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

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

Contents

1 Names and objects **2**
- 1.1 Assignment as binding . . . 2
- 1.2 Identity and equality . . . 3
- 1.3 Mutability and aliasing . . . 4
- 1.4 Aliasing in a loop . . . 5
- 1.5 Views of arrays . . . 7

2 Calls and scope **8**
- 2.1 The call as binding . . . 8
- 2.2 Scopes . . . 9

3 Functions as objects and closures **11**
- 3.1 Functions as objects . . . 11
- 3.2 Closures . . . 12
- 3.3 Late binding . . . 13

4 Modules and import time **14**
- 4.1 Module execution at first import . . . 15
- 4.2 Names bound by import . . . 16

Troubleshooting **17**

Key points **18**

Exercises **19**

Solutions to the exercises **21**

References **24**

A Python name is a label on an object. An assignment attaches a label. A call, a `def` and an `import` also attach labels. No binding copies the object.

Section 1 sets out names, identity and mutability. Section 2 treats calls and scopes. Section 3 treats functions as objects and closures. Section 4 treats modules and imports.

The examples come from the `attractor` project. The project measures the cycle period of ten chemostat runs. A chemostat run is an experiment in a vessel fed with fresh medium. Algae grow in the vessel, and rotifers eat the algae. This chapter adds a simulation of the cycles.

The algae density x and the rotifer density z follow the Lotka–Volterra equations

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

with $a = 0.8$, $b = 0.08$ and $c = d = 0.77$ per day.

The function that evaluates the two derivatives is called the right-hand side. The state is the pair (x, z). The integrator is the code that advances the state in small time steps. Here the integrator is the fourth-order Runge–Kutta method, RK4.

The length h of a step is the step size. The states at successive times form the trajectory. The project has three files of code:

- `models.py` holds the right-hand side;
- `integrators.py` holds the RK4 step and the loop over steps;
- `simulate.py`, the driver, calls the integrator on the right-hand side and measures the period with the peak finder of `cycles.py`.

The model and the numerical method are only background. The public repository of the project holds the code at the tags `lecture-09-start` and `lecture-09`. A tag is a name given to a commit. Each session typed at the prompt in this chapter is a file of `sessions/lecture09/`. The command `python -m doctest` replays the session files.

At `lecture-09-start` the simulated states are one array stored 801 times. As a result, every row of the trajectory shows the final state. A change of one line fixes the bug at `lecture-09` (Section 1.4).

1 Names and objects

A program that keeps intermediate results can keep a copy of a value, or a name bound to an object that may still change. The two choices give different results when the object changes later.

1.1 Assignment as binding

Definition 1.1: Name, object and binding

An **object** is a value in memory. The **type** of an object defines the operations that the object supports. The **identity** of an object distinguishes the object from all other objects that exist at the same time.

A **namespace** is a dictionary from names to objects. A **name** is an entry in a namespace. The assignment `b = a` **binds** the name `b` to the object that `a` is bound to. The assignment `b = a` copies nothing. As a result, a change made through `b` is seen through `a`.

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

In Listing 1, `b = a` binds a second name to the list. So the change made by `b.append(4)` is seen through `a`. The slice `a[:]` makes a new list. The name `c` is bound to the new list (Figure 1). A name can be bound to objects of different types in turn. For example, after `x = 1` and then `x = "one"`, `type(x)` is `str`.

```
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)
```

Listing 1: Two names on one list, a third on a copy, and two equal integers: `puzzles/p1_names.py`, lines 1–8, at `lecture-09-start`.

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

Listing 2: The run of `puzzles/p1_names.py` at `lecture-09-start`.

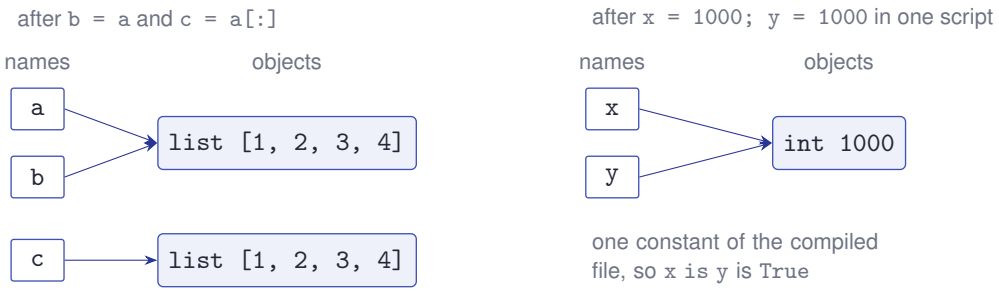


Figure 1: The names and objects of Listing 1 after its run. Left: `a` and `b` are bound to one list. The name `c` is bound to the copy made by the slice. Right: the two literals `1000` of one script are one constant object.

A chained assignment binds several names to one object. Two list literals make two objects (Listing 3).

```
>>> a = b = []
>>> a is b
True
>>> c, d = [], []
>>> c is d
False
```

Listing 3: One list behind two names, two lists behind two literals: `sessions/lecture09/binding.txt`, lines 4–9, at `lecture-09-start`.

1.2 Identity and equality

The operator `==` compares values. The operator `is` compares identities. The built-in `id(x)` returns an integer that identifies `x` while `x` exists. In CPython this integer is the address of `x` in memory.

The test `a is b` means `id(a) == id(b)`. For example, after `b = a`, `id(a) == id(b)` is `True`. The second line of Listing 2 shows two equal lists that are two objects.

Python compiles a script into **bytecode** before it runs the script. Bytecode is the list of instruc-

tions that the interpreter executes. A **literal** is a value written out in the code.

The script of Listing 1 contains two equal literals `1000`. The compiler turns the two literals into a single constant. So the third line of output prints `True` for `x is y`. At the prompt each statement is compiled on its own. So `p = 1000` and `q = 1000` create two distinct objects (Listing 4).

