Lecture 9 — The Python execution model: names, objects, scope and closures

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
$ cd ~/GitHub/attractor
$ git fetch --tags
$ git switch --detach lecture-09-start
$ source setup_env.sh
$ python simulate.py
$ python -m doctest sessions/lecture09/*.txt
```

- The clone is made once.
- If `git status` lists local edits, run `git stash` first.
- `lecture-09` is the state at the end of the lecture.
- `doctest` prints nothing when each session matches the notes.

Outline

- Names and objects: binding, identity, mutability, views
- Calls and scope: defaults, `*args` and `**kwargs`, LEGB
- Functions as objects and closures: cells and late binding
- Modules and import time: the module cache `sys.modules`, values read at import
- Live demonstration: a simulated trajectory whose rows are one object, then models as closures

A Python name is a label on an object. Assignments, calls, imports and closures bind names. None of them copies an object.

The model side of the project

- `attractor` measures the cycle period of ten chemostat runs. Three files now simulate the runs:
  - `models.py`: the right-hand side `rhs(t, y)`, the derivatives of the two densities
  - `integrators.py`: the RK4 step (Runge–Kutta of order four) and the loop of steps
  - `simulate.py`: the driver, the script that runs the two and measures the period
- Lotka–Volterra on the chemostat scales, algae x in 10^6 cells/ml and rotifers z in animals/ml:

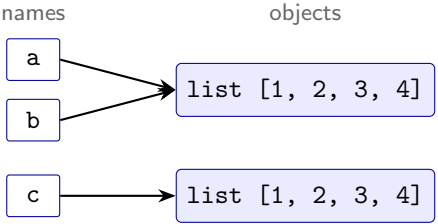
$$\frac{\mathrm{d}x}{\mathrm{d}t} = ax - bxz, \qquad \frac{\mathrm{d}z}{\mathrm{d}t} = cxz - dz,$$

with $a = 0.8$, $b = 0.08$, $c = 0.77$, $d = 0.77$ per day (`CHEMOSTAT` in `models.py`).

The first integrator returns 801 rows. Each row is the final state, $(1.2384, 6.3268)$. All rows are one array object.

`github.com/borisbolliet/attractor/blob/lecture-09-start/integrators.py`

Assignment as binding



`a = [1, 2, 3]` creates a list and binds `a`. `b = a` binds a second name to it. `c = a[:]` binds `c` to a new list.

Definition: name, object, binding

An **object** is a value in memory with a **type** and an **identity**. A **name** is an entry in a **namespace**, a dictionary from names to objects. An assignment **binds** a name to an object. It never copies the object. After `b = a`, a change through `b` is seen through `a`.

- A name has no type. The type belongs to the object.

Puzzle: aliases and copies

```
a = [1, 2, 3]
b = a
b.append(4)
print(a, a is b)
c = a[:]
print(c == a, c is a)
x = 1000; y = 1000
print(x == y, x is y)
```

Puzzle: aliases and copies

```
a = [1, 2, 3]
b = a
b.append(4)
print(a, a is b)
c = a[:]
print(c == a, c is a)
x = 1000; y = 1000
print(x == y, x is y)
```

```
$ python puzzles/p1_names.py
[1, 2, 3, 4] True
True False
True True
```

- == compares values. `is` compares identities.
- The slice makes a new list.

Identity, equality and interned objects

```
x = 256; y = 256
p = 1000; q = 1000
print(x is y, p is q)
s = "".join(["rk", "4"]); t = "rk4"
print(s == t, s is t)
```

```
$ python puzzles/interning.py
True True
True False
```

- In one file, the two literals `1000` are one constant.
- At the prompt, `p is q` is `False`. CPython shares only the integers from -5 to 256.

Use `is` for `x is None` and for alias checks. Compare values with `==`.

Mutability and aliasing

types	mutable	`x += v`
`int`, `float`, `str`, `tuple`	no	computes `x + v`, rebinds `x` to the new object
`list`, `dict`, `set`	yes	changes the object in place (`extend` for a list)
`numpy.ndarray`	yes	writes the sum into the existing buffer

- A **mutable** object can change after it is created. An operation on an **immutable** object makes a new object.
- Two names on one object are **aliases**. A change through one is seen through the other.
- A parameter is an alias of the caller's object. So the table also applies to arguments.

`v += [1]` changes the shared list. `v = v + [1]` binds `v` to a new list.

Aliasing in a loop

