CSSE1001 — Week 10 Notes

Table of contents

Advanced Inheritance

See csse1001[1] for course logistics — this note covers Lecture 10B's technical content. See python-inheritance[2] for the full reference, now extended with abstract base classes and MRO.

Today's outline

- A useful trick for Assignment 2: `type(c).__name__`
- Abstract base classes
- Method Resolution Order (MRO), and Python's inheritance models
- The diamond problem, and C3 linearisation
- `super()` and MRO

A useful trick for A2: getting the class name of an object

```
class Car():
    pass

c = Car()
print(type(c))          # <class '__main__.Car'>
print(type(c).__name__)  # 'Car' -- e.g. can be used in __repr__
```

`type(c).__name__` returns the class name as a plain string — handy inside a `__repr__` (see python-dunder-methods[3]) so it stays correct even if the class is renamed or subclassed.

Abstract base classes

It is common to have an **abstract base class** that doesn't have concrete methods/attributes, but *enforces contracts* in its children classes. You're not meant to instantiate objects from these abstract classes directly.

Example: an abstract `Shape` class has an `area()` method without a concrete implementation. Every (concrete) child class of `Shape` must provide a concrete implementation of `area()`. Abstract base classes can sometimes have concrete implementations for *some* of their methods too (especially if those are meant to be used as-is by child classes).

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

```
import math

# Abstract class -- defines the interfaces (child classes must implement these methods)
class Shape:
    def area(self) -> float:
        raise NotImplementedError("Subclasses must implement area()")

    def perimeter(self) -> float:
        raise NotImplementedError("Subclasses must implement perimeter()")

class Circle(Shape):
    def __init__(self, r: float):
        self.r = r
    def area(self) -> float:
        return math.pi * self.r * self.r
    def perimeter(self) -> float:
        return 2 * math.pi * self.r

class Rectangle(Shape):
    def __init__(self, w: float, h: float):
        self.w, self.h = w, h
    def area(self) -> float:
        return self.w * self.h
    def perimeter(self) -> float:
        return 2 * (self.w + self.h)


s = Shape()   # BAD -- you're not meant to create an object from this abstract base class
s.area()      # ERROR
```

> `Shape()` itself doesn't raise an error here — it's still just a regular class. The error only happens on `s.area()`, which raises `NotImplementedError`. This is a plain-Python convention, not an enforced restriction: nothing actually stops you from instantiating `Shape`, only from usefully calling its unimplemented methods.

Optional: enforcing this properly with `abc`

Using the standard-library `abc` module *does* enforce this at instantiation time:

```
from abc import ABC, abstractmethod

class Shape(ABC):
```

```
    @abstractmethod
    def area(self) -> float:
        ...
    @abstractmethod
    def perimeter(self) -> float:
        ...
    # (optional) concrete helper: allowed in an abstract class
    def describe(self) -> str:
        return f"{self.__class__.__name__}"
```

With this version, `Shape()` itself raises `TypeError: Can't instantiate abstract class Shape with abstract methods area, perimeter` — the abstract methods are enforced immediately, rather than only failing later when called.

Advanced inheritance

If a class inherits from two parents, and both parents have a method with the same name, which one does Python use?

Method Resolution Order (MRO)

MRO stands for **Method Resolution Order** — it defines the order in which Python looks through classes to find a method or attribute when it's called on an object. It determines which method gets called when there are multiple implementations, and is stored in `cls.__mro__`.

Python supports several inheritance models:

Single inheritance

A class inherits from a single parent class:

```
>>> class A(object):   # 'object' is the universal class
...     def __init__(self, x):
...         self.x = x
...     def f(self):
...         return self.x
...     def g(self):
...         return 2 * self.x
...     def fg(self):
...         return self.f() - self.g()
```

```
>>> a = A(3)
>>> a.x
3
>>> a.f()
3
>>> a.g()
6
>>> a.fg()
-3


>>> class B(A):
...     def g(self):   # override
...         return self.x ** 2

>>> b = B(7)
>>> b.x
7
>>> b.f()   # inherited from A
7
>>> b.g()   # overridden
49
>>> b.fg()  # inherited from A, but uses B's overridden g()
-42
```