For 256, the test `x is y` gives `True` both at the prompt and in a script. CPython keeps one object for each integer from −5 to 256. This sharing is called **interning**. The language does not require interning.

Integers computed at run time follow the same rule. For example, `int("1000") is int("1000")` is `False`, and `int("256") is int("256")` is `True`.

```
>>> p = 1000
>>> q = 1000
>>> p is q
False
>>> x = 256
>>> y = 256
>>> x is y
True
```

Listing 4: Equal integers bound at the prompt: `sessions/lecture09/identity.txt`, lines 4–11, at `lecture-09-start`.

> The operator `is` between numbers or strings tests a detail of the implementation. Since Python 3.8, Python issues a `SyntaxWarning` for an `is` with a literal operand. The operator `is` has two intended uses: `x is None`, and a check that two names share one object.

1.3 Mutability and aliasing

Definition 1.2: Mutable and immutable objects, aliases

An object is **mutable** when its value can change after it is created. An object is **immutable** when its value cannot change. Lists, dictionaries, sets and NumPy arrays are mutable. Numbers, strings and tuples are immutable. An operation on an immutable object creates a new object.

Two names bound to one object are **aliases**. A change made through one alias is seen through the other.

In Listing 5 a write through `q` is seen through `p`. Two names on one tuple are aliases too. But no operation can change a tuple. For example, `t[0] = 1` raises `TypeError: 'tuple' object does not support item assignment`.

The statement `u += (0.77,)` creates a new tuple and binds `u` to the new tuple. The name `t` remains bound to the old tuple.

```
>>> p = {"a": 0.8}
>>> q = p
>>> q["a"] = 0.5
>>> p
{'a': 0.5}
>>> t = (0.8, 0.08)
>>> u = t
>>> u is t
True
>>> u += (0.77,)
>>> t, u is t
((0.8, 0.08), False)
```

Listing 5: Aliases of a dictionary and of a tuple: `sessions/lecture09/aliases.txt`, lines 4–15, at `lecture-09-start`.

An augmented assignment such as `x += v` is an operation and an assignment in one statement. The effect of an augmented assignment depends on whether the object is mutable (Table 1).

On a list, `x += v` extends the list in place. Every alias of the list sees the change. For example, after `w = v = [0]` and `v += [1]`, `w` is `[0, 1]`.

On a number or a tuple, `x += v` creates a new object. Only the name `x` is rebound. For example, after `n = 1`, `m = n` and `n += 1`, `m` is still `1`.

Table 1: The effect of `x += v` by type.

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

`v += [1]` changes a list that other names share. `v = v + [1]` binds `v` to a new list. The shared list keeps its value.

1.4 Aliasing in a loop

A loop that stores a state at every step must store a new object each time. In Listing 6, `state += 1` writes into the one array. So the list `rows` holds three references to the array `state`. The list `rows` shows the last value of `state` three times.

With `state = state + 1`, each pass creates a new array. The three rows then differ. The call `rows.append(state.copy())` also works. The call stores a copy of `state` at each pass. The update `state += 1` then stays unchanged.

```
>>> 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
```

Listing 6: A loop that stores one array three times: `sessions/lecture09/loop.txt`, lines 5–14, at `lecture-09-start`.

`y += ...` on a NumPy array writes into the array. Every name bound to the array sees the new values. Inside a function, `y +=` changes the caller's data. For example, take `def g(y): y += 1` and `a = np.zeros(2)`. After `g(a)`, `a` is `array([1., 1.])`.

In `attractor`

At `lecture-09-start` the RK4 step ends with an augmented assignment. The step then returns the array it received as its argument `y` (Listing 7). The driver appends each returned state to the list `rows`, as in Listing 6. So the 801 rows are one array.

The run prints the same first and last row. The run also prints a period of `nan`. The reason is that `cycle_period` of `cycles.py` finds no peak in a constant signal.

The command `python -i` runs the script and then opens the interactive prompt. The session at this prompt shows that all 801 rows have one identity (Listing 8).

```
def rk4_step(rhs, t, y, h):
    """One classical Runge-Kutta step of order four."""
    k1 = rhs(t, y)
    k2 = rhs(t + h / 2, y + h / 2 * k1)
    k3 = rhs(t + h / 2, y + h / 2 * k2)
    k4 = rhs(t + h, y + h * k3)
    y += h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)    # in place
    return y                                    # the same object
```

Listing 7: The RK4 step at the start of the lecture: `integrators.py`, lines 10–17, at `lecture-09-start`.

```
$ 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
```

Listing 8: The driver at `lecture-09-start`, run with `python -i` and inspected: `simulate.py` at `lecture-09-start`.

The fix at `lecture-09` makes the step return a new array (Listing 9). The period becomes 8.16 days. A copy in the driver would also fix the bug. However, a step that writes into its argument would require every caller to copy the returned array.

```
$ 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)
```

Listing 9: The fix: `lecture-09-start` to `lecture-09`.

1.5 Views of arrays

A slice of a list makes a new list. A basic slice of a NumPy array is a slice written `start:stop:step`. A basic slice creates a **view**. A view is a new array object that uses the memory of another array.

The attribute `base` of a view is the array that owns the memory. For an owner, `base` is `None`. For example, `np.arange(6.0).base` is `None`.

The function `np.shares_memory(a, b)` tells whether two arrays overlap in memory. The method `a.copy()` returns a new array that owns its memory.

In Listing 10, a write through the view `v` changes `a`. The views `v` and `c` are two objects. So `c is v` is `False`. The call `np.shares_memory(c, v)` shows that the two views overlap. The copy `a.copy()` shares no memory with `a`.

```
>>> 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
```

Listing 10: Two views of one array, and a copy: `sessions/lecture09/views.txt`, lines 5–16, at `lecture-09-start`.