```
>>> state = np.zeros(2)
>>> rows = []
>>> for k in range(3):
...     state += 1            # writes into the one array
...     rows.append(state)    # appends that array again
...
>>> rows
[array([3., 3.]), array([3., 3.]), array([3., 3.])]
>>> len({id(r) for r in rows})
1
```

- A loop that stores its state needs a new object at each pass. Here the list holds one array three times.
- `state = state + 1` makes a new array at each pass. `rows.append(state.copy())` stores a copy instead.

✎ Aliasing through a returned argument

```
$ python -i simulate.py
first row  1.2384 6.3268
last row   1.2384 6.3268
801 rows, step 0.05, algae period nan days
written: results/simulated.png
>>> len({id(r) for r in rows})
1
>>> rows[0] is rows[-1]
True
>>> np.shares_memory(rows[0], rows[5])
True
```

- In `integrators.py`, the loop runs `y = rk4_step(...)`, then `rows.append(y)`.
- The step ends with `y += ...` and `return y`. It returns the caller's array. So one array is appended 800 times.

✎ Rebinding instead of writing in place

```
$ git diff -U0 lecture-09-start lecture-09
[...]
@@ -16 +16 @@ def rk4_step(rhs, t, y, h):
-    y += h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
+    y = y + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
$ python simulate.py
first row  1.5000 8.0000
last row   1.2384 6.3268
801 rows, step 0.05, algae period 8.16 days
written: results/simulated.png
```

- `y + ...` makes a new array at every step. The argument keeps its values.
- `rows.append(y.copy())` would also work.
- `integrators.py` rebinds. A step that writes into its argument forces each caller to copy.

✎ Views of an array

```
>>> a = np.arange(6.0)
>>> v = a[::2]
>>> v.base is a
True
>>> v[0] = -1
>>> a
array([-1.,  1.,  2.,  3.,  4.,  5.])
>>> c = a[1:4]
>>> c is v, np.shares_memory(c, v)
(False, True)
>>> np.shares_memory(a.copy(), a)
False
```

- A basic slice of an array is a **view**. It uses the memory of its owner, `.base`. A write through a view changes the owner.
- `np.shares_memory` tells whether two arrays overlap.
- `a.copy()` and a list slice are copies.

Puzzle: a mutable default

```
def step(y, h, log=[]):
    log.append(h)
    return y + h, log
print(step(0.0, 0.1)[1])
print(step(0.0, 0.2)[1])
```

Puzzle: a mutable default

```
def step(y, h, log=[]):
    log.append(h)
    return y + h, log
print(step(0.0, 0.1)[1])
print(step(0.0, 0.2)[1])
```

```
$ python puzzles/p2_calls.py
[0.1]
[0.1, 0.2]
[...]
```

- `def` runs once. The default is evaluated then. It is stored in `__defaults__`.
- So every call that omits `log` shares one list.
- The idiom: `log=None`, then a new list inside the function.

Puzzle: += on an argument

```
def bump(v, w):
    v += w
def regrow(v):
    v = v + [1]
u = [0]; bump(u, [1]); regrow(u); print(u)
t = (0,); bump(t, (1,)); print(t)
n = 0; bump(n, 1); print(n)
```

Puzzle: += on an argument

```
def bump(v, w):
    v += w
def regrow(v):
    v = v + [1]
u = [0]; bump(u, [1]); regrow(u); print(u)
t = (0,); bump(t, (1,)); print(t)
n = 0; bump(n, 1); print(n)
```

```
$ python puzzles/p2_calls.py
[...]
[0, 1]
(0,)
0
```

- The parameter `v` is an alias of the caller's object. `v += w` extends the shared list in place.
- For a tuple or an int, `v += w` rebinds the local `v`. So does `v = v + [1]`. The caller keeps its object.

The call as binding

