Lecture 11 — Classes and object-oriented design for simulation code

C1 Research Computing — MPhil in Data Intensive Science, Michaelmas 2026

Boris Bolliet

University of Cambridge

Code and material

```
$ git clone https://github.com/borisbolliet/attractor.git ~/GitHub/attractor  # once
$ cd ~/GitHub/attractor
$ git fetch --tags
$ git switch --detach lecture-11
$ source setup_env.sh
$ python -m doctest sessions/lecture11/*.txt
```

- `git stash` first if `git status` lists local edits.
- The plain examples are in `sessions/lecture11/plain_*.txt`. The other session files use the project.
- `doctest` prints nothing when every output matches.
- `git diff lecture-10 lecture-11` shows what the lecture adds.

Outline

- Classes and attribute lookup: instances, the class dictionary, the MRO, bound methods, `type` and `object`
- Special methods: `len`, indexing, `repr`, `==` and a call
- Copies of objects that hold arrays: `copy.copy`, views, `copy.deepcopy`
- Interfaces, dataclasses, composition, and one function over several classes
- Live demonstration: the sessions of `sessions/lecture11/`

A class bundles data with the functions that act on it. Attribute lookup, special methods and copying follow fixed rules.

Classes and instances

```
>>> class Point:
...     unit = "cm"
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def norm(self):
...         return (self.x**2 + self.y**2) ** 0.5
>>> p = Point(3, 4)
>>> vars(p), p.unit, p.norm()
({'x': 3, 'y': 4}, 'cm', 5.0)
```

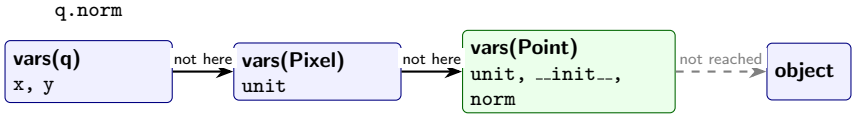
- The class body runs once. `unit` and `norm` are **class attributes**.
- Calling the class creates an **instance**. `__init__` fills its dictionary, `vars(p)`.
- A **method** receives the instance as `self`.

The class body and two dictionaries

```
>>> class Particle:
...     print("defining Particle")
...     species = "photon"
defining Particle
>>> p, p2 = Particle(), Particle()
>>> p is p2, vars(p), "species" in vars(Particle)
(False, {}, True)
>>> p.species = "electron"
>>> vars(p), Particle.species, p2.species
({'species': 'electron'}, 'photon', 'photon')
```

- The body runs at the `class` statement, before any instance exists.
- Each call of the class makes a new instance, with an empty dictionary.
- `species` now names two attributes: one in `vars(p)`, one in `vars(Particle)`.

Attribute lookup and the MRO




```
>>> class Pixel(Point):
...     unit = "px"
>>> q = Pixel(3, 4)
>>> [c.__name__ for c in Pixel.__mro__]
['Pixel', 'Point', 'object']
>>> q.unit, q.norm(), "norm" in vars(Pixel)
('px', 5.0, False)
```

- `Pixel` is a **subclass** of its **base class** `Point`.
- `q.norm` searches `vars(q)` first. Then it searches each class of `Pixel.__mro__`, the **method resolution order**.
- A function found in a class comes back as a **bound method**. `q.norm()` is `Point.norm(q)`.

Puzzle: a name from the class body

```
>>> class Label:
...     unit = "cm"
...     def text(self):
...         return unit
>>> Label().text()
```

Puzzle: a name from the class body

```
>>> class Label:
...     unit = "cm"
...     def text(self):
...         return unit
>>> Label().text()
```

Printed

```
NameError:  name 'unit' is not defined
```

- A bare name searches the local, enclosing, global and built-in scopes. The class body is not an enclosing scope of its methods.
- The method reads the class attribute as `self.unit`.

Puzzle: a shared class attribute

```
>>> class Tally:
...     seen = []
...     def add(self, x):
...         self.seen.append(x)
>>> a, b = Tally(), Tally()
>>> a.add(1)
>>> b.seen, "seen" in vars(a), a.seen is Tally.seen
```

Puzzle: a shared class attribute

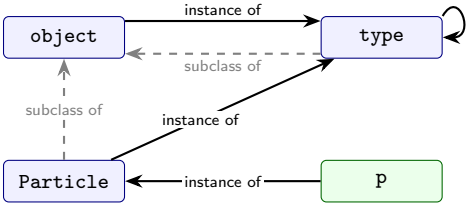
```
>>> class Tally:
...     seen = []
...     def add(self, x):
...         self.seen.append(x)
>>> a, b = Tally(), Tally()
>>> a.add(1)
>>> b.seen, "seen" in vars(a), a.seen is Tally.seen
```

Printed

```
([1], False, True)
```

- `self.seen` in `add` is a read. It finds no entry in `vars(a)` and reaches the one list of the class.
- `append` changes that list in place, so `b` sees it. State of one instance is created in `__init__`: `self.seen = []`.