On a list, the slice `x[::-1]` returns a copy. On an array, the slice `x[::-1]` returns a view. A write through the reversed list leaves the original list unchanged. A write through the reversed array changes the original array (Listing 11).

```
>>> l = [1.5, 8.0, 2.0]
>>> b = np.array(l)
>>> m = l[::-1]; m[0] = 0; l
[1.5, 8.0, 2.0]
>>> w = b[::-1]; w[0] = 0; b
array([1.5, 8. , 0. ])
>>> m is l, w.base is b
(False, True)
```

Listing 11: One slice on a list and on an array: `sessions/lecture09/views.txt`, lines 18–25, at `lecture-09-start`.

The call `copy.copy(obj)` makes a shallow copy of `obj`. If `obj` holds arrays, the copy holds the same arrays. For example, take `d = {"y": np.zeros(2)}`. Then `copy.copy(d)["y"] is d["y"]` is `True`.

In `attractor`

`simulate.py` passes the column `y[:, 0]` of the trajectory to `cycle_period` (`simulate.py` at `lecture-09`). The column is a view. So a write into the column would change the trajectory. The function `cycle_period` only reads the column.

2 Calls and scope

A function receives its inputs through its parameters. The body of the function can also read names defined outside the function. Python follows two rules in a call.

The first rule binds the parameters of the function. The second rule looks up the names used in the body. The binding rule decides whether a call can change the caller's objects. The lookup rule decides which binding a name in the body refers to.

2.1 The call as binding

Definition 2.1: Call binding

A call creates a new local namespace. Python binds the parameters in this namespace in three steps:

1. the positional arguments, in order;
2. the keyword arguments, by name;
3. the defaults of the parameters still unbound.

`*args` is bound to a tuple of the positional arguments left over. `**kwargs` is bound to a dictionary of the keyword arguments left over. Parameters after a bare `*` are **keyword-only**. A keyword-only parameter can be passed by name only.

Python copies no argument. Each parameter is an alias of the caller's object.

Python evaluates the defaults once, when `def` runs. The function object stores the defaults in `__defaults__` and `__kwdefaults__`.

In Listing 12 the function `rhs` returns the arguments it receives. In a call, `**p` unpacks a dictionary into keyword arguments.

The **signature** of a function is the list of its parameters with their kinds and defaults. The function `inspect.signature` returns the signature of a function. For the function `f` of the listing, the signature is `(a, b=2, *, c)`.

The parameter `c` has no default. So a call must pass `c` by name. For example, `f(1)` raises `TypeError: f() missing 1 required keyword-only argument: 'c'`.

```
def rhs(t, y, *args, **kwargs):
    return t, y, args, kwargs
print(rhs(0.0, 1.0))                      # (0.0, 1.0, (), {})
print(rhs(0.0, 1.0, 0.8, 0.4, h=0.05))    # (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, (), {'a': 0.8, 'b': 0.08})
def f(a, b=2, *, c):
    return a, b, c
print(f(1, c=3), f.__defaults__, f.__kwdefaults__)   # (1, 2, 3) (2,) None
```

Listing 12: Positional, keyword, `*args`, `**kwargs` and keyword-only parameters: `puzzles/p5_kwargs.py`, lines 1–9, at `lecture-09-start`.

A `def` statement runs once. So Python evaluates each default once. A mutable default is therefore a single object. All calls that omit the argument share the default object.

In Listing 13 the second call finds the first call's entry still in `log`. The default list is stored on the function object. For example, after the two calls, `step.__defaults__` is `([0.1, 0.2],)`.

The usual idiom uses `None` as the default. The body of the function then creates a new object when the argument is `None`.

```
def step(y, h, log=[]):      # one list, made when def runs
    log.append(h)
    return y + h, log
print(step(0.0, 0.1)[1])     # [0.1]
print(step(0.0, 0.2)[1])     # [0.1, 0.2]
```

Listing 13: A default list that outlives the call: `puzzles/p2_calls.py`, lines 1–5, at `lecture-09-start`.

A scientific computation should give the same result for the same inputs, whatever was computed before. State shared between calls, such as a mutable default, breaks this property without raising an error.

In Listing 14 the parameter `v` is an alias of the caller's object (Definition 1.2). On a list, `v += w` extends the caller's list in place. On a tuple or an integer, `v += w` binds the local `v` to a new object (Table 1).

The statement `v = v + [1]` always binds the local `v` to a new list. The list `u` therefore comes back as `[0, 1]`. The tuple `t` and the integer `n` keep their values.

```
def bump(v, w):
    v += w                   # extends a list, rebinds a tuple or an int
def regrow(v):
    v = v + [1]              # a new list, bound to the local v
u = [0]; bump(u, [1]); regrow(u); print(u)   # [0, 1]
t = (0,); bump(t, (1,)); print(t)            # (0,)
n = 0; bump(n, 1); print(n)                  # 0
```

Listing 14: += on a list, a tuple and an integer argument: `puzzles/p2_calls.py`, lines 6–12, at `lecture-09-start`.

> **In `attractor`**
>
> `models.py` holds the values of a, b, c and d in the dictionary `CHEMOSTAT`. `simulate.py` builds its right-hand side with `make_lotka_volterra(**CHEMOSTAT)`. The unpacked dictionary binds the parameters `a`, `b`, `c` and `d` of `make_lotka_volterra` by name (`simulate.py` at `lecture-09`).

2.2 Scopes

Definition 2.2: Scope and the LEGB rule

A **scope** is the part of a program in which a name refers to a given namespace. A name used in a function is looked up in four scopes, in this order:

1. **local**: the function's own namespace;
2. **enclosing**: the functions inside which the function is defined;
3. **global**: the module;

4. **built-in**: the names such as `print` and `len` that every module can use without importing them.

