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

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

Contents

1 Objects, classes and attribute lookup **2**
- 1.1 Classes and instances . . . 2
- 1.2 Attribute lookup and the method resolution order . . . 3
- 1.3 Class and instance attributes . . . 4
- 1.4 Classes as objects . . . 5

2 Special methods **6**
- 2.1 Representation, length and indexing . . . 6
- 2.2 Objects that can be called . . . 7
- 2.3 Equality with a tolerance . . . 8

3 Copies of objects holding arrays **9**
- 3.1 Shallow and deep copies . . . 9
- 3.2 Views inside copies . . . 10

4 Interfaces, dataclasses and composition **11**
- 4.1 Abstract base classes and Protocols . . . 11
- 4.2 Dataclasses . . . 13
- 4.3 Composition over inheritance . . . 13
- 4.4 One interface, several classes . . . 14

Troubleshooting **15**

Key points **16**

Exercises **17**

Solutions to the exercises **19**

References **21**

A class is a type defined in Python code. Each object of a class holds its own data. The class holds the functions that act on that data.

Section 1 introduces classes and instances. Section 1 also shows the order in which Python searches for an attribute. Section 2 shows which methods of a class Python calls for `len`, indexing, `repr`, `==` and a call.

Section 3 treats copies of objects that hold arrays. Section 4 treats interfaces, dataclasses and composition. An interface lets one function work with the objects of several classes.

The listings are at the tag `lecture-11` of the course repository. Each session typed at the prompt is a file of `sessions/lecture11/`. The command `python -m doctest` replays the session files. The outputs were produced with Python 3.13 and NumPy 2.

1 Objects, classes and attribute lookup

A computation often uses several related values, such as its parameters and their names. Without an object, each value must be passed to every function that uses the value. One object can hold all these values instead. A class defines the type of such an object. The class also defines the functions that act on the data of the object.

1.1 Classes and instances

Definition 1.1: Class, instance and attribute

A **class** is a type created by a `class` statement. The body of the statement runs once. The names that the body binds are the **class attributes**. Class attributes can be functions or other values. Python stores the class attributes in the dictionary of the class.

An **instance** is an object of the class. Calling the class creates an instance. Each instance has its own dictionary, `vars(obj)`. The dictionary of an instance holds the **instance attributes**.

Right after Python creates an instance, Python calls the method `__init__` on the new instance. `__init__` binds the instance attributes.

A **method** is a function defined in the class body. The first parameter of a method is named `self` by convention. The parameter `self` receives the instance on which the method is called.

In Listing 1, `unit` is a class attribute. The call `Point(3, 4)` creates an instance. Then `__init__` binds the instance attributes `x` and `y`. The call `p.norm()` passes `p` as `self`.

```
>>> 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)
```

Listing 1: A class with a class attribute, `__init__` and a method, and one of its instances: `sessions/lecture11/plain_classes.txt`, lines 4–12, at `lecture-11`.

The built-in types are classes too. `int` is a class and `3` an instance of it. `int("12")` creates another instance of `int`. A module such as `math` has attributes, but it is not a class. For example, `isinstance(math, type)` is `False` (`sessions/lecture11/plain_classes.txt`, lines 14–18, at `lecture-11`).

The class body runs when the `class` statement runs, before any instance exists. In Listing 2, the call of `print` in the body shows when the body runs. The class dictionary holds `species` and `describe`.

Each call of the class creates a new instance. So `p` and `p2` are two different objects. The instances `p` and `p2` both start with an empty instance dictionary.

The assignment `p.species = "electron"` adds an entry to `vars(p)`. The class `Particle` keeps its own value of `species`. So `p2.species` is still `"photon"`. After the assignment, both `vars(p)` and `vars(Particle)` hold an entry `species`.

```
>>> class Particle:
...     print("defining Particle")
...     species = "photon"
...     def describe(self):
...         return self.species
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')
```

Listing 2: The class body at the `class` statement, two instances, and an instance attribute next to a class attribute of the same name: `sessions/lecture11/plain_types.txt`, lines 4–15, at `lecture-11`.

The operator `is` tests identity. The operator == tests value. CPython keeps a single object for each integer from −5 to 256. So `int("256") is int("256")` is `True`. But `int("257") is int("257")` is `False` (`sessions/lecture11/plain_types.txt`, lines 31–32, at `lecture-11`). Numbers must be compared with ==.

1.2 Attribute lookup and the method resolution order

A `class` statement may name other classes in parentheses. The classes named in parentheses are the bases of the new class. In Listing 3, `class Pixel(Point)` makes `Pixel` a **subclass** of the **base class** `Point`. `Pixel` inherits every attribute of `Point`.

`Pixel` can also define an attribute with the same name as an attribute of `Point`. The attribute of `Pixel` then overrides the attribute of `Point`.

Definition 1.2: Attribute lookup and the method resolution order

The **method resolution order** (MRO) of a class is a tuple of classes. The tuple starts with the class itself. The base classes follow. The tuple ends with `object`. For an object `obj`, the MRO is `type(obj).__mro__`. The expression `obj.name` is evaluated in three steps.

1. Python searches the instance dictionary `vars(obj)`.
2. Python then searches the dictionary of each class of the MRO, in order.
3. Python returns the value from the first dictionary that holds `name`.

Python returns a function found in a class as a **bound method**. A bound method is a callable that holds `obj`. A call of the bound method passes `obj` as the first argument. For example, `q.norm.__self__ is q` is `True`.

Figure 1 draws the search for `q.norm`. The name `norm` is not in `vars(q)` and not in `vars(Pixel)`. Python finds `norm` in `vars(Point)`. The search for `q.unit` stops earlier, at `vars(Pixel)`. The dictionary `vars(Pixel)` holds the value `"px"`.