Classes as objects




```
>>> type(Particle), Particle.__bases__, type(type)
(<class 'type'>, (<class 'object'>,), <class 'type'>)
```

- The type of a class is its **metaclass**: `type`, for every plain class.
- `object` ends every MRO. Calling a class runs `type.__call__`, which creates the instance and runs `__init__`.

Special methods

```
>>> class Vector:
...     def __init__(self, x, y): self.x, self.y = x, y
...     def __repr__(self): return f"Vector({self.x}, {self.y})"
...     def __len__(self): return 2
...     def __getitem__(self, i): return (self.x, self.y)[i]
>>> v = Vector(3, 4)
>>> v, len(v), v[1], list(v)
(Vector(3, 4), 2, 4, [3, 4])
```

- `len(v)` is `type(v).__len__(v)`. Iteration calls `__getitem__` with 0, 1, 2, ... until `IndexError`. Without `__len__`, `len` raises `TypeError`.

`len`, indexing, `repr`, `==` and a call run **special methods**. Their names start and end with two underscores.

Objects that can be called

```
>>> class Scale:
...     def __init__(self, k): self.k = k
...     def __call__(self, x): return self.k * x
>>> double = Scale(2)
>>> double(21), callable(double), list(map(double, [1, 2]))
(42, True, [2, 4])
```

- `double(21)` runs `type(double).__call__(double, 21)`.
- `map` accepts the instance as it accepts a function.
- A **callable** object can replace a closure. It keeps its values in attributes, such as `double.k`.

Puzzle: a callable class, its instance

```
>>> class Vector:
...     def norm(self): return 5
>>> v = Vector()
>>> hasattr(Vector, "__call__"), hasattr(v, "__call__"), callable(v)
```

Puzzle: a callable class, its instance

```
>>> class Vector:
...     def norm(self): return 5
>>> v = Vector()
>>> hasattr(Vector, "__call__"), hasattr(v, "__call__"), callable(v)
```

Printed

```
(True, False, False)
```

- `Vector` is an instance of `type`, and `vars(type)` holds `__call__`: the class is callable.
- `v.__call__` searches `vars(v)` and `Vector.__mro__`, which does not contain `type`.

Equality with a tolerance

```
>>> class Approx:
...     def __init__(self, v): self.v = v
...     def __eq__(self, other):
...         if not isinstance(other, Approx):
...             return NotImplemented
...         return abs(self.v - other.v) <= 1e-9
>>> Approx(1.0) == Approx(1.0 + 1e-12), Approx(1.0) == 1.0
(True, False)
>>> Approx.__hash__ is None
True
```

- `NotImplemented` lets Python try the other operand, then identity. A class with `__eq__` and no `__hash__` is unhashable: no dictionary key.
- `a == b` on arrays is elementwise, so `if a == b:` raises `ValueError`. Compare with `np.allclose`.

Special methods in the course project

```
>>> tr = RK4(0.05).run(lv, [1.5, 8.0], 0.0, 40.0)
>>> tr, len(tr)
(Trajectory(801 points, t in [0, 40]), 801)
>>> tr[1]
(0.05, array([1.51157577, 8.1573318 ]))
[...]
>>> lv(0.0, np.array([1.5, 8.0]))
array([0.24, 3.08])
>>> t, y = integrate(rk4_step, lv, np.array([1.5, 8.0]), 0.0, 40.0, 0.05)
>>> np.array_equal(y, tr.y), np.array_equal(t, tr.t)
(True, True)
```

- `trajectory.py`: a `Trajectory` holds the times and states of one run. It defines `__len__`, `__getitem__`, `__repr__` and `__eq__`.
- `models.py`: the `__call__` of a model calls its `rhs`. So a loop written for a function accepts the model.

Shallow and deep copies

```
>>> class Points:
...     def __init__(self, x, y): self.x, self.y = x, y
...     def moved(self, dx):
...         new = copy.copy(self)
...         new.x = self.x + dx            # a new array; y is shared
...         return new
>>> p = Points(np.zeros(3), np.zeros(3))
>>> q = p.moved(1.0)
>>> q.x is p.x, q.y is p.y, copy.deepcopy(p).y is p.y
(False, True, False)
>>> q.y[0] = 5.0; p.y
array([5., 0., 0.])
```

- `copy.copy` copies the instance dictionary. The keys are copied. The values are shared.
- `moved` rebinds `x`. So `y` is still the array of `p`.
- `copy.deepcopy` copies every object reachable from the original. Nothing is shared.