Python uses the first binding it finds. This lookup order is the **LEGB rule**, named after the initials of the four scopes.

A global name hides a built-in name of the same spelling. For example, in a module that runs `len = 3`, the call `len("ab")` raises `TypeError: 'int' object is not callable`.

In Listing 15, `show` has no local `h`. So `show` reads the module's `h`. The function `shrink` assigns to `h`. So `h` is local throughout `shrink`. The read on the right of `h = h / 2` finds a local name `h` that is not yet bound.

The script prints `0.05`. The script then prints `UnboundLocalError: cannot access local variable 'h' where it is not associated with a value`. Figure 2 draws the four scopes of the file.

```
h = 0.05
def show():
    print(h)                # 0.05: no local h, the global one is read
def shrink():
    h = h / 2               # h is local here, and not yet bound
    return h
```

Listing 15: A read of the module's `h`, and an assignment that makes `h` local: `puzzles/p3_scope.py`, lines 1–6, at `lecture-09-start`.

Whether a name is local is decided when the function is compiled into bytecode. Compilation happens before any line of the function runs.

An assignment anywhere in the body makes the name local throughout the body. The assignments include `+=` and the target of a `for` loop. So a read before the assignment raises `UnboundLocalError`. The read does not fall back to the module's value.

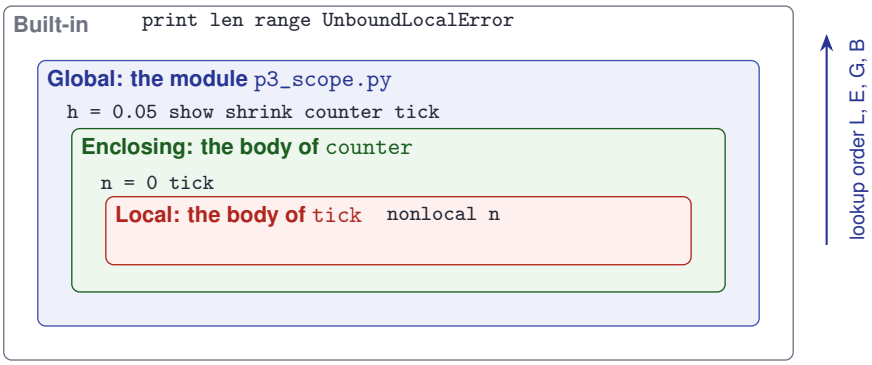


Figure 2: The four scopes of `puzzles/p3_scope.py`. Python looks up a name read in `tick` in four places in turn: the local namespace, the running `counter` call, the module and the built-ins. `nonlocal n` makes `n += 1` in `tick` act on the enclosing `n`.

A comprehension is a scope of its own. For example, `[h for h in (0.1, 0.2)]` leaves the module's `h` at `0.05` (`sessions/lecture09/scopes.txt` at `lecture-09-start`).

A function that must rebind a module name declares the name `global`. The function `halve` of `puzzles/global_h.py` at `lecture-09-start` declares `h` global. The script prints `0.025`.

A nested function rebinds a name of the enclosing function with `nonlocal`. In Listing 16 each call of `counter` creates a new `n`. So the second `counter()` starts again at 1.

Suppose the `nonlocal` line is removed. Then `n += 1` makes `n` local to `tick`. A call of `tick` then raises the `UnboundLocalError` of Listing 15.

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

Listing 16: A count kept in the enclosing scope with `nonlocal`: `puzzles/p3_scope.py`, lines 12–20, at `lecture-09-start`.

A module name rebound through `global` is a hidden input of the function. The module name is also a hidden output of the function. The call shows neither the input nor the output.

> **In** `attractor`
>
> `integrate(step, rhs, y0, t0, t1, h)` of `integrators.py` takes the step size as its parameter `h` (`integrators.py`, lines 20–29, at `lecture-09`). So each call states the step size it uses. No module name holds a step size. `simulate.py` reads the step size once, at start-up (Section 4.1).

3 Functions as objects and closures

A program can build functions while it runs. The program can build one function for each value of a parameter, in a loop or inside another function. Each such function must keep the value it was built with. A function defined in a loop reads the loop variable only when the function is called. The call comes after the loop has ended.

3.1 Functions as objects

A `def` statement creates a function object. The `def` statement binds the function's name to the function object. A function object can be used like any other object (Listing 17). A function object can be:

- bound to other names;
- stored in a list;
- passed as an argument;
- returned by another function.

The attribute `__name__` holds the name of the function. The attribute `__defaults__` holds the defaults.

The attribute `__qualname__` holds the **qualified name**. The qualified name is the name prefixed by the names of the functions or classes the function is defined inside. For example, `make_adder(3)[0].__qualname__` is `'make_adder.<locals>.add'` for the factory of Listing 18.

The attribute `__code__` holds a **code object**. The code object contains the bytecode and the names of the parameters. The attribute `__closure__` holds the cells of Definition 3.1.

```
>>> def square(x): return x * x
...
>>> def apply(f, v): return f(v)
...
>>> g = square
>>> apply(g, 3), g is square, g.__name__
(9, True, 'square')
>>> square.__defaults__, square.__code__.co_varnames
(None, ('x',))
```

Listing 17: A function bound to a second name, passed as an argument and inspected: `sessions/lecture09/functions.txt`, lines 4–12, at `lecture-09-start`.

> **In** `attractor`
>
> `integrate` receives two functions: the one-step update and the right-hand side. `integrate` calls the two functions through its parameters `step` and `rhs`, as in `step(rhs, t0 + i * h, y, h)` (`integrators.py`, lines 20–29, at `lecture-09`). So the same loop runs `euler_step` or `rk4_step`.