Read on the class, `Point.norm` is the plain function of the class body. Read on the instance, `q.norm` is a bound method. The attribute `q.norm.__func__` is the function `Point.norm`. So the call `q.norm()` is the call `Point.norm(q)`. The call `Point.norm(q)` passes the instance by hand (Listing 3).

A bound method already supplies the instance. So `p.describe(p)` passes the instance twice. Python raises `TypeError: Particle.describe() takes 1 positional argument but 2 were given` (`sessions/lecture11/plain_types.txt`, lines 17–21, at `lecture-11`).

```
>>> 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)
>>> type(Point.norm).__name__, type(q.norm).__name__
('function', 'method')
>>> q.norm.__func__ is Point.norm, Point.norm(q)
(True, 5.0)
```

Listing 3: A subclass, its MRO, an attribute found one class up, and a function against a bound method: `sessions/lecture11/plain_classes.txt`, lines 20–30, at `lecture-11`.

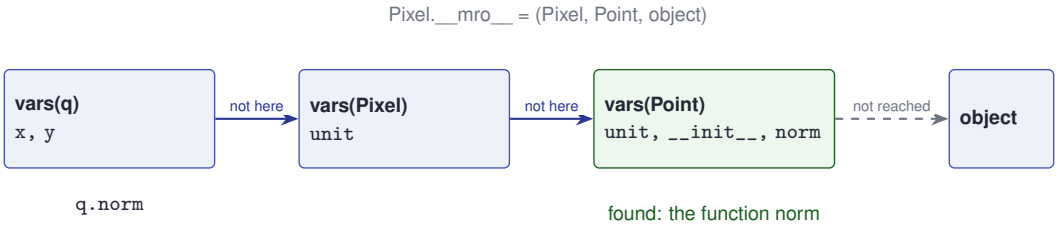


Figure 1: The lookup of `q.norm` for an instance `q` of `Pixel`. Python searches the instance dictionary first. Python then searches the classes in the order of `__mro__`. The search stops at the first dictionary that holds the name.

The built-in types follow the same search. `bool` is a subclass of `int`. `bool` defines no `__add__`. So for the sum `True + True`, Python calls the `__add__` of `int`. The result is 2 (`sessions/lecture11/plain_classes.txt`, lines 39–42, at `lecture-11`).

An assignment `obj.name = value` does not search. The assignment writes to the instance dictionary. For example, `q.unit = "mm"` adds `unit` to `vars(q)`. The read `q.unit` now finds `"mm"` first. `Pixel.unit` is still `"px"`. The class value is **shadowed** for the instance `q`.

`del q.unit` removes the instance entry. The read `q.unit` then finds the class value again (`sessions/lecture11/plain_classes.txt`, lines 32–37, at `lecture-11`).

> Reading `obj.name` searches the instance, the class and the bases. Assigning `obj.name = value` writes to the instance dictionary only. To change a class attribute for every instance, assign to the class: `Class.name = value`.

Attribute lookup is different from the lookup of a bare name. A bare name is searched in the local, enclosing, global and built-in scopes. The class body is not among the scopes that Python searches for a bare name in a method.

> Suppose a method reads a bare name bound in the class body, as in `return unit`. Python raises `NameError: name 'unit' is not defined` when the method is called (`sessions/lecture11/plain_types.txt`, lines 23–29, at `lecture-11`). The method must write `self.unit` or `Label.unit`.

1.3 Class and instance attributes

The class body runs once. So a value created in the class body exists only once. For an immutable value, a single shared object is useful. The string `"cm"` of `Point` is a default that every instance shares.

For a list, a single shared object is a trap. In Listing 4, the call `a.add(1)` runs in two steps.

1. `self.seen` is a read. Python finds no `seen` in `vars(a)`. So Python finds the list in `vars(Tally)`.
2. `append` changes the list of the class in place.

`b.seen` reads the same list. The list now holds 1.

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

Listing 4: A list in the class body, one object for every instance: `sessions/lecture11/plain_classes.txt`, lines 44–51, at `lecture-11`.

Each run of an analysis should hold its own data. Data shared by accident between objects mixes the results of different runs.

Data that belongs to one instance is created in `__init__`, as in `self.seen = []`. A dataclass field with a `default_factory` also creates one object per instance (Section 4.2).

> A list created in a class body is one object for all the instances. A dictionary or an array created in a class body is also shared. A default argument of a function is shared in the same way.
>
> Python evaluates a default argument once, when the `def` statement runs. For example, let `f` append its argument `x` to the default `acc=[]` and return `acc`. After `f(1)`, the call `f(2)` returns `[1, 2]`.

1.4 Classes as objects

A class is an object. Like every object, it has a type. Two relations therefore connect classes.

The first relation is `type(x)`. `type(x)` returns the class of which `x` is an instance. The second relation is `X.__bases__`. The tuple `X.__bases__` holds the classes from which the class `X` directly inherits.

Definition 1.3: Metaclass, `type` and `object`

The type of a class is its **metaclass**. The built-in class `type` is the metaclass of most classes. `type` is the type of `int` and of `object`. `type` is also the type of every class created by a plain `class` statement. `type` is even its own type.

The built-in class `object` is the last class of every MRO. `type` is a subclass of `object`. `object` is an instance of `type`.

Listing 5 shows both relations for `Particle`, `int`, `object` and `type`. Figure 2 draws the two relations.

A call of a class, such as `Particle()`, calls the method `__call__` of `type`. The method `type.__call__` creates the instance. `type.__call__` then calls `__init__` on the new instance.