What each operation creates

operation	new object	new memory	`.base`	`np.shares_memory`
`copy.copy(obj).a`	no, the same `a`	no	unchanged	`True`
`a[::-1]`, `a[::k]`	yes	no	`a`	`True`
`a[1:][::2]`	yes	no	`a`, not the inner view	`True`
`a[mask]`	yes	yes	`None`	`False`
`a + dx`	yes	yes	`None`	`False`
`a.copy()`, `np.copy(a)`	yes	yes	`None`	`False`
`copy.deepcopy(obj).a`	yes	yes	`None`	`False`

- A basic slice is a view. A write through the view lands in the original array.
- A boolean mask makes a copy.
- `.base` names the array that owns the memory. `np.shares_memory` tests sharing directly.

Puzzle: a view inside a copy

```
>>> p = Points(np.arange(4.0), np.zeros(4))
>>> s = copy.copy(p)
>>> s.x = p.x[::2]                          # a view
>>> s.x[1] = -1.0; p.x
```

Puzzle: a view inside a copy

```
>>> p = Points(np.arange(4.0), np.zeros(4))
>>> s = copy.copy(p)
>>> s.x = p.x[::2]                          # a view
>>> s.x[1] = -1.0; p.x
```

Printed

```
array([ 0., 1., -1., 3.])
```

- `s.x` is a view of `p.x` with step 2.
- Element 1 of `s.x` is element 2 of `p.x`.
- A boolean mask, `p.x[p.x > 0]`, makes a copy. A write through it leaves `p.x` unchanged.

Abstract base classes and Protocols

```
>>> class Shape(ABC):
...     @abstractmethod
...     def area(self): ...
>>> class Circle(Shape):                  # area not defined
...     def __init__(self, r): self.r = r
>>> Circle(1.0)
Traceback (most recent call last):
  ...
TypeError: Can't instantiate abstract class Circle without an implementation for abstract method '
    area'
```

- The **interface** of a class is the set of methods its callers rely on.
- An **abstract method** is marked `@abstractmethod`. A subclass that leaves it undefined cannot be instantiated.
- The `Protocol` `HasArea` accepts any object with a method `area`. It checks the name only: `area = 3` passes.

Dataclasses

```
>>> @dataclass(frozen=True)
... class Point:
...     x: float
...     y: float = 0.0
>>> p = Point(1.0)
>>> p, p == Point(1.0, 0.0), hash(p) == hash(Point(1.0))
(Point(x=1.0, y=0.0), True, True)
>>> p.x = 2.0
Traceback (most recent call last):
  ...
dataclasses.FrozenInstanceError: cannot assign to field 'x'
```

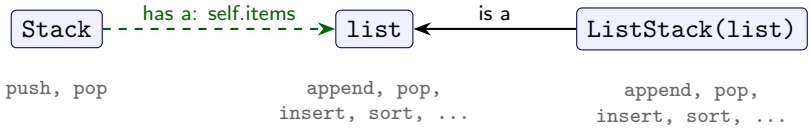
- The annotated names are the **fields**. The decorator writes `__init__`, `__repr__` and `__eq__` from them.
- `frozen=True` refuses assignment. It also adds `__hash__`.
- `field(default_factory=list)` gives each instance its own list.

Composition over inheritance

```
>>> class Stack:
...     def __init__(self): self.items = []          # has a list
...     def push(self, x): self.items.append(x)
...     def pop(self): return self.items.pop()
>>> s = Stack(); s.push(1); s.push(2)
>>> s.pop(), hasattr(s, "insert"), hasattr(list, "insert")
(2, False, True)
```

- **Inheritance** says "is a": the subclass inherits every attribute of the base.
- **Composition** says "has a": an object holds another and forwards calls to it.

Has a and is a



- A stack returns the last element it received first.
- `Stack` offers only `push` and `pop`.
- A subclass of `list` also inherits `insert` and `sort`, which break that order.

One interface, several classes

```
>>> def total_area(shapes):
...     return sum(s.area() for s in shapes)
>>> class Rectangle:
...     def __init__(self, w, h): self.w, self.h = w, h
...     def area(self): return self.w * self.h
>>> total_area([Square(2), Rectangle(2, 3)])
10
>>> [len(x) for x in ("abc", [1, 2], {"k": 0})]
[3, 2, 1]
```

- Code that calls a method by name works with any class that defines the method. This is **polymorphism**.
- `len` serves strings, lists and dictionaries. Each type has its own `__len__`.
- `total_area` works for any class written later that defines `area`.

A function that calls only the methods of an interface works with every class that defines these methods.

Summary

- `obj.name` searches the instance dictionary, then the classes in `__mro__` order. An assignment writes the instance dictionary. A class is itself an instance of `type`.
- `len`, indexing, `repr`, `==` and a call run the special methods of the class.
- A shallow copy shares every attribute that is not rebound. A deep copy shares nothing.
- An abstract base class or a Protocol declares an interface. `total_area` works with every class that defines `area`.