```
def rhs(t, y, *args, **kwargs):
    return t, y, args, kwargs
print(rhs(0.0, 1.0))
print(rhs(0.0, 1.0, 0.8, 0.4, h=0.05))
p = {"a": 0.8, "b": 0.08}
print(rhs(0.0, 1.0, **p))
```

```
(0.0, 1.0, (), {})
(0.0, 1.0, (0.8, 0.4), {'h': 0.05})
(0.0, 1.0, (), {'a': 0.8, 'b': 0.08})
```

- A call binds the parameters in a new local namespace. Positional arguments come first, then keyword arguments, then defaults.
- `*args` is a tuple of the extra positional arguments. `**kwargs` is a dictionary of the extra keyword arguments.
- `**p` in a call unpacks a dictionary into keyword arguments.

Puzzle: a local not yet bound

```
h = 0.05
def show():
    print(h)
def shrink():
    h = h / 2
    return h
show()
```

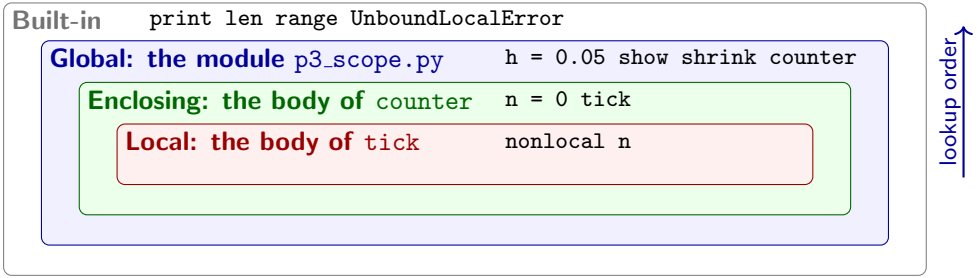
Puzzle: a local not yet bound

```
h = 0.05
def show():
    print(h)
def shrink():
    h = h / 2
    return h
show()
```

```
$ python puzzles/p3_scope.py
0.05
UnboundLocalError: cannot access local variable 'h' where it is not associated with a
    value
```

- `shrink` assigns to `h`. So `h` is local throughout its body.
- The file calls `shrink()` inside `try/except`. It prints the message.

Name lookup: the LEGB rule



- A name is looked up in the **local, enclosing, global and built-in scopes**, in that order. The first binding found is used.
- Locality is decided at compile time. An assignment anywhere in the body makes the name local, unless `global` or `nonlocal` says otherwise.

global and nonlocal

```
def counter():
    n = 0
    def tick():
        nonlocal n                    # the n of the enclosing call
        n += 1
        return n
    return tick
tick = counter()
print(tick(), tick(), counter()())  # 1 2 1
```

- `nonlocal n` makes `n += 1` act on the `n` of the enclosing call. Without it, the call raises `UnboundLocalError`.
- Each call of `counter` creates a new `n`.
- `global h` makes `h = h / 2` rebind the module's `h` (`puzzles/global_h.py`).

Closures and their cells

```
>>> def make_adder(n):
...     def add(x): return x + n
...     def zero(x): return 0 * x
...     return add, zero
...
>>> add3, zero = make_adder(3)
>>> add3(4), add3.__closure__[0].cell_contents, zero.__closure__
(7, 3, None)
>>> add5, _ = make_adder(5)
>>> add5.__code__ is add3.__code__, add5.__closure__[0] is add3.__closure__[0]
(True, False)
```

- A **factory** is a function that builds and returns another function.
- A **free variable** is a name bound in an enclosing function. A **closure** is a function plus the **cells** of its free variables.
- `add` has one cell. `zero` has none. Two calls make two cells.

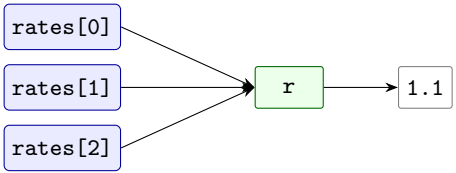
Puzzle: lambdas in a loop

```
rates = [lambda x: r * x for r in (0.5, 0.8, 1.1)]
print([f(1.0) for f in rates])
```

Puzzle: lambdas in a loop