The name `map` looks like a function name. But `map` is a class. The call `map(abs, [])` creates an instance of `map`.

```
>>> type(Particle), type(int), Particle.__bases__
(<class 'type'>, <class 'type'>, (<class 'object'>,))
>>> type(object), type(type), type.__mro__
(<class 'type'>, <class 'type'>, (<class 'type'>, <class 'object'>))
>>> isinstance(map, type), type(map(abs, []))
(True, <class 'map'>)
```

Listing 5: The type and the bases of a class, and the classes `type` and `object`: `sessions/lecture11/plain_types.txt`, lines 34–39, at `lecture-11`.

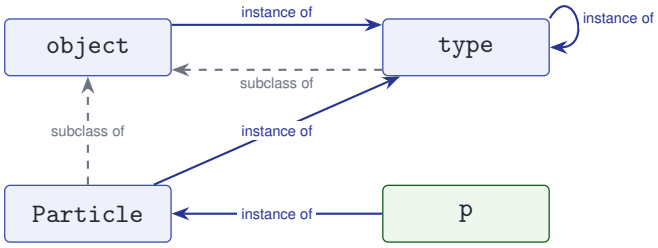


Figure 2: The two relations between the objects of Listing 5. A solid arrow leads from an object to its type, `type(x)`. A dashed arrow leads from a class to its base, `X.__bases__`.

2 Special methods

Built-in functions and operators should also work on the objects of new classes. Examples are `len`, indexing, `print`, `==` and a call. A class that defines nothing for these operations gets a default behaviour. For example, `len(obj)` raises `TypeError`. The operator `==` compares identities.

2.1 Representation, length and indexing

Definition 2.1: Special method

A **special method** is a method whose name begins and ends with two underscores. The interpreter calls a special method when an operator or a built-in function is applied to an instance. For example, `len(obj)` calls `type(obj).__len__(obj)`.

The interpreter looks up a special method on the type of the object. The lookup skips the instance dictionary.

Figure 3 shows the five special methods of this section. The `Vector` of Listing 6 defines `__repr__`, `__len__` and `__getitem__`. Iteration needs no method of its own. Python calls `__getitem__` with 0, 1, 2, ... until `__getitem__` raises `IndexError`. This is why `list(v)` works.

```
>>> 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])
```

Listing 6: A class under `repr`, `len`, indexing and iteration: `sessions/lecture11/plain_special.txt`, lines 4–11, at `lecture-11`.

A result is useful only if the setup that produced it is recorded. Objects that print their parameters turn logs and notebooks into a record of each run.

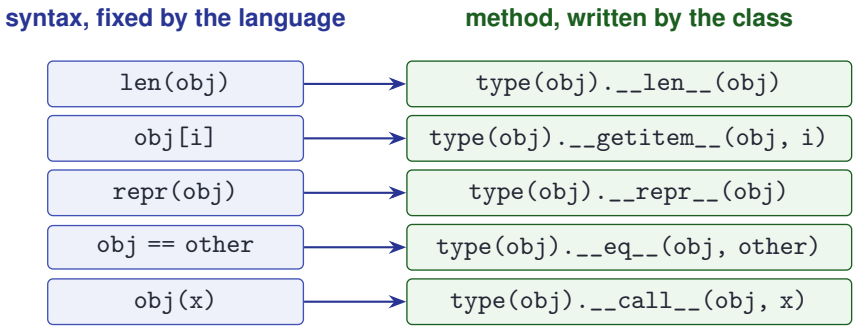


Figure 3: The five special methods of this section. For each piece of syntax on the left, the interpreter calls the method on the right. The interpreter looks up the method on the class of the object.

The lookup on the type skips the instance dictionary. For example, after `v.__len__ = lambda: 5`, `len(v)` is still 2. The direct call `v.__len__()` returns 5.

The built-in function `len` works in the same way on a built-in type. For example, `len("abc")` calls `type("abc").__len__("abc")`. Now take a class that defines no `__len__`. On the instances of the class, `len` raises `TypeError: object of type 'Bare' has no len()` (`sessions/lecture11/plain_special.txt`, lines 12–19, at `lecture-11`).

2.2 Objects that can be called

Suppose a class defines `__call__`. The instances of the class can then be called like functions. An object that can be called is **callable**. Functions, classes and bound methods are callable too. The built-in `callable(obj)` tests whether `obj` is callable. For example, `callable(Point)` is `True`, and `callable(3)` is `False`.

In Listing 7, `double` stores its factor in the attribute `k`. The call `double(21)` calls `type(double).__call__(double, 21)`. `map` accepts `double` wherever it accepts a function.

```
>>> 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])
```

Listing 7: An instance called as a function: `sessions/lecture11/plain_special.txt`, lines 21–26, at `lecture-11`.

A callable object can replace a closure. A closure is a function that keeps values from the scope that created it. The callable object keeps its values in named attributes, such as `double.k`. A closure keeps its values in cells.

Special methods are looked up on the type. The lookup on the type decides which objects are callable.

Take a class `Vector` that defines no `__call__`. The class `Vector` is callable. The type of `Vector` is `type`. The dictionary `vars(type)` holds `__call__`.

The instance `v` is not callable. The search for `v.__call__` covers `vars(v)` and the classes of `Vector.__mro__`. The class `type` is not in `Vector.__mro__`. So `callable(Vector)` is `True`, and `callable(v)` is `False` (`sessions/lecture11/plain_types.txt`, lines 41–47, at `lecture-11`).

In `attractor`

In the course project, `models.py` defines models of two interacting populations as classes. The base class of the model classes is `DynamicalSystem`. `DynamicalSystem` defines `__call__` to call the method `rhs`. The method `rhs` returns the time derivative of a state (`models.py`, lines 23–24, at `lecture-11`).

