3)
>>> q = Fraction(1, 2)
>>> p + q
7 / 6
```

Subtraction (__neg__, __sub__)

```
>>> from __future__ import annotations
>>> class Fraction():
...     def __neg__(self) -> Fraction:
...         return Fraction(-self._numer, self._denom)
...     def __sub__(self, other) -> Fraction:
...         return self + -other
>>> p = Fraction(2, 3)
>>> q = Fraction(1, 2)
>>> p - q
1 / 6
```

Overloadable operators

Binary operator	Magic method
+	__add__
-	__sub__
*	__mul__
**	__pow__
//	__floordiv__
/	__truediv__

Unary operator	Magic method
-	__neg__
abs	__abs__
~	__invert__

Comparison	Magic method
<	__lt__
<=	__le__
==	__eq__
!=	__ne__
>	__gt__
>=	__ge__

Instance vs. class variables

Recall the class we wrote for counting clicks:

```
class Clicker():
    def __init__(self) -> None:
        self._clicks = 0   # each instance has its own

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

Can we calculate the number of clicks *across all counters*? We can use a **class variable**:

```
class Clicker():
    _all_clicks = 0   # every instance has access to this

    def __init__(self) -> None:
        self._clicks = 0

    def click(self) -> None:
        self._clicks += 1        # access instance variable
        Clicker._all_clicks += 1  # access class variable
```

```
>>> c = Clicker(); d = Clicker(); e = Clicker()   # semi-colons can be used instead of newli
>>> c.click(); c.click(); c.click();
>>> d.click(); d.click();
>>> e.click()

>>> (c._clicks, d._clicks, e._clicks)   # bad practice (accessing privates directly)
(3, 2, 1)

>>> (c._all_clicks, d._all_clicks, e._all_clicks)
(6, 6, 6)

>>> Clicker._all_clicks   # you don't even need an instance
6
```

> The exercise on the previous slide asked for a class called `Clicker`, but the companion `Clicker.py` file actually defines a class called `Counter` instead (matching the earlier Lecture 7C `Counter` exercise, plus extra `set_count`/`print_counter` methods) — the naming doesn't match the exercise prompt. The file's final two

lines, `d = Counter("second counter")`, also don't work: `Counter.__init__` only takes `self`, so passing an extra argument raises `TypeError: Counter.__init__() takes 1 positional argument but 2 were given`. This looks like leftover exploratory code rather than a demonstrated feature.

Summary

Classes (or objects) are like functions that maintain their state even after returning. Classes have attributes and methods, and provide a public interface — through setters and getters — for manipulating values considered private to the object.

Exercises

Task (Vectors). Notice that `+` concatenates lists:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Implement a `Vector` class so that we can do:

```
>>> x = Vector(1, 2)
>>> y = Vector(3, 4)
>>> x + y
<4, 6>
>>> -x
<-1, -2>
```

Starter code (unsolved in the source):

```
class Vector():
    def __init__(self, x: int, y: int):
        self._x, self._y = x, y

    def __add__(self, other):
        ...

    def __neg__(self):
        ...

    def __repr__(self):
        ...
```

Extension: try creating a `Vector` class that handles an arbitrary dimension. If two vectors of different sizes are added, `__add__` should raise a `ValueError`.

> The companion `magic.py` file contains a separate, fully-worked `Vector2D` class (2D-only, not the arbitrary-dimension extension) with `__init__`, `length()`, `__repr__`, `__str__`, `__eq__`, `__add__`, and `__len__` — useful as a worked reference for this style of task, even though it doesn't solve the exercise as stated (it's fixed at two dimensions and doesn't raise on mismatched sizes). See python-dunder-methods[3] for the full listing.

Task (Currency). Create a class for working with the currencies `AUD`, `EUR`, and `JPY`. Implement the `__repr__`, `__gt__`, and `__add__` magic methods — you'll need the dollar/euro/yen symbols, and to do currency conversions when adding different currencies together. Use: 1 AUD is 0.62 EUR; 1 AUD is 79.7 JPY.

Starter code (unsolved in the source):

```
class Currency():
    def __init__(self, value: float, currency: str) -> None:
        """ <currency> is one of 'AUD', 'EUR', 'JPY'. """
        self.value = value
        self.currency = currency

    def __repr__(self) -> str:
        ...

    def __add__(self, other) -> object:
        ...

    def __gt__(self, other) -> bool:
        ...
```

Task (Greeter). A fully worked example, using a class variable as a shared lookup table:

```
class Greeter():
    _lang_to_hello = {
        "FR": "Bonjour",
        "AU": "G'Day",
        "DE": "Hallo",
        "CN": "Ni Hao"
    }
```

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

```
    def __init__(self, country: str) -> None:
        self._country = country

    def greet(self) -> str:
        return Greeter._lang_to_hello[self._country]
```

```
>>> a = Greeter("FR"); b = Greeter("AU")
>>> c = Greeter("DE"); d = Greeter("CN")
>>> a.greet()
'Bonjour'
>>> b.greet()
"G'Day"
>>> c.greet()
'Hallo'
>>> d.greet()
'Ni Hao'
```

Next: 2025-09-16-representation-invariants[4] (Lecture 8B).

[4] courses/csse1001/2025-09-16-representation-invariants.pdf