Multilevel inheritance

A class inherits from a child class, which in turn inherits from another parent class — forming a linear parent → child → grandchild chain:

```
>>> class C(B):
...     def __init__(self, x, y):   # extends B's (and A's) __init__
...         super().__init__(x)
...         self.y = y
...     def fg(self):                # extends B's (and A's) fg
...         return super().fg() * self.y

>>> c = C(3, 5)
>>> c.x
3
>>> c.y
5
```

```
>>> c.f()    # inherited from B, from A
3
>>> c.g()    # inherited from B (overridden there)
9
>>> c.fg()   # extends B's fg: (3 - 9) * 5
-30
```

For multilevel inheritance, the MRO is simple — it just follows the chain from child to parent to grandparent, etc.

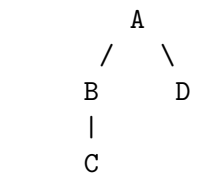
Hierarchical inheritance

Multiple child classes inherit from a single parent class:

```
>>> class D(A):
...     def f(self):   # override
...         return -2 * self.g()

>>> d = D(3)
>>> d.x
3
>>> d.f()   # overridden: -2 * g()
-12
>>> d.g()   # inherited from A
6
>>> d.fg()  # inherited from A, uses D's overridden f()
-18
```

The combined UML diagram for `A`, `B`, `C`, `D` above:



Multiple inheritance

A class inherits from *multiple* parent classes:

```
>>> class E(B, D):   # inherit from B and D
...     pass
```

```
  B   D
   \ /
    E
```

The diamond problem

- `E` inherits from both `B` and `D`.
- `B` and `D` both inherit from `A`.
- Which version of `A`'s methods should `E` use?

```
     A
    / \
   B   D
    \ /
     E
```

Python resolves this with **MRO C3 linearisation**:

- Child classes are checked before parents.
- Parents are checked in the order they are listed in the class definition.
- If a class appears multiple times in the MRO, only the *last* occurrence is kept.

```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.B'>,
 <class '__main__.D'>, <class '__main__.A'>,
 <class 'object'>]

>>> for cls in E.__mro__:
...     print(cls.__name__)
E
B
D
A
object
```

```
>>> e = E(3)
>>> e.x
3
>>> e.f()    # E has none; B has none of its own; D's f() is used
-18
>>> e.g()    # B's overridden g() is used (B comes before D in the MRO)
9
>>> e.fg()   # A's fg(): self.f() - self.g() = -18 - 9
-27
```

Even though `B` doesn't define its own `f()`, and `D` doesn't define its own `g()`, Python resolves each *name* independently by walking the MRO — `e.f()` finds `D`'s `f()` (since `B` has none of its own), while `e.g()` finds `B`'s `g()` (since `B` comes before `D` in the MRO).

A larger example, showing the general C3 rule (child before parents, parents in listed order, keep only the last occurrence of a repeated class):

```
>>> class A: pass
>>> class B: pass
>>> class C(A): pass
>>> class D(A, B): pass
>>> class E(C, D, B): pass
>>> print([cls.__name__ for cls in E.__mro__])
['E', 'C', 'D', 'A', 'B', 'object']
```

`super()` and MRO

The `super()` function follows the *MRO*, not just the immediate parent listed in the class definition:

```
class A:
    def ping(self):
        print("A")

class B(A):
    def ping(self):
        print("B")
        super().ping()

class C(A):
    def ping(self):
```

```
        print("C")
        super().ping()

class D(B, C):
    def ping(self):
        print("D")
        super().ping()

>>> D().ping()
D
B
C
A
```

`D`'s MRO is `[D, B, C, A, object]`. When `B.ping()` calls `super().ping()`, it doesn't jump straight to `A` (`B`'s statically-declared parent) — it calls the *next class in the actual runtime MRO of the instance*, which is `C`. This is what makes cooperative multiple inheritance work: each class's `super()` call advances one step through the shared MRO, regardless of what its own declared parent is.

> If a class in the chain doesn't call `super()` at all, the chain of calls simply stops there — the MRO itself is unaffected (it's purely a function of the inheritance structure), but fewer `ping()` implementations actually get executed. For example, if `B.ping()` omits its `super().ping()` call, `D().ping()` only prints `D` and `B` — `C` and `A` are never reached, even though they're still part of `D`'s MRO.