`lv = LotkaVolterra()` is one model, with default rates. The call `lv(0.0, np.array([1.5, 8.0]))` calls `lv.rhs`. The call returns `array([0.24, 3.08])`.

An integration loop written for a function `rhs(t, y)` only calls `rhs`. So the loop accepts `lv` unchanged (`sessions/lecture11/special.txt`, lines 19–23, at `lecture-11`).

2.3 Equality with a tolerance

Without `__eq__`, `a == b` tests `a is b`. Two numbers computed by two routes agree only up to rounding. For example, `0.1 + 0.2 == 0.3` is `False`. A class of such values may therefore define == as equality within a tolerance.

In Listing 8, `__eq__` returns `NotImplemented` when the other operand has another type. The value `NotImplemented` tells Python to call the `__eq__` of the other operand. If the other `__eq__` also returns `NotImplemented`, Python compares identities.

```
>>> 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
```

Listing 8: Equality within a tolerance, and the hash it removes: `sessions/lecture11/plain_special.txt`, lines 28–37, at `lecture-11`.

A dictionary or a set stores an object by its hash, `hash(obj)`, an integer. Two objects that compare equal must have the same hash. For example, `1 == 1.0` holds, and `hash(1) == hash(1.0)` is `True`.

The hash inherited from `object` is based on identity. With the inherited hash, two equal `Approx` objects would have different hashes. To prevent different hashes for equal objects, Python sets `__hash__` to `None` in a class that defines `__eq__` and no `__hash__`. The instances of such a class are not **hashable**. Unhashable objects cannot be dictionary keys.

For two arrays, `a == b` compares element by element. The comparison returns an array of booleans. So `if a == b:` raises `ValueError` (`sessions/lecture11/plain_special.txt`, lines 39–46, at `lecture-11`). A method that compares arrays first checks the shapes. The method then reduces the comparison to one boolean, with `np.array_equal` or `np.allclose`.

In `attractor`

The class `Trajectory` of the course project holds the times `t` and the states `y` of one simulation run. The `__eq__` of `Trajectory` works in three steps (`trajectory.py`, lines 32–37, at `lecture-11`).

1. `__eq__` returns `NotImplemented` for an operand that is not a trajectory.
2. `__eq__` checks that the shapes agree.
3. `__eq__` calls `np.allclose` on the times and on the states.

The shape check comes first. `np.allclose` broadcasts arrays of different shapes when broadcasting is possible. For example, `np.allclose(np.zeros(3), np.zeros((2, 3)))` is `True`. `np.allclose` raises an error when broadcasting is impossible.

A run whose states are all moved by 10^{-9} compares equal. A run that keeps every second point does not compare equal. `Trajectory.__hash__` is `None` (sessions/lecture11/special.txt, lines 25–29, at lecture-11).

3 Copies of objects holding arrays

A method may return a changed version of an object and leave the original intact. For each attribute, the result either gets a new object or shares the object of the original. A write through a shared array changes both objects.

3.1 Shallow and deep copies

Definition 3.1: Shallow copy and deep copy

`copy.copy(x)` makes a **shallow copy**. A shallow copy is a new object of the same type as `x`. The instance dictionary of the shallow copy is a copy of `vars(x)`. `copy.copy` copies the keys, but not the values. So each attribute of the copy is bound to the same object as in `x`.

`copy.deepcopy(x)` makes a **deep copy**. A deep copy copies `x`. `copy.deepcopy` also copies, recursively, every object reachable from `x`.

`deepcopy` keeps a record of the objects already copied. So an object referenced twice is copied once. For example, for a list `a`, let `c = copy.deepcopy([a, a])`. Then `c[0] is c[1]` is `True`. The record of copied objects also lets `deepcopy` terminate on a cycle of references.

Take a list of lists, `grid`. The shallow copy of `grid` is a new outer list. The new outer list holds the same inner lists as `grid`.

A write into an inner list of the copy therefore changes `grid`. For example, let `grid = [[0, 0], [0, 0]]` and `sh = copy.copy(grid)`. After `sh[0][0] = 1`, `grid` is `[[1, 0], [0, 0]]`. The deep copy of `grid` has new inner lists (sessions/lecture11/plain_copies.txt, lines 6–10, at lecture-11).

A method that returns a changed object often starts with `copy.copy(self)`. The method then rebinds the attribute that changes. In Listing 9, `moved` works in two steps.

1. `moved` makes the shallow copy `new`. At this point, `new.x` and `new.y` are the arrays of `p`.
2. `moved` binds `self.x + dx`, a new array, to `new.x`.

`new.y` is still the array of `p`. So a write through `q.y` changes `p.y`. Only a deep copy of `p` owns its own `y` (Figure 4).

```
>>> 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.])
```

Listing 9: A copy with one attribute rebound, and a write through the attribute it shares: `sessions/lecture11/plain_copies.txt`, lines 12–23, at `lecture-11`.