3.2 Closures

Definition 3.1: Closure

A **free variable** of a function is a name used in its body that is bound in an enclosing function. A free variable is neither local nor global.

A **cell** is a small object that holds one value. A cell can be shared between functions. A **closure** is a function object together with the cells that hold its free variables.

The names of the free variables are listed in `__code__.co_freevars`. The cells are stored in `__closure__`, one cell per name. The attribute `cell_contents` of a cell holds the value of the cell.

A cell is shared by the enclosing function and by each inner function that uses the name. The cell outlives the call that created it.

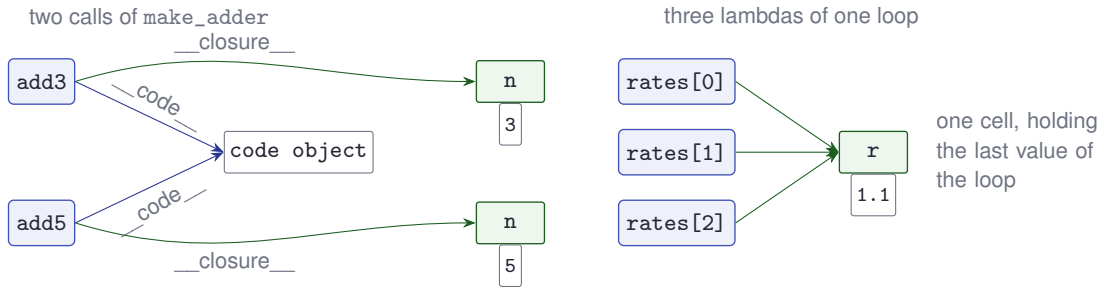


Figure 3: Closures and their cells. Left: two calls of a factory return two function objects. The two function objects share one code object. Each function object holds one cell. Right: three lambdas made in one loop share the single cell of the loop variable. When the loop ends, the cell holds the last value.

A **factory** is a function that builds and returns another function. In Listing 18, `make_adder` is a factory. The function `add` uses the parameter `n` of `make_adder`. So `add` is a closure with one cell. For example, `add3.__code__.co_freevars` is `('n',)`.

The function `zero` is defined in the same body. The function `zero` uses no name of `make_adder`. So `zero` is not a closure, and its `__closure__` is `None`.

Two calls of the factory return two functions. The two functions share one code object. Each function holds a cell of its own (Figure 3).

```
>>> 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)
```

Listing 18: A factory, a closure and a function with no free variable: `sessions/lecture09/closures.txt`, lines 4–14, at `lecture-09-start`.

In `attractor`

`make_lotka_volterra(a, b, c, d)` of `models.py` is a factory. The factory defines `rhs(t, y)` in its body and returns `rhs` (`models.py`, lines 9–14, at `lecture-09`). The four parameters are free variables of `rhs`. So the returned function carries four cells. The returned function reads the four cells at each call (Listing 19).

```
rhs = make_lotka_volterra(0.8, 0.08, 0.77, 0.77)
print(rhs.__qualname__)                          # make_lotka_volterra.<locals>.rhs
print(rhs.__defaults__, rhs.__code__.co_freevars)  # None ('a', 'b', 'c', 'd')
print([c.cell_contents for c in rhs.__closure__])  # [0.8, 0.08, 0.77, 0.77]
```

Listing 19: The cells of the right-hand side returned by the factory: `puzzles/function_objects.py`, lines 4–7, at `lecture-09-start`.

3.3 Late binding

The names in a function body are looked up when the body runs, at each call. A free variable is read from its cell at that moment. This rule is called **late binding**.

With late binding, two functions of a module can call each other in either order of definition. For example, `def f(): return g()` can come before `def g(): return 1`. The call `f()` then returns `1`.

A cell holds one variable of one call of the enclosing function. Each rebinding of the variable writes into the same cell.

The three lambdas of the first line of Listing 20 use the loop variable `r` of their comprehension. So the three lambdas share one cell (Figure 3, right, and `sessions/lecture09/closures.txt`, lines 16–22, at `lecture-09-start`). Each lambda reads `r` when it is called. By then the loop has left the value 1.1 in the cell.

Three repairs evaluate the rate `r` inside the loop.

- Python evaluates a default argument `r=r` when it creates each lambda (Definition 2.1).
- `functools.partial(scale, r)` builds a **partial** function. A partial function is a callable, an object that can be called like a function. The partial function stores `r` when it is created. At each call, the partial function passes `r` to `scale` before the call's arguments. For example, `partial(scale, 0.5)(2.0)` is `1.0`.

- Each call of the factory `make_rate(r)` binds a new `r`. So each lambda gets a cell of its own.

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

Listing 20: Three rates in a loop, the trap and three repairs: `puzzles/p4_closures.py`, lines 2–13, at `lecture-09-start`.

A parameter study is meaningful only if each run uses the parameters it was given. Functions that read their values at call time can give every run the same parameters without raising an error.

In `attractor`

Three right-hand sides that differ only in a have the same bug. The bare lambdas all return 1.1 as the first component (Listing 21).

The file `model_loop.py` imports `models`. So `model_loop.py` must run from the project root as `python -m puzzles.model_loop`. The flag `-m` runs a module that Python finds from the current directory.

```
y = np.array([1.0, 0.0])        # at z = 0 the first component is a*x = a
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))   # 1.1 1.1 1.1
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))   # 0.5 0.8 1.1
print([f.__closure__[0].cell_contents for f in models])   # [0.5, 0.8, 1.1]
```

Listing 21: Bare lambdas against factory calls, evaluated where `rhs(0.0, y)[0]` equals a: `puzzles/model_loop.py`, lines 8–15, at `lecture-09`.

4 Modules and import time

A program reads some values from the environment or from a file. A file path and a numerical parameter are examples. Several modules of the program use such values. The code that reads such a value runs when its module is first imported. So the time of the first import fixes the value that the program sees.