Summary

When a class inherits from multiple parents, it's possible for more than one parent to define the same method or attribute. To avoid confusion and ensure consistency, Python uses a rule called **Method Resolution Order (MRO)** to determine the order in which classes are searched. MRO follows a well-defined path based on class hierarchy and inheritance order, ensuring that each method or attribute is found in a predictable and logical way. This is especially important in complex inheritance situations like the diamond pattern.

Next: design patterns and MVC (Week 11).

Unified Modelling Language (UML)

See csse1001[4] for course logistics — this note covers Lecture 10A's technical content. See uml[5] for the full reference on UML diagram notation.

Today's outline

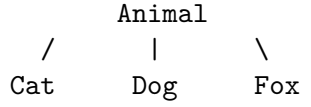
- Recap: inheritance vs. composition
- UML class diagrams: attributes, methods, and visibility
- Relationship diagrams: inheritance, association, multiplicity, composition, aggregation
- A larger, real-world example

Recap: inheritance vs. composition

Feature	Inheritance	Composition
Relationship	"Is-a"	"Has-a"
Flexibility	Less flexible (changes in parent affect children)	More flexible (components can be changed)
Reusability	Extends base class functionality	Contains objects that provide functionality
Example	`Dog` **is a** `Animal`	`Drone` **has a** `Camera`

Unified Modelling Language (UML)

A **UML diagram** is the standard way of illustrating relationships among classes — e.g. for our `Animal` class:



[4] courses/csse1001/csse1001.pdf
[5] courses/csse1001/uml.pdf

UML class diagram

A **class diagram** is a type of UML diagram that shows:

- Classes and their attributes/methods.
- Relationships (e.g. inheritance, composition, association).
- It's great for object-oriented design.

A class box has 3 parts, plus a visibility marker for each attribute/method:

- **Top section**: class name (e.g. `Animal`).
- **Middle section**: attributes (e.g. `name: str`).
- **Bottom section**: methods (e.g. `speak(): str`).
- **Visibility**: `-` (private), `+` (public), `#` (protected).

```
ClassName
-----------------------
-privateAttribute
+publicAttribute
#protectedAttribute
-----------------------
+method()
-privateMethod()
```

```
Animal
-----------------------
#name: str
#age: int
-----------------------
+info(): str
```

UML inheritance diagram

Each subclass inherits from `Animal` and overrides `speak()`. Inheritance is drawn as a solid line with a **hollow arrowhead** pointing from the subclass to the superclass:

```
    Animal
-----------------
#name: str
#age: int
-----------------
+info()
```

```
                ^
                | (inheritance)
    ----------------------------
    |           |               |
   Cat         Dog             Fox
 --------   -----------    ------------------
            -breed: str    -nationality: str
 --------   -----------    ------------------
 +speak()   +speak()       +speak()
                            +info()
```

UML association diagram

Association — drawn as a plain solid line — represents one class *using* or *knowing about* another, without owning it. For example, a `Cat` can `drink(Milk)`:

```
Animal             Milk
  ^           ----------------
  |           -expiration: str
Cat           ----------------
--------      +is_fresh(int): bool
+speak()
+drink(Milk)  ------ (association: "drinks") ------> Milk
```

UML multiplicity

Multiplicity shows how many objects are involved in a relationship:

Multiplicity	Meaning
`0..1`	Zero to one
`n`	Specific number
`0..*`	Zero to many
`1..*`	One to many
`m..n`	Specific number range

For example, an `Animal` can be associated with `1..*` `Vet`s, and a `Vet` is associated with `1..*` `Animal`s:

```
Animal  1..* ------ Associated ------ 1..*  Vet
```

UML composition diagram

Composition ("has a" relationship, filled diamond): here the "part" *cannot* exist independently of the "whole". For example, an `Animal` has exactly one `Heart`:

```
Animal    1     1    Heart
```

UML aggregation diagram

Aggregation (hollow diamond) is a weaker form of composition: here the "part" *can* exist independently of the "whole". For example, an `Owner` has zero or more `Animal`s (pets), but an `Animal` can exist without an `Owner`:

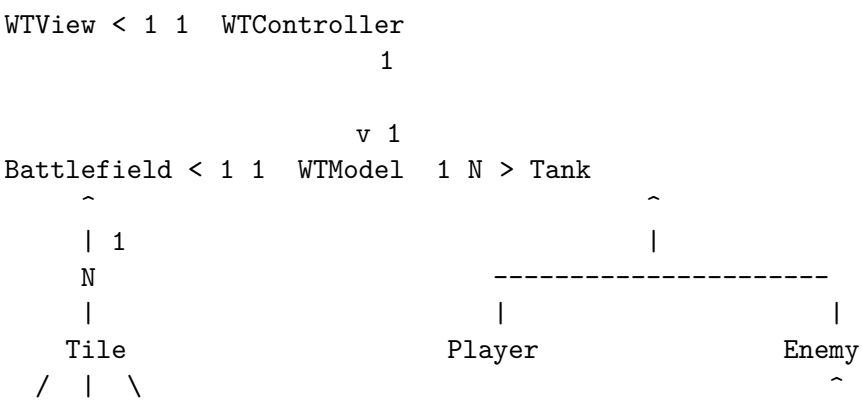
```
Owner      0..*    Animal
```

Exercise

Try modelling a UML diagram for the `DroneFlight` class (from 2025-09-23-composition[6]) and its child `DeliveryDrone` class (from 2025-09-23-inheritance[7]'s exercise) — left unsolved in the source.

A larger example

Real systems combine all of these relationships. A simplified tank-battle game architecture might look like:



[6] courses/csse1001/2025-09-23-composition.pdf
[7] courses/csse1001/2025-09-23-inheritance.pdf

The dashed arrows here represent *dependencies* between the controller, model, and view — this is the classic **Model-View-Controller (MVC)** pattern, which we'll cover properly next week.

Summary

UML is a visual language used to design and describe software systems. In object-oriented programming, class diagrams are one of the most-used UML tools. They help represent the structure of a system by showing classes, their attributes and methods, and the relationships between them — such as inheritance, association, aggregation, and composition. Using UML before writing code makes it easier to plan, communicate, and maintain software designs effectively.

Next: 2025-10-07-advanced-inheritance[8] (Lecture 10B).

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()`.

[8] courses/csse1001/2025-10-07-advanced-inheritance.pdf

Method	Called by	Purpose
`__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 Inheritance

Inheritance lets a class (the **subclass/child class**) reuse the attributes and methods of another class (the **superclass/parent class**), while adding new behaviour or overriding existing behaviour. It models an **"is-a"** relationship (a `DeliveryRobot` *is a* `Robot`) — contrast with python-composition[9]'s **"has-a"** relationship.

Syntax

```
class Parent():
    # parent class' implementation

class Child(Parent):
    # child class' implementation -- inherits everything from Parent
```

A subclass automatically gets all of its parent's attributes and methods. It can:

- **Add** new methods/attributes that only it has.
- **Override** an inherited method by redefining it with the same name.
- **Extend** an inherited method using `super()`, rather than fully replacing it.

[9] courses/csse1001/python-composition.pdf

`super()`

`super().method(...)` calls the *parent* class' version of a method — most commonly used inside an overridden `__init__` to reuse the parent's initialization logic before adding subclass-specific setup:

```
class DeliveryRobot(Robot):
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        super().__init__(name, batt_level)   # reuse Robot's __init__
        self.load_capacity = load_capacity
```

Terminology cheat-sheet

Given `class Student(Person):`

Term	Refers to
subclass / child class	`Student`
superclass / parent class	`Person`
"`Student` inherits from / extends `Person`"	the relationship between them

Polymorphism

Polymorphism ("many forms") describes an interface that works across different underlying types. Built-ins like `len()` are polymorphic (works on strings, lists, dicts, …). Classes that share a method name (whether via a common superclass or just by convention) are also polymorphic: the same call, e.g. `r.charge()`, produces different behaviour depending on which actual class `r` is an instance of.

When to use inheritance

Only when there's a genuine **"is-a"** relationship between the two classes. If the relationship is really "has-a" (one object *contains* or *uses* another), prefer python-composition[10] instead.

[10] courses/csse1001/python-composition.pdf

Abstract base classes

An **abstract base class** doesn't provide concrete implementations for some/all of its methods — it just defines the *interface* that every concrete subclass must implement. You're not meant to instantiate the abstract class directly.