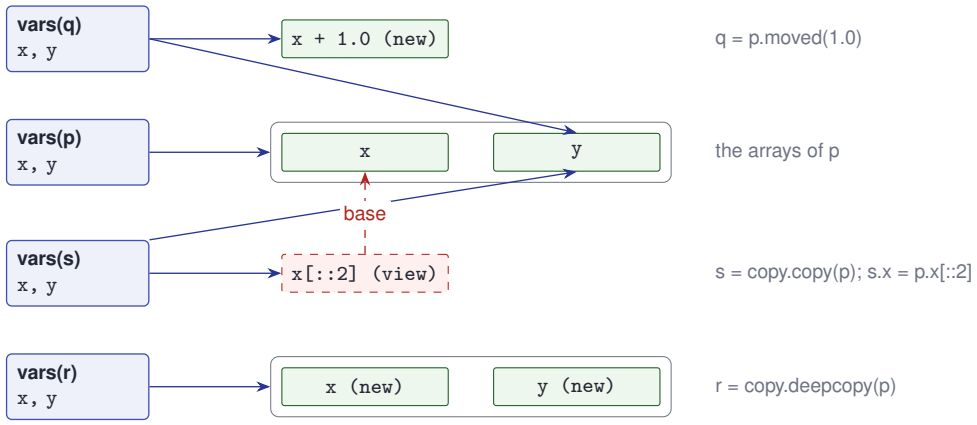


Figure 4: The objects behind three copies of `p`. The shallow copy `q` owns a new `x` and shares `y`. The shallow copy `s` also shares `y`. The `x` of `s` is a view that uses the memory of `p.x`. The arrow marked `base` leads from the view to `p.x`. The deep copy `r` owns new arrays.

> `copy.copy` copies the instance dictionary only. `copy.copy` does not copy the arrays. Suppose a method rebinds one attribute of the copy. All the other attributes of the copy are still shared with the original.
>
> Lists and arrays are shared in the same way. For example, `q.y is p.y` is `True`: both names refer to one array. A write through `q.y` therefore changes `p.y`.

A comparison of simulation runs is valid only if the runs are independent. Copies that share arrays can tie the runs together without raising an error.

3.2 Views inside copies

A basic slice of an array is written `start:stop:step`. A basic slice returns a view. A **view** is a new array object that uses the memory of the original array. An attribute of a copy that is bound to a view shares the memory of the original.

In Listing 10, `s.x` is a view of `p.x` with step 2. Element 1 of `s.x` is element 2 of `p.x`. So the write to `s.x[1]` changes `p.x[2]`.

A boolean mask behaves differently. A mask is an array of `True` and `False` that selects elements. Indexing with a mask returns a copy. The copy owns its memory. The `.base` of the copy is `None`.

```
>>> p = Points(np.arange(4.0), np.zeros(4))
>>> s = copy.copy(p)
>>> s.x = p.x[::2]                      # a view
>>> s.x.base is p.x, np.shares_memory(s.x, p.x)
(True, True)
>>> s.x[1] = -1.0; p.x
array([ 0.,  1., -1.,  3.])
>>> m = p.x[p.x > 0]                    # a boolean mask copies
>>> m.base is None, np.shares_memory(m, p.x)
(True, False)
```

Listing 10: A view bound to a copy, a write through it, and a mask that copies: `sessions/lecture11/plain_copies.txt`, lines 25–34, at `lecture-11`.

Table 1 lists what each operation creates. The attribute `.base` is the array that owns the memory. For a view of a view, `.base` is the first array.

For example, let `v = p.x[1:]`. Then `v[::2].base is p.x` is `True`. And `v[::2].base is v` is `False` (`sessions/lecture11/plain_copies.txt`, lines 35–37, at `lecture-11`). `np.shares_memory` returns the right answer for any chain of views.

Table 1: What each operation creates, for an array `a` that owns its memory and an object `obj` that holds it as an attribute. "New object" is a new Python object, "new memory" a fresh buffer for the elements.

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 method whose result must not write into the original binds a copy of the slice, as in `self.x[::2].copy()`. A view would pass every write on to the original array. For example, `p.x[::2].copy().base is None` is `True`.

4 Interfaces, dataclasses and composition

A function that works with objects of several classes uses the methods that the classes have in common. If no part of the code lists the common methods, a class that does not implement one of the common methods fails late. Python raises an error only when the function calls the missing method.

4.1 Abstract base classes and Protocols

Definition 4.1: Interface, abstract base class and Protocol

The **interface** of a class is the set of methods that other code relies on when using its instances. An **abstract base class** defines required methods using `@abstractmethod`. A subclass that does not implement all these methods cannot be instantiated. Python raises an error identifying the missing methods.

A **Protocol** is a class that lists the methods of an interface. A class satisfies a Protocol by defining all these methods. The class does not need to inherit from the Protocol. The function `isinstance` can test a Protocol decorated with `@runtime_checkable`.

In Listing 11, `Shape` inherits from `ABC` of the module `abc`. `Shape` defines `area` as an **abstract method**. An abstract method is a method that every subclass must implement. `Circle` does not implement `area`. So Python raises `TypeError` at the call `Circle(1.0)`. The error message identifies the missing method `area`.

```
>>> 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'
```

Listing 11: An abstract base class, and a subclass that does not implement the abstract method: `sessions/lecture11/plain_design.txt`, lines 7–15, at `lecture-11`.

A Protocol states the same requirement by method names. `Square` in Listing 12 inherits from no class. `Square` defines `area`. So `Square` satisfies `HasArea`.

`Square` does not inherit from `Shape`. So `isinstance(sq, Shape)` is `False`. The standard library defines its own interfaces as abstract base classes. For example, `collections.abc.Sized` is the abstract base class of the objects that `len()` accepts. `isinstance("abc", Sized)` is `True`. `isinstance(3, Sized)` is `False` (`sessions/lecture11/plain_design.txt`, lines 34–36, at `lecture-11`).

```
>>> @runtime_checkable
... class HasArea(Protocol):
...     def area(self): ...
>>> class Square:                          # no base class
...     def __init__(self, s): self.s = s
...     def area(self): return self.s ** 2
>>> sq = Square(3)
>>> isinstance(sq, HasArea), isinstance(sq, Shape), isinstance(3, HasArea)
(True, False, False)
```

Listing 12: A Protocol satisfied by a class that does not inherit from it: `sessions/lecture11/plain_design.txt`, lines 17–25, at `lecture-11`.