4.1 Module execution at first import

Definition 4.1: Module object and the module cache

A **module** is an object that holds the names defined by one Python file. Python fills the namespace of the module by running the file once, from top to bottom.

The **module cache** is the dictionary `sys.modules`. At the first import, Python stores the module object in the cache, under the module's name.

A later import of the same name finds the module object in the cache. The later import binds a name to the cached object and runs no code of the file.

The **environment variables** are named strings. The shell exports environment variables to the programs it starts. The mapping `os.environ` is the program's copy of the environment variables.

The values of `os.environ` are strings. For example, when `ATTRACTOR_H` is unset, `os.environ.get("ATTRACTOR_H", "0.05")` is the string `'0.05'`.

The module `stepsize.py` of `puzzles/imports/` prints a line when it runs. The module also reads a number from `os.environ` (Listing 22). The script `driver.py` imports `stepsize` twice (Listing 23). The file `stepsize.py` runs at the `from` import. The second import binds the cached object and prints nothing.

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

Listing 22: A module that reads the environment when it runs: `puzzles/imports/stepsize.py`, lines 1–5, at `lecture-09-start`.

```
import importlib, os, sys
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
```

Listing 23: Two imports of one module: `puzzles/imports/driver.py`, lines 1–5, at `lecture-09-start`.

Every import goes through the cache. The function `importlib.import_module` performs the same import as the `import` statement. The call `importlib.import_module("math")` returns the module object that `import math` stored (Listing 24).

```
>>> import math, importlib, sys
>>> importlib.import_module("math") is math is sys.modules["math"]
True
```

Listing 24: One module object behind two imports: `sessions/lecture09/modules.txt`, lines 4–6, at `lecture-09-start`.

A **child process** is a second program started by a running program. The script `envread.py` works in three steps (`puzzles/imports/envread.py` at `lecture-09-start`):

1. the script reads a constant from `os.environ` at import;
2. the script sets `ATTRACTOR_T1` to `"80"` in `os.environ`;

3. the script starts a second interpreter as a child process with `subprocess.run`.

The script prints `40.0 80`. The constant keeps its import-time value. The child process inherits the current environment of the parent process. In the same way, the child processes of a shell inherit the variables that the shell exports.

The top level of a module is the set of its statements outside any function. The top level runs in every program that imports the module.

In `attractor`

`simulate.py` reads the environment once, at start-up (Listing 25). `load_dotenv()` adds the variables of the file `.env` to `os.environ`. Variables already set keep their values. The next line reads `ATTRACTOR_H`.

```
from cycles import cycle_period
[...]
load_dotenv()
H = float(os.environ.get("ATTRACTOR_H", "0.05"))
```

Listing 25: An import of the driver and its read of the environment: `simulate.py`, lines 12–17, at `lecture-09`.

The functions `peaks` and `cycle_period` moved from `analyse.py` to `cycles.py` (`cycles.py` at `lecture-09`). The top level of `analyse.py` calls `load_dotenv()`, reads two paths and imports Matplotlib. The only top-level statement of `cycles.py` is `import numpy as np`.

4.2 Names bound by import

`from stepsize import H` binds `H` in `driver.py`. The name is bound to the object that `stepsize.H` is bound to at the moment of the import. The `from` import works like an assignment across two namespaces (Definition 1.1). After the import, nothing links the name `H` of `driver.py` to the module.

In Listing 26, `stepsize.H = 0.02` rebinds the module's attribute `H`. An attribute of a module is an entry of its namespace. The `H` of `driver.py` remains bound to 0.05 (Figure 4). Code that must see the current value reads `stepsize.H` at call time.

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

Listing 26: A name bound by a `from` import, and an assignment to the module: `puzzles/imports/driver.py`, lines 3–7, at `lecture-09-start`.

A value read at import stays fixed in the same way. `H` and `STEPS` of `stepsize.py` hold the value that `ATTRACTOR_H` had when the file ran. Only a new process reads the variable again.

For example, the command `ATTRACTOR_H=0.02 python driver.py` starts a new process. The script then prints `True 0.02` and `0.02 0.02`.

`importlib.reload(m)` runs the file of `m` again in the same module object. The call `importlib.reload(m)` returns the same module object. For example, `importlib.reload(math) is math` is `True`. Names bound earlier by `from m import X` remain bound to their old objects.

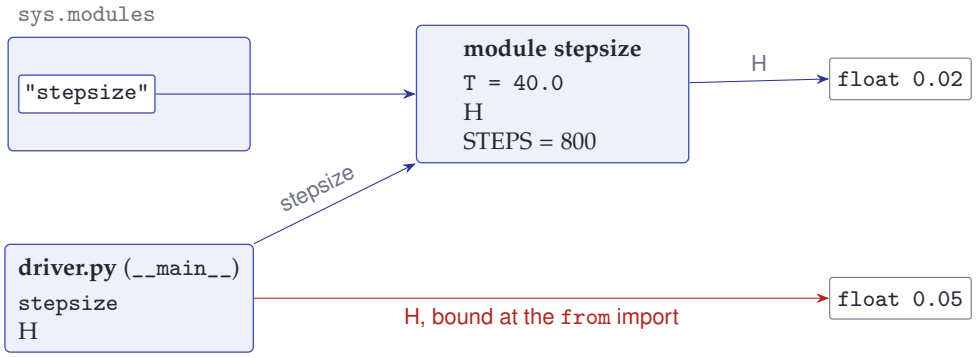


Figure 4: The objects of `driver.py` after `stepsize.H = 0.02`. The `from` import bound the `H` of `driver.py` to the float 0.05. The assignment `stepsize.H = 0.02` rebinds only the attribute of the module. The `H` of `driver.py` stays bound to 0.05.