The simplest (unenforced) version just raises an error from each method that subclasses must override:

```
class Shape:
    def area(self) -> float:
        raise NotImplementedError("Subclasses must implement area()")
```

This only fails when the unimplemented method is actually *called* — `Shape()` itself still succeeds. The standard-library `abc` module enforces this properly, raising `TypeError` at the point of instantiation:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self) -> float:
        ...

    # concrete helpers are still allowed alongside abstract methods
    def describe(self) -> str:
        return f"{self.__class__.__name__}"
```

With `ABC`, `Shape()` raises immediately: `TypeError: Can't instantiate abstract class Shape with abstract methods area`.

Method Resolution Order (MRO)

When a class inherits from multiple parents (or a chain of parents) that define the same method/attribute name, Python needs a deterministic rule for which one wins. That rule is the **Method Resolution Order (MRO)** — the order Python searches through classes to resolve a name on an object. It's stored on the class as `cls.__mro__` (or `cls.mro()`).

Python's inheritance models

- **Single inheritance**: `class B(A):` — one parent.
- **Multilevel inheritance**: `class C(B):` (where `B` itself inherits from `A`) — a linear chain; the MRO just follows the chain upward.
- **Hierarchical inheritance**: multiple children (`B`, `D`, …) inherit from the same parent `A`.
- **Multiple inheritance**: `class E(B, D):` — a class inherits from more than one parent directly.

The diamond problem and C3 linearisation

If `E(B, D)` and both `B` and `D` inherit from `A`, which version of `A`'s methods should `E` use? Python resolves this with **C3 linearisation**:

- Child classes are checked before parents.
- Parents are checked in the order they're listed in the class definition.
- If a class would appear more than once, only its *last* occurrence is kept.

```
>>> class E(B, D): pass
>>> [cls.__name__ for cls in E.__mro__]
['E', 'B', 'D', 'A', 'object']
```

Each attribute/method name is resolved independently by walking this list and taking the first class that defines it — so two different method calls on the same object can effectively "come from" two different classes in the hierarchy.

`super()` follows the MRO

`super()` doesn't call the immediate parent named in the `class` statement — it calls the *next class after the current one in the instance's actual MRO*. This is what makes cooperative multiple inheritance work: as long as every class in the chain calls `super()`, a single call can ripple through every class in the MRO exactly once, in order.

```
class A:
    def ping(self): print("A")
class B(A):
    def ping(self): print("B"); super().ping()
class C(A):
    def ping(self): print("C"); super().ping()
class D(B, C):
    def ping(self): print("D"); super().ping()
```

```
>>> D().ping()
D
B
C
A
```

If any class in the chain omits its `super()` call, the chain simply stops there for that call — the MRO itself doesn't change (it's purely a function of the class hierarchy), but fewer methods actually get executed.

UML Class Diagrams

UML (Unified Modelling Language) diagrams are the standard, language-agnostic way of visually describing the classes in a system and the relationships between them — useful for planning a design *before* writing code, and for communicating that design to others.

Class boxes

A class box has three parts:

```
ClassName
-----------------------
attributes
-----------------------
methods()
```

Each attribute/method is prefixed with a **visibility** marker:

Symbol	Visibility
`-`	private
`+`	public
`#`	protected

Relationship types

Relationship	Line style	Meaning
Inheritance	solid line, hollow (open) triangle arrowhead pointing to the parent	"is-a" — subclass inherits from superclass
Association	plain solid line	one class uses/references another, without ownership
Aggregation	solid line, hollow (open) diamond at the "whole" end	weak "has-a" — the part *can* exist independently of the whole
Composition	solid line, filled (solid) diamond at the "whole" end	strong "has-a" — the part *cannot* exist independently of the whole

Multiplicity

Multiplicity annotations on association/aggregation/composition lines describe how many objects on each end participate in the relationship:

Notation	Meaning
`0..1`	zero to one
`n`	an exact number
`0..*`	zero to many
`1..*`	one to many
`m..n`	a specific range

Why use UML?

- Plan a class hierarchy/object graph before writing any code.
- Communicate a design to teammates without needing to read source code.
- Document the intended structure and relationships for future maintenance.
- Class diagrams are the most commonly used UML diagram type in object-oriented design, but UML also includes other diagram types (e.g. sequence diagrams, use-case diagrams) not covered here.