An abstract base class and a Protocol check at different times. An abstract base class checks when an instance is created. The subclasses of an abstract base class also inherit its other methods. A Protocol checks nothing until `isinstance` is called. `isinstance` then accepts any object that defines the methods of the Protocol.

Long computations are expensive to run again. A check of the interface when an object is created finds a missing method before the computation starts.

A runtime-checkable Protocol checks names only. Take an object whose attribute `area` is the number 3. The object passes the test `isinstance(obj, HasArea)`. But the call `obj.area()` raises `TypeError: 'int' object is not callable` (`sessions/lecture11/plain_design.txt`, lines 26–33, at `lecture-11`).

4.2 Dataclasses

Definition 4.2: Dataclass

A **dataclass** is a class for which the decorator `@dataclass` generates the methods `__init__`, `__repr__` and `__eq__`. The decorator generates these methods from the **fields**. The fields are the annotated names of the class body, in the order they are written.

With `frozen=True`, assigning to a field after `__init__` raises an error. With `frozen=True`, the decorator also generates `__hash__` from the fields. So the instances can be dictionary keys. A field whose default is a new list for each instance is written `field(default_factory=list)`.

A decorator receives a function or a class. The decorator returns a replacement, as in the previous chapter. `@dataclass` receives the class. `@dataclass` adds methods to the class and returns the same class.

In Listing 13, the annotated names `x` and `y` are the fields. `y` has a default value. The generated `__repr__` shows the fields. The generated `__eq__` compares the fields in order.

With `frozen=True`, the assignment `p.x = 2.0` raises `FrozenInstanceError`. `frozen=True` also adds `__hash__`. The function `dataclasses.replace` returns a changed copy. For example, `replace(p, x=2.0)` returns `Point(x=2.0, y=0.0)` (sessions/lecture11/plain_design.txt, lines 49–50, at lecture-11).

```
>>> @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'
```

Listing 13: The methods that `@dataclass(frozen=True)` generates: sessions/lecture11/plain_design.txt, lines 38–48, at lecture-11.

Without the decorator, a class with the same `__init__` compares by identity. So `PlainPoint(1.0) == PlainPoint(1.0)` is `False`.

The option `default_factory=list` of `field` defines a field whose default is a new list. The generated `__init__` calls `list()` for each new instance. So the test `Path().points is Path().points` is `False` (sessions/lecture11/plain_design.txt, lines 51–59, at lecture-11). For a bare list as a default, the decorator raises `ValueError` when it builds the class.

4.3 Composition over inheritance

Inheritance is the subclass relation of Section 1.2. Inheritance expresses "is a". Code that accepts instances of the base class also accepts instances of the subclass. For example, `isinstance(Pixel(3, 4), Point)` is `True`.

Composition expresses "has a". In composition, an object holds another object as an attribute. The outer object forwards calls to the inner object.

A subclass inherits every attribute of its base, wanted or not. Consider a stack. A stack returns first the last element it received. The order of a stack is called last in, first out.

The `Stack` of Listing 14 holds a list. `Stack` defines only `push` and `pop`. A subclass of `list` would also inherit `insert` and `sort`. The methods `insert` and `sort` break the order of a stack (Figure 5).

```
>>> 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)
```

Listing 14: A stack that has a list: `sessions/lecture11/plain_design.txt`, lines 61–67, at `lecture-11`.

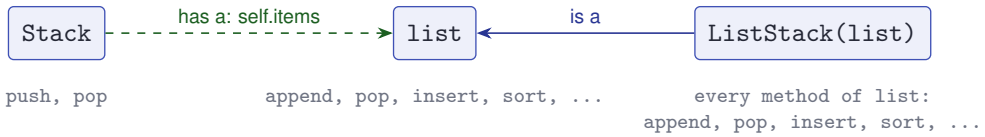


Figure 5: "Has a" and "is a". `Stack` holds a list. `Stack` defines the two methods of a stack. A subclass of `list` inherits every method of `list`, including `insert` and `sort`.

4.4 One interface, several classes

Code that calls a method by name works with any object whose class defines that method. **Polymorphism** is the use of one piece of code with the objects of several classes that define the same methods.

The built-in `len` is an example of polymorphism. The call `len(x)` works for strings, lists and dictionaries. Each of these types defines its own `__len__`. For example, `[len(x) for x in ("abc", [1, 2], {"k": 0})]` is `[3, 2, 1]` (`sessions/lecture11/plain_design.txt`, lines 76–77, at `lecture-11`).

In Listing 15, `total_area` calls `area()` on each element. `total_area` returns the sum of the areas of squares and rectangles. `total_area` also accepts the instances of any class written later that defines `area`.

```
>>> 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
```

Listing 15: One function over two classes that share a method: `sessions/lecture11/plain_design.txt`, lines 69–75, at `lecture-11`.

Troubleshooting

symptom	cause	remedy
`TypeError: Can't instantiate abstract class ...`	A subclass does not implement a method decorated with `@abstractmethod` (Definition 4.1).	Implement the method that the error message identifies.
`TypeError: unhashable type` when an object is a dictionary key	`__eq__` without `__hash__`, or a dataclass with `eq=True` and no `frozen` (Sections 2.3 and 4.2).	Declare the class `@dataclass(frozen=True)`.
`ValueError: mutable default <class 'list'> ...`	A bare list as the default of a dataclass field (Section 4.2).	Pass `default_factory=list` to `field`.
`ModuleNotFoundError: No module named ...` from a script in a subfolder of the project	The script was run as a file. Python then puts the folder of the script first on the import path. The project root is not on the path.	Run it from the project root as `python -m folder.name`.