> A module's top-level code runs once per process. An edit of the file after the import has no effect until the program starts again. A change of an environment variable after the import also has no effect until the program starts again.

A result is reproducible only if it can be regenerated from the files as they are now. A long-running notebook can hold code that no longer matches its files.

Troubleshooting

symptom	cause	remedy
`ModuleNotFoundError: No module named 'models'` from `python puzzles/model_loop.py`.	`sys.path` is the list of folders that `import` searches. A script run as a file has its own folder first on this list. The project root is not on `sys.path` (Section 3.3).	`python -m puzzles.model_loop` from the project root.
A notebook or an interactive session keeps the old version of an edited module.	The module is cached in `sys.modules`. So a second `import` runs no code of the module (Section 4.1).	Restart the kernel or the interpreter. The kernel is the Python process that runs the code of the notebook. Use `importlib.reload(m)` for a quick check.
`NameError: cannot access free variable 'k' [...]`	The inner function reads a name through its cell. The enclosing function deleted the name, or never bound the name (Definition 3.1).	Pass the value as an argument or a default.

Table 2: Objects and introspection of this chapter.

expression	effect	section
`id(x)`, `x is y`	the identity of `x`, and whether two names share one object	1.2
`a.base`, `np.shares_memory(a, b)`	the owner of a view's memory, and whether two arrays overlap	1.5
`f.__defaults__`	the defaults, evaluated once at `def`	2.1
`f.__code__.co_freevars`	the free variables of `f`, whose cells are `f.__closure__`	3.2
`functools.partial(f, *a)`	a callable that stores `a` now and passes it at each call	3.3
`sys.modules`	the module cache, the imported modules by name	4.1
`os.environ.get(name, default)`	the process's copy of the variable, read when the line runs	4.1
`importlib.reload(m)`	runs the file of `m` again in the same module object	4.2

Key points

- An assignment, a call, a `def` and an import bind names. No binding copies an object (Definition 1.1).
- On a list or an array, `+=` writes through the aliases. On a number or a tuple, `+=` rebinds the name (Definition 1.2, Table 1).
- Defaults are evaluated once, at `def` (Definition 2.1).
- A name is looked up in the local, enclosing, global and built-in scopes. An assignment anywhere in a function makes the name local throughout the function (Definition 2.2).
- A closure reads its cells when it is called. So functions defined in a loop see the last value of the loop variable. A default, `partial` or a factory fixes the value (Definition 3.1, Listing 20).
- A module's file runs once per process, at the first import. Each later import binds the module object stored in `sys.modules`. A value read at import stays fixed. A name bound by `from m import X` remains bound to its object (Definition 4.1).

Exercises

Solutions start on page 21. The scripts that check the solutions are in `exercises/lecture09/` of the project.

Rows written into an array

The loop of Listing 27 doubles a state array three times. After each pass the loop stores the state in two places. The first place is the array `out`, created before the loop with its final shape. The second place is the list `kept`.

(a) Give the two lines printed. (b) The line `state[0] = 0.0` is then run. Give `out[2]` and `kept[0]` afterwards. (c) Change the line that appends, and nothing else, so that `kept` holds the three states.

```
import numpy as np
state = np.array([1.0, 2.0])
out = np.empty((3, 2))
kept = []
for i in range(3):
    state *= 2
    out[i] = state
    kept.append(state)
print(out.tolist())
print([s.tolist() for s in kept], len({id(s) for s in kept}))
```

Listing 27: One state stored by item assignment and by `append`: `exercises/lecture09/e1_preallocated.py`, lines 1–10, at `lecture-09-start`.

A default that remembers

The function of Listing 28 records the values it has seen. (a) Give the three lists printed by `print(record(3), record(1), record(3))`. (b) The default becomes `seen=None`. The body creates a new set when `seen is None`. Give the three lists now. (c) Where is the set of (a) stored between the calls?

```
def record(x, seen=set()):
    seen.add(x)
    return sorted(seen)
```

Listing 28: A set as a default: `exercises/lecture09/e2_default_set.py`, lines 1–3, at `lecture-09-start`.

A loop variable deleted

The lines of Listing 29 run at module level. (a) Give the two lines printed. (b) The same lines are placed in the body of a function `build()`. The function is then called. Give the two lines printed now. Explain why the two error messages differ.

```
fs = []
for k in (2, 3, 5):
    fs.append(lambda: k)
print(fs[0]())
del k
try:
    print(fs[0]())
except NameError as e:
    print("NameError:", e)
```

Listing 29: Lambdas over a loop variable that is then deleted: `exercises/lecture09/e3_deleted_loop_variable.py`, lines 1–9, at `lecture-09-start`.

Rebinding against writing in place

Two loops each add `k = np.full(10**7, 0.05)` to `y = np.ones(10**7)` twenty times. One loop uses `y = y + k`. The second loop uses `y += k`.

(a) Predict which loop is faster, and give the reason (Section 1.4). (b) Time both with `time.perf_counter` and report the two times. (c) Give the reason `rk4_step` of the project uses the slower form.

Two models that share a list

The factory of Listing 30 closes over a parameter list. That is, the inner function `rhs` reads the list `params` as a free variable. Two models are made from the same list.

(a) Give the two values printed and the value of `f.__closure__[0].cell_contents is g.__closure__[0].cell_contents`. (b) Change the two factory calls, and nothing else, so that `params[0] = 1.6` changes neither model. Give two different ways.

(c) Explain why the cells of `f` and `g` are distinct objects although the test in (a) gives `True`.

```
import numpy as np
def make_growth(params):
    def rhs(t, y):
        return params[0] * y
    return rhs
params = [0.8, 0.08]
f, g = make_growth(params), make_growth(params)
params[0] = 1.6
print(f"{f(0.0, np.array([1.0]))[0]:g} {g(0.0, np.array([1.0]))[0]:g}")
```

Listing 30: Two closures over one list: `exercises/lecture09/e6_shared_cell.py`, lines 1–9, at `lecture-09-start`.

