CSSE1001 — Week 8 Notes

Table of contents

Dunder (Magic) Methods

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

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

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
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.

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

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"
    }

    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).

Representation Invariants

See csse1001[5] for course logistics — this note covers Lecture 8B's technical content. See python-representation-invariants[6] for the full reference on writing and enforcing invariants.

Today's outline

- What representation invariants are, and why they matter
- Documenting invariants in docstrings
- Enforcing invariants with `assert` and a `_check_invariants()` helper
- Encapsulation and avoiding exposed mutable state
- Worked example: `DroneFlight`

What are representation invariants?

Representation invariants are conditions or properties that must remain true about the internal state of an object throughout its lifetime (usually enforced during development).

Why do they matter?

- Discover and prevent bugs and inconsistent behaviour.
- Simplify the implementation of methods (you can assume a valid starting state).
- Make code more maintainable and robust.
- Help detect errors early.
- Serve as documentation for developers.

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

Examples

- A stack's size must never be negative.
- A list's length must equal the number of elements it contains.
- All elements in a set must be unique.
- A fraction's denominator cannot be zero.
- A date must follow calendar rules (e.g. no 30 Feb).
- For free/basic X (Twitter) accounts, the standard limit is 280 characters per tweet.

Writing representation invariants

Clearly state invariants in **docstrings**, underneath a class's attributes:

```
class Fraction():
    """Represents a mathematical fraction.

    Representation Invariants:
    - denominator != 0
    - if fraction is zero, it is represented as 0/1
    - if fraction is negative, numerator is negative
    """
```

Enforcing representation invariants

Assertions

Directly check conditions in the initialiser and setter methods, using an `assert` statement to raise an `AssertionError` if an assumption isn't met:

```
assert <statement that should be true>, "Error message if not True"
```

- Assertions can be disabled in production when running Python with optimization (`python -O`).
- Assertions are meant to find bugs (e.g. invalid states) *during development*.

```
class Fraction():
    def __init__(self, numer: int, denom: int) -> None:
        assert denom != 0, "Denominator cannot be zero."
        assert isinstance(numer, int), "Numerator must be an integer."
        assert isinstance(denom, int), "Denominator must be an integer."
```

```
        # Ensure denominator is positive
        if denom < 0:
            numer, denom = -numer, -denom
        self._numer = numer
        self._denom = denom

        # Special case for zero
        if self._numer == 0:
            assert self._denom == 1, "Zero must be 0/1"
```

Private attributes and encapsulation

- Use private (`_variable`) names to discourage direct access.
- Provide public methods that maintain the invariants.
- **Never expose mutable objects directly**:

```
class Team():
    def __init__(self):
        self._members = []   # internal mutable state

    # classic getter (dangerous!)
    # BAD!
    def get_members(self):
        return self._members

    # instead, return a copy
    # GOOD!
    # def get_members(self):
    #     return list(self._members)  # caller can't modify internal state directly

t = Team()
members = t.get_members()   # gets the actual list inside the class
members.append("Alice")      # modifies it directly!

print(t.get_members())        # ['Alice'] <-- internal state was changed externally
```

Helper methods

Define a private method (e.g. `_check_invariants()`) that validates the object's state, and call it after any state changes:

state change → `_check_invariants()` → valid

When to call `_check_invariants()`:

- At the end of `__init__`.
- At the end of methods that modify object state.
- At the beginning of methods that rely on invariants being true.

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

    def _check_invariants(self) -> None:
        """Verify that representation invariants hold."""
        assert isinstance(self._numer, int), "Numerator must be an integer."
        assert isinstance(self._denom, int), "Denominator must be an integer."
        assert self._denom != 0, "Denominator cannot be zero."
        assert self._denom > 0, "Denominator must be positive."
        if self._numer == 0:
            assert self._denom == 1, "Zero must be 0/1"
```

Best practices

- Document invariants in docstrings.
- Centralise invariant checking in a single method.
- Do not expose methods that could violate invariants.
- Create comprehensive test cases focusing on edge cases.
- Python's type hints can express some invariants.
- Balance strictness with practicality.

Exercise: `DroneFlight`

You have joined a start-up that records quick test flights for hobby drones. Write a minimal `DroneFlight` class that always keeps its state valid by enforcing these representation invariants:

- **Drone ID** — exactly six alphanumeric characters (e.g. `"A1B2C3"`).
- **Altitude** — must stay within 0 m–120 m, the CASA (Civil Aviation Safety Authority) legal ceiling for recreational drones.

- **Battery** — an integer percentage in the range 0–100.

Solution

```
class DroneFlight:
    """
    Model a quick test-flight for a small hobby drone.

    Attributes:
    _id (str) : 6-character alphanumeric flight identifier.
    _altitude (float): Current altitude in metres above launch point.
    _battery (int): Remaining battery charge as a percentage (0-100).

    Representation invariants:
    - _id is a 6-character alphanumeric string
    - 0 <= _altitude <= 120
    - 0 <= _battery <= 100
    """

    _ID_LEN = 6         # one place to change if the rule changes
    _MAX_ALT = 120.0    # metres (CASA limit)

    def __init__(self, flight_id: str) -> None:
        self._id = flight_id
        self._altitude = 0.0
        self._battery = 100
        self._check_invariants()

    def _check_invariants(self) -> None:
        assert (
            isinstance(self._id, str)
            and self._id.isalnum()
            and len(self._id) == self._ID_LEN
        ), "ID must be a 6-character alphanumeric string."
        assert 0.0 <= self._altitude <= self._MAX_ALT, "Altitude out of range."
        assert 0 <= self._battery <= 100, "Battery out of range."