Table 2: Objects and introspection of this chapter.

expression	effect	section
`vars(obj)`, `type(obj).__mro__`	the instance dictionary, and the classes that attribute lookup searches after it	1.1, 1.2
`type(x)`, `X.__bases__`	the class of an object, and the direct base classes of a class	1.4
`__repr__`, `__len__`, `__getitem__`, `__call__`, `__eq__`	what `repr`, `len`, indexing and iteration, a call and == call	2.1–2.3
`copy.copy`, `copy.deepcopy`	a new object that shares every attribute, and a recursive copy	3.1
`np.shares_memory(a, b)`, `a.base`	shared memory whatever the chain of views, and the owner of a view's memory	3.2
`ABC`, `@abstractmethod`, `Protocol`	an interface checked at instantiation, and an interface checked by method names	4.1
`@dataclass(frozen=True)`, `field`, `replace`	generated methods and immutability, a fresh default per instance, a changed copy	4.2

Key points

- Reading `obj.name` searches the instance, then the classes of `__mro__`. Assigning `obj.name = v` writes to the instance dictionary. A mutable value in a class body is shared by all the instances (Definitions 1.1 and 1.2). A class is itself an instance of `type` (Definition 1.3).
- `len`, indexing, `repr`, `==` and a call run special methods. Python looks up special methods on the class (Figure 3). A class with `__eq__` and no `__hash__` is unhashable.
- A shallow copy shares every attribute with the original. A write through a shared attribute changes the original. A write through a view bound to the copy also changes the original. A deep copy shares nothing (Definition 3.1, Table 1).
- A subclass that does not implement all the abstract methods of its base cannot be instantiated. A Protocol accepts any object with the right method names. For a frozen dataclass, `@dataclass` generates `__init__`, `__repr__`, `__eq__` and `__hash__` from the fields (Definitions 4.1 and 4.2).
- A function that calls a method by name works with every class that defines the method. For example, `total_area` adds the areas of squares and rectangles (Listing 15).

Exercises

Solutions are given from page 19. The scripts that check them are in `exercises/lecture11/` of the project.

Four bodies for one method

A class `Field` holds a grid `x` and a field `phi`, the `float64` arrays `np.arange(5.0)` and `np.arange(5.0) * 10`. The method `transformed` of `Field` works in three steps. `transformed` starts with `new = copy.copy(self)`. `transformed` then runs one of the four bodies below. Last, `transformed` returns `new`.

body	
(a)	`new.phi = self.phi`
(b)	`new.phi = self.phi[::-1]`
(c)	`new.phi = self.phi[self.x > 1]`
(d)	`new.phi = self.phi.copy()[::-1]`

Let `F = Field(...)` and `G = F.transformed()`. For each body, give three answers: the value of `np.shares_memory(G.phi, F.phi)`, the value of `G.phi.base is F.phi`, and whether `G.phi[0] = -1.0` changes `F.phi`. Then explain why (a) and (b) differ in the second answer but agree in the third.

Lookup order

Give the two lines printed by the program of Listing 16.

```
class Grid:
    step = 0.1
a, b = Grid(), Grid()
a.step = 0.05
print(a.step, b.step, "step" in vars(a), "step" in vars(b))
del a.step
print(a.step, b.step, "step" in vars(a))
```

Listing 16: A class attribute shadowed and uncovered: `exercises/lecture11/e2_lookup_order.py`, lines 1–7, at `lecture-11`.

A dataclass default

Give the two lines printed by the program of Listing 17. A third class `Bad` has the field `entries: list = []`. Give the exception, with its message, that the `@dataclass` decorator raises for this class.

```
@dataclass
class Log:
    entries: list = field(default_factory=list)
class PlainLog:
    def __init__(self, entries=[]):
        self.entries = entries
r1, r2 = Log(), Log()
r1.entries.append("start")
print(r1.entries, r2.entries, r1.entries is r2.entries)
p1, p2 = PlainLog(), PlainLog()
p1.entries.append("start")
print(p1.entries, p2.entries, p1.entries is p2.entries)
```

Listing 17: A list default, generated and written by hand:
`exercises/lecture11/e3_dataclass_default.py`, lines 2–13, at `lecture-11`.

Cost of the interface

The model `lv = LotkaVolterra()` of Section 2.2 can be evaluated by three routes.

1. A closure `rhs(t, y)` that holds the same four rates.
2. The bound method `lv.rhs`.
3. The instance `lv` itself, called through `__call__`.

Time 10^5 evaluations by each route, at the time 0 and the state $(1.5, 8.0)$. Report the three times, and the ratio of the last to the first. Does the difference between the routes exceed the variation between repeated runs?

A tolerant equality that is not an equivalence

`np.allclose(a, b)` holds when every element satisfies $|a_i - b_i| \le 10^{-8} + 10^{-5}\,|b_i|$. Construct three trajectories `A`, `B` and `C` on three times. In each trajectory, all the states are equal to one constant. `A == B` and `B == C` must hold, and `A == C` must not.

Give the three constants and the value of `Trajectory.__hash__` (Section 2.3). Explain why `__hash__` is `None` for a class with such an `__eq__`.

Solutions to the exercises

Four bodies for one method

body	`shares_memory`	`G.phi.base is F.phi`	`F.phi` changed	reason
(a)	`True`	`False`	yes	the same array object, no view
(b)	`True`	`True`	yes	a reversed view of `F.phi`
(c)	`False`	`False`	no	a boolean mask copies
(d)	`False`	`False`	no	a view of the copy, whose base is the copy

In (a), `G.phi is F.phi`. An array that owns its memory has `.base` equal to `None`. So the second test is `False`. `G.phi` and `F.phi` still share every element.