```
rates = [lambda x: r * x for r in (0.5, 0.8, 1.1)]
print([f(1.0) for f in rates])
```

```
$ python puzzles/p4_closures.py
[1.1, 1.1, 1.1]
[...]
```



One cell holds the loop variable. The three lambdas share it. Each reads it when called, after the loop ends at 1.1.

Late binding and the three repairs

```
rates = [lambda x, r=r: r * x for r in (0.5, 0.8, 1.1)]
def scale(r, x):
    return r * x
rates = [partial(scale, r) for r in (0.5, 0.8, 1.1)]
def make_rate(r):
    return lambda x: r * x          # its own cell per call
rates = [make_rate(r) for r in (0.5, 0.8, 1.1)]
```

- Each version prints `[0.5, 0.8, 1.1]` for `[f(1.0) for f in rates]`.
- **Late binding**: a free variable is read from its cell when the function is called.
- Each repair fixes `r` inside the loop. The default `r=r` is evaluated per lambda. `partial` stores `r`. Each call of `make_rate` makes a new cell.

✎ Factory calls against lambdas in a loop

```
models = [lambda t, y: np.array([a*y[0] - 0.08*y[0]*y[1],
                                 0.77*y[0]*y[1] - 0.77*y[1]])
          for a in (0.5, 0.8, 1.1)]
print(" ".join(f"{f(0.0, y)[0]:g}" for f in models))
models = [make_lotka_volterra(a, 0.08, 0.77, 0.77) for a in (0.5, 0.8, 1.1)]
print(" ".join(f"{f(0.0, y)[0]:g}" for f in models))
```

```
$ python -m puzzles.model_loop
1.1 1.1 1.1
0.5 0.8 1.1
[...]
```

- At `y = np.array([1.0, 0.0])`, the first component of the right-hand side is a.
- The lambdas read `a` after the loop. Each factory call binds its own `a`.
- `simulate.py` calls the same factory: `make_lotka_volterra(**CHEMOSTAT)`.

Module execution at first import

```
>>> import math, importlib, sys
>>> importlib.import_module("math") is math is sys.modules["math"]
True
>>> from math import pi
>>> math.pi = 3
>>> pi, math.pi
(3.141592653589793, 3)
>>> math.pi = pi
```

- A **module** is filled by running its file once, at the first import. The import stores it in `sys.modules`.
- A later import binds that object and runs nothing.
- `from math import pi` binds `pi` to the current object. A later rebinding of `math.pi` leaves `pi` unchanged.

attractor ✎ Code run at the first import

```
import os                                                # stepsize.py
print("stepsize.py runs")
T = 40.0
H = float(os.environ.get("ATTRACTOR_H", "0.05"))    # read now
STEPS = round(T / H)
```

```
import importlib, os, sys                                # driver.py, lines 1-5
print("stepsize" in sys.modules)                         # False
from stepsize import H                                   # prints: stepsize.py runs
import stepsize                                          # prints nothing
print(stepsize is sys.modules["stepsize"], H)            # True 0.05
```

- The file runs once, at the first import. The second import binds the cached object.
- `os.environ` is read when its line runs. `simulate.py` calls `load_dotenv()` before its read.

✎ Values fixed at import

```
from stepsize import H                  # driver.py, line 3: H bound to the float 0.05
[...]
stepsize.H = 0.02                       # line 6: the module's H only
print(H, stepsize.H)                    # 0.05 0.02
```

- The assignment rebinds the module's `H` only. The `H` of `driver.py` keeps 0.05.
- A value read at import stays fixed. A new process reads it again: `ATTRACTOR_H=0.02 python driver.py` prints `0.02 0.02`.
- `importlib.reload(m)` runs the file again. Names from `from m import X` keep their old objects.

Summary

- Assignments, calls, `def` and imports bind names. They copy nothing. On a list or an array, += writes through aliases.
- A basic slice of an array is a view on its owner's memory.
- Defaults are evaluated once. Names follow the LEGB rule.
- A closure reads its cells when called. Functions made in a loop need a default, `partial` or a factory.
- A module runs once per process. A value read at import stays fixed. Next: higher-order functions and decorators.