    # Getter and setter methods
    def get_flight_id(self) -> str:
        """Return the immutable flight identifier."""
        return self._id
```

```
    def get_altitude(self) -> float:
        """Return current altitude in metres."""
        return self._altitude

    def get_battery(self) -> int:
        """Return remaining battery charge (percentage)."""
        return self._battery

    def set_altitude(self, value: float) -> None:
        """Set altitude, clamping to legal ceiling."""
        if not isinstance(value, (int, float)):
            raise TypeError("Altitude must be a number.")
        if not 0.0 <= value <= self._MAX_ALT:
            raise ValueError(f"Altitude must be 0-{self._MAX_ALT} m.")
        self._altitude = float(value)
        self._check_invariants()

    def __repr__(self) -> str:
        return (f"DroneFlight({self._id}, "
                f"alt={self._altitude:.1f} m, bat={self._battery} %)")

    def ascend(self, metres: float) -> None:
        """Climb *metres* metres (cannot exceed the legal ceiling)."""
        if metres < 0:
            raise ValueError("ascend() expects a non-negative distance.")
        self.set_altitude(min(self._altitude + metres, self._MAX_ALT))

    def land(self) -> None:
        """Land the drone and consume 5 % battery."""
        self.set_altitude(0.0)
        self._battery = max(self._battery - 5, 0)
        self._check_invariants()

>>> d = DroneFlight("ABC123")
>>> d.ascend(50)
>>> print(d)
DroneFlight(ABC123, alt=50.0 m, bat=100 %)

>>> d.altitude = 200
Caught: Altitude must be 0-120 m.

>>> DroneFlight("BAD!")
```

```
Caught: ID must be a 6-character alphanumeric string.

>>> d.land()
DroneFlight(ABC123, alt=0.0 m, bat=95 %)
```

> The `d.altitude = 200` line is presented as raising a "Caught" error, but as written this class only defines `set_altitude()` as an ordinary method — it has no `@property`/`@altitude.setter`. Plain attribute assignment like `d.altitude = 200` doesn't call `set_altitude()` at all; it silently creates a *brand-new*, unrelated `altitude` attribute (alongside the real `_altitude`) and raises nothing. To actually trigger the validation shown, the demo would need `d.set_altitude(200)` wrapped in a `try/except (TypeError, ValueError) as e: print("Caught:", e)`. The companion `drone.py` script does contain the bare `d.altitude = 200` line with no such wrapper, confirming it wouldn't actually raise in practice. The second demo line (constructing `DroneFlight("BAD!")`) is legitimate, though — `__init__` really does call `_check_invariants()`, so an invalid ID genuinely raises an `AssertionError` there (assuming it's likewise wrapped in a `try/except AssertionError`).

Summary

- Representation invariants define the valid states of an object.
- They help catch bugs early and document assumptions (mostly during development time).
- We can use `_check_invariants()` to verify invariants, and call it after state changes.
- Good invariants are specific, testable conditions.

Next: composition and inheritance (Week 9).

Reference material

Python Dunder (Magic) Methods

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.

Common dunder methods

Method	Called by	Purpose
`__init__(self, ...)`	`ClassName(...)`	Initializes a new instance.
`__str__(self)`	`print(obj)`, `str(obj)`	User-friendly display string.
`__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.

`__repr__` vs. `__str__`

- `__repr__` is for the programmer: unambiguous, meant for debugging/logging, ideally `eval(repr(obj))` reconstructs an equal object.
- `__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.

Overloadable operators

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

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
```

Worked example: `Vector2D`

A fuller worked example combining several dunder methods together:

```
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
>>> 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
```

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.

Python Representation Invariants

A **representation invariant** is a condition or property that must remain true about an object's internal state throughout its lifetime. They're primarily a *development-time* tool: they help catch bugs early, simplify method implementations (since you can assume a valid starting state), and document the assumptions a class relies on.

Documenting invariants

State invariants in the class docstring, underneath its attributes:

```
class Fraction():
    """Represents a mathematical fraction.

    Representation Invariants:
    - denominator != 0
    - if fraction is zero, it is represented as 0/1
    """
```

Enforcing invariants

Assertions

```
assert <condition that should be true>, "message if it's not"
```

- Raises `AssertionError` if the condition is false.
- Disabled when Python is run with optimization (`python -O`) — so asserts should catch *bugs*, not perform real input validation that must always run (use `raise ValueError(...)` etc. for that).

The `_check_invariants()` helper pattern

Centralise all invariant checks in one private method, and call it:

- At the end of `__init__`.
- At the end of any method that mutates state.
- At the start of any method that relies on the invariants already holding.

```
def _check_invariants(self) -> None:
    assert self._denom != 0, "Denominator cannot be zero."
    ...
```

Encapsulation: don't expose mutable internals

Returning a direct reference to a private mutable attribute (like a list or dict) lets external code silently violate the class's invariants:

```
def get_members(self):        # BAD -- exposes the real list
    return self._members

def get_members(self):        # GOOD -- caller gets an independent copy
    return list(self._members)
```

Best practices

- Document invariants in docstrings.
- Centralise invariant checking in a single method.
- Don't expose methods/attributes that could let external code violate invariants.
- Write test cases that target edge cases.
- Type hints can express some invariants, but not all (e.g. numeric ranges still need explicit checks).
- Balance strictness with practicality — not every property needs an assertion.