In (b), `G.phi` is a new object that uses the memory of `F.phi`. In both cases, `G.phi[0] = -1.0` writes into the memory of `F.phi`. The write changes index 0 of `F.phi` in (a), and index 4 in (b).

Answer check

```
$ python exercises/lecture11/e1_four_bodies.py
(a) self.phi                shares True   base is False  F.phi changed True
(b) self.phi[::-1]          shares True   base is True   F.phi changed True
(c) self.phi[self.x > 1]    shares False  base is False  F.phi changed False
(d) self.phi.copy()[::-1]   shares False  base is False  F.phi changed False
```

Listing 18: `exercises/lecture11/e1_four_bodies.py` at `lecture-11`.

Lookup order

`0.05 0.1 True False`, then `0.1 0.1 False`. The assignment `a.step = 0.05` writes to the instance dictionary of `a`. The instance entry shadows the class value. `b` has no instance entry `step`. So `b.step` reads the class value.

`del a.step` removes the entry of `a`. The lookup of `a.step` then finds the class value again (Definition 1.2).

Answer check

```
$ python exercises/lecture11/e2_lookup_order.py
0.05 0.1 True False
0.1 0.1 False
```

Listing 19: `exercises/lecture11/e2_lookup_order.py` at `lecture-11`.

A dataclass default

`['start'] [] False`, then `['start'] ['start'] True`. The generated `__init__` calls the factory once per `Log()`. So the two lists are distinct.

Python evaluates the default `entries=[]` of `PlainLog` once, when the `def` runs. `p1` and `p2` share the default list. The third class raises `ValueError: mutable default <class 'list'> for field entries is not allowed: use default_factory.` The decorator raises the `ValueError` while it builds the class `Bad`.

Answer check

```
$ python exercises/lecture11/e3_dataclass_default.py
['start'] [] False
['start'] ['start'] True
ValueError: mutable default <class 'list'> for field entries is not allowed: use
    default_factory
```

Listing 20: `exercises/lecture11/e3_dataclass_default.py` at `lecture-11`. The script catches the exception and prints its message.

Cost of the interface

The three routes cost the same, within the variation between runs. On the lecture laptop, the ratio of the call through `__call__` to the closure was 1.08 under Python 3.13. Another run, under Python 3.12, gave 0.94.

Answer check

```
$ python -m exercises.lecture11.e4_interface_cost
closure  0.063 s
lv.rhs   0.063 s
lv()     0.067 s
lv() / closure = 1.08
```

Listing 21: `exercises/lecture11/e4_interface_cost.py` at `lecture-11`. One run under Python 3.13 on the lecture laptop.

A tolerant equality that is not an equivalence

Take the constants 1, $1 + 0.8 \times 10^{-5}$ and $1 + 1.6 \times 10^{-5}$. Each neighbouring pair is within the tolerance $10^{-8} + 10^{-5}$. The outer pair, 1 and $1 + 1.6 \times 10^{-5}$, is not within the tolerance. So the relation is not transitive. A relation that is not transitive is not an equivalence.

A hash must agree for objects that compare equal. Think of a chain of states, each within the tolerance of the next. The hash would have to take one value along the whole chain. So no useful hash exists for such an equality.

Python sets `__hash__` to `None` for every class that defines `__eq__` without `__hash__`. The reason is that the hash inherited from `object` is based on identity. The identity-based hash would differ for two equal objects.

Answer check

```
$ python -m exercises.lecture11.e5_tolerance
True True False
None
```

Listing 22: `exercises/lecture11/e5_tolerance.py` at `lecture-11`. The script prints `A == B, B == C, A == C` and then `Trajectory.__hash__`.

References

Boris Bolliet. `attractor`: cycle periods of the chemostat runs. GitHub, https://github.com/borisbolliet/attractor. tag `lecture-11`.

Python Software Foundation. Data model. Python 3.13 language reference, https://docs.python.org/3/reference/datamodel.html. accessed 24 October 2026.

NumPy developers. `numpy.allclose`. NumPy 2.5 reference, https://numpy.org/doc/stable/reference/generated/numpy.allclose.html. accessed 24 October 2026.

Python Software Foundation. `copy`: shallow and deep copy operations. Python 3.13 documentation, https://docs.python.org/3/library/copy.html. accessed 24 October 2026.

NumPy developers. `numpy.shares_memory`. NumPy 2.5 reference, https://numpy.org/doc/stable/reference/generated/numpy.shares_memory.html. accessed 24 October 2026.

NumPy developers. `numpy.ndarray.base`. NumPy 2.5 reference, https://numpy.org/doc/stable/reference/generated/numpy.ndarray.base.html. accessed 24 October 2026.

Python Software Foundation. `abc`: abstract base classes. Python 3.13 documentation, https://docs.python.org/3/library/abc.html. accessed 24 October 2026.

Python Software Foundation. `typing`: support for type hints. Python 3.13 documentation, https://docs.python.org/3/library/typing.html. accessed 24 October 2026.

Python Software Foundation. `collections.abc`: abstract base classes for containers. Python 3.13 documentation, https://docs.python.org/3/library/collections.abc.html. accessed 25 October 2026.

Python Software Foundation. `dataclasses`: data classes. Python 3.13 documentation, https://docs.python.org/3/library/dataclasses.html. accessed 24 October 2026.

Index

abstract base class, 11
abstract method, 12

base class, 3
bound method, 3

callable, 7
class, 2
class attributes, 2
composition, 13

dataclass, 13
deep copy, 9

fields, 13

hashable, 8

inheritance, 13
instance, 2
instance attributes, 2
interface, 11

metaclass, 5
method, 2
method resolution order, 3

Polymorphism, 14
Protocol, 12

shadowed, 4
shallow copy, 9
special method, 6
subclass, 3

view, 10