Dunder (Magic) Methods

Table of contents

Outline 1
Class vs. object 1
Exercise: `Clicker` 2
Underscores 3
III: Magic (overloading built-in functions) 3
Instance vs. class variables 8
Summary 9
Exercises 9

Lecture 8A. See csse1001[1] for course logistics and python-dunder-methods[2] for the full reference on `__init__`, `__str__`, `__repr__`, and operator overloading. Diagrams below follow csse1001-diagram-notation[3].

Outline

- Recap: class vs. object
- Exercise: a `Clicker` tally counter
- 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

A class collects **attributes** on one side and **methods** on the other, and every object is one instantiation of it (Figure 1).

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

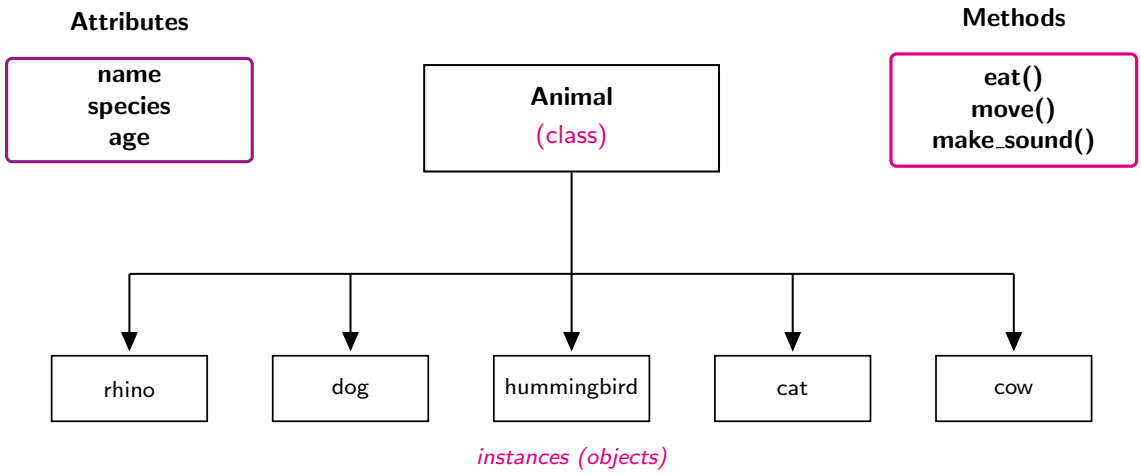

Figure 1: The class as blueprint: the attributes it declares (left, purple outline), the methods it provides (right, magenta outline), and the objects instantiated from it below. The arrows run *downward*, class to instance. Inheritance arrows run the other way, up into the parent.

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

Exercise: `Clicker`

Create an object called `Clicker` that simulates the functionality of a mechanical hand tally counter.

The class this produces is used again under Instance vs. class variables.

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)

Magic is part III of the OOP material. Parts I (structure) and II (methods) are in Lecture 7C (2025-09-09-object-oriented-programming[4]).

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:

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

```
>>> 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
```

`__str__` must return a string. Returning anything else is a `TypeError` raised at print time, not at definition time: nothing checks the return annotation when the `def` is executed.

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)
    ```

The round trip is `u` → `repr()` → `"Vector2D(x=2, y=3)"` → `eval()` → a second object that prints the same way. (The transcript shows the two `repr`s matching; it never evaluates `u == u_copy`.)

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

In the transcript above, `>>> u` and `>>> print(u)` produce *different* output from the same object: the first goes through `__repr__`, the second through `__str__`.

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
```

`4/6 == 2/3` because the method cross-multiplies (`4·3 == 6·2`) rather than comparing the stored pairs. The second parameter is named `other`, and the convention holds for every binary dunder, which receives the left operand as `self` and the right one as `other`.

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
```

The `from __future__ import annotations` line is there so that `-> Fraction` can name the class inside its own body, before the class object exists.

The last line shows `7 / 6` only because `__repr__` is in the class too: `__add__` builds a new `Fraction` and the console then displays it through `__repr__`. Two magic methods are cooperating in one line of output. Read literally, though, each of these transcripts re-enters `class Fraction():` from scratch at the prompt, so the class here has *only* `__add__`. With no `__init__`, `p = Fraction(2, 3)` would fail on line 7 with `TypeError: Fraction() takes no arguments`, long before `p + q` was reached. The transcripts only work if you read the class as accumulating from one to the next.

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
```

`__sub__` is defined entirely in terms of `__add__` and `__neg__`: `p - q` becomes `p + (-q)` = `Fraction(2·2 + (-1)·3, 3·2)` = `Fraction(1, 6)`.

The negation method is `__neg__`, not `__neq__`. Spelled `def __neq__(self)`, the class would define an unused `__neq__` and leave `-other` inside `__sub__` unsupported, so `p - q` would raise `TypeError: bad operand type for unary -: 'Fraction'` rather than printing `1 / 6`.

What "magic" actually means

Every method above is invoked *for* you. You write the ordinary Python in the left column of Figure 2; the interpreter turns it into the call on the right. The one place above where a dunder method is called by name is `print(p.__repr__())`. Even that is not an operator being invoked, and it is the equivalent you *don't* type. Dunder *names* do appear elsewhere, in `__main__` inside the default repr, in the `from __future__ import annotations` import, in the three operator tables below and in the wording of the Vector and Currency exercises, but never as a call you write.

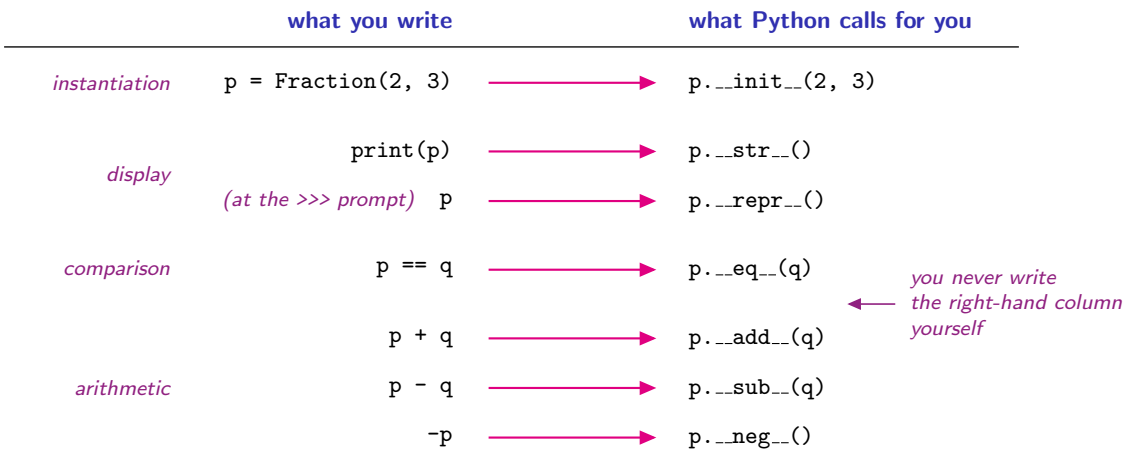

Figure 2: What 'magic' means: you write the left-hand column, Python calls the right-hand one. We don't manually invoke these methods; Python is the one doing it. One nuance the arrows flatten: `__init__` really runs on the freshly created object *before* the name `p` is bound to it.

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__`

Comparison	Magic method
`!=`	`__ne__`
`>`	`__gt__`
`>=`	`__ge__`

Instance vs. class variables

The `Clicker` class for counting clicks:

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

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

To calculate the number of clicks *across all counters*, 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
(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 `(3, 2, 1)` against `(6, 6, 6)` is the whole point: three objects, three private counters, one shared total.

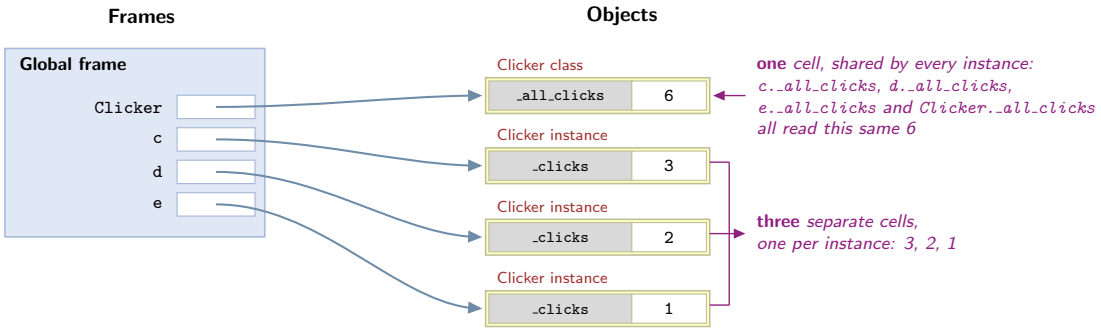

Figure 3: The click counts drawn as a memory picture, in Python Tutor's idiom. Three instances carry three separate `_clicks` cells; the single `Clicker class` object carries one `_all_clicks` cell, which is why `c._all_clicks`, `d._all_clicks`, `e._all_clicks` and `Clicker._all_clicks` all read 6.

Figure 3 also explains why `click` writes `Clicker._all_clicks += 1` rather than `self._all_clicks += 1`: the class variable lives on the class object, so the class object is what has to be updated. See python-dunder-methods[5] for what goes wrong if you write it the other way.

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). `+` concatenates lists:

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

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

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

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

Both output lines come back at a bare prompt, so the `<4, 6>` format is `__repr__`'s job, not `__str__`'s.

Starter code:

```
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 `Vector2D` in the `__repr__` vs. `__str__` transcript earlier is a related class, with a `__repr__` of `Vector2D(x=2, y=3)` and a `__str__` that also reports a length. It is 2D-only, so it is not the arbitrary-dimension extension asked for here. python-dunder-methods[6] has a `Vector2D` consistent with that transcript.

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:

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

```
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:
        ...
```

This starter breaks the convention used everywhere else: `value` and `currency` are public, where every other class here uses `_value`-style private attributes.

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"
    }

    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'
```

`b.greet()` comes back as `"G'Day"`, in double quotes where the other three are in single quotes: that is the repr Python prints for a string containing an apostrophe.

This is the class-variable idea from Figure 3 used for something other than counting: one dictionary, built once on the class object, read by every instance. Four `Greeter` objects, four `_country` strings, one `_lang_to_hello`.

Next: 2025-09-16-representation-invariants[7] (Lecture 8B).

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