Solutions to the exercises

Rows written into an array

(a) `[[2.0, 4.0], [4.0, 8.0], [8.0, 16.0]]`, then `[[8.0, 16.0], [8.0, 16.0], [8.0, 16.0]] 1`. The item assignment `out[i] = state` copies the values of `state`. The copied values go into row `i` of the memory that `out` owns.

The call `kept.append(state)` stores the one array three times. The statement `state *= 2` writes into the array `state`.

(b) `out[2]` stays `[8.0, 16.0]`. `kept[0]` becomes `[0.0, 16.0]`. The reason is that each entry of `kept` is `state`.

(c) `kept.append(state.copy())`. `kept.append(out[i])` also works. `out[i]` is a view of row `i`. The loop writes row `i` only once.

Answer check

```
$ python exercises/lecture09/e1_preallocated.py
[[2.0, 4.0], [4.0, 8.0], [8.0, 16.0]]
[[8.0, 16.0], [8.0, 16.0], [8.0, 16.0]] 1
[8.0, 16.0] [0.0, 16.0]
[[2.0, 4.0], [4.0, 8.0], [8.0, 16.0]]
```

Listing 31: The loop, part (b), then the loop with `state.copy()`: `exercises/lecture09/e1_preallocated.py` at `lecture-09-start`.

A default that remembers

(a) `[3] [1, 3] [1, 3]`. The set is made once, when `def` runs. Each call that omits `seen` adds to the default set. (b) `[3] [1] [3]`. Each call creates its own set.

(c) In the tuple `record.__defaults__`. The tuple `record.__defaults__` is an attribute of the function object. The function object lives as long as the module.

Answer check

```
$ python exercises/lecture09/e2_default_set.py
[3] [1, 3] [1, 3]
[3] [1] [3]
```

Listing 32: `record`, then the repaired `record2`: `exercises/lecture09/e2_default_set.py` at `lecture-09-start`.

A loop variable deleted

(a) `5`, then `NameError: name 'k' is not defined`. At module level, `k` is a global name. The name `k` is not a free variable. So the lambdas have no cell. The lambdas look up `k` in the module at each call.

(b) Inside `build`, `k` is a free variable. The three lambdas share one cell. The first print again gives `5`.

`del k` empties the cell. The second print gives `NameError: cannot access free variable 'k' where it is not associated with a value in enclosing scope`.

In (a) the lookup fails in the module namespace. In (b) the lookup fails in an emptied cell.

Answer check

```
$ python exercises/lecture09/e3_deleted_loop_variable.py
5
NameError: name 'k' is not defined
None
5
NameError: cannot access free variable 'k' where it is not associated with a value in
    enclosing scope
```

Listing 33: Module level, then inside `build`; the `None` between them is `fs[0].__closure__` (Python 3.13): `exercises/lecture09/e3_deleted_loop_variable.py` at `lecture-09-start`.

Rebinding against writing in place

(a) The in-place loop. `y += k` writes into the existing array. `y = y + k` allocates a new array of 80 MB at every pass. To allocate is to reserve memory. The loop fills the new array and frees the old array.

(b) On the lecture laptop the in-place loop took 0.035 s. The rebinding loop took 0.048 s. These times are for Python 3.13 and NumPy 2.5. With Python 3.12 and NumPy 1.26 the times were 0.035 s and 0.059 s. The order held in every run.

(c) A step that writes into its argument changes the caller's array. The loop of the driver would again hold one array 801 times (Section 1.4). The integrator needs a new array per step in any case. So the allocation is not wasted.

Answer check

```
$ python exercises/lecture09/e4_inplace_cost.py
rebind   0.048 s
in place 0.035 s
```

Listing 34: Twenty passes over 10^7 values on the lecture laptop, Python 3.13 and NumPy 2.5: `exercises/lecture09/e4_inplace_cost.py` at `lecture-09-start`.

Two models that share a list

(a) `1.6 1.6` and `True`. Each closure holds a cell for its own `params`. Both cells hold the one list. The statement `params[0] = 1.6` changes the list `params` in place.

(b) The call `make_growth(tuple(params))` gives the closure an immutable copy. The calls `make_growth(list(params))` and `make_growth(params[:])` each give the closure a new list. Both models then print `0.8`.

(c) Each call of `make_growth` creates a cell. So the two cells are distinct objects. The test in (a) compares the `cell_contents` of the two cells. Both cells hold the same list.

Answer check

```
$ python exercises/lecture09/e6_shared_cell.py
1.6 1.6
True
0.8 0.8
```

Listing 35: The shared list, then the repaired calls `tuple(params)` and `list(params)`: `exercises/lecture09/e6_shared_cell.py` at `lecture-09-start`.

References

Bernd Blasius, Lars Rudolf, Guntram Weithoff, Ursula Gaedke, and Gregor F. Fussmann. Long-term cyclic persistence in an experimental predator–prey system. *Nature*, 577:226–230, 2020. Data: https://doi.org/10.6084/m9.figshare.10045976, CC BY 4.0.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. Section 17.1, Runge–Kutta method.

Boris Bolliet. `attractor`: cycle periods of the chemostat runs. GitHub, https://github.com/borisbolliet/attractor. tags `lecture-09-start` and `lecture-09`.

`doctest`: Test interactive python examples. Python 3.13 documentation, https://docs.python.org/3/library/doctest.html. accessed 25 October 2026.

Python Software Foundation. Data model: objects, values and types. Python 3.13 language reference, https://docs.python.org/3/reference/datamodel.html#objects-values-and-types. accessed 24 October 2026.

Python Software Foundation. Built-in functions: `id()`. Python 3.13 documentation, https://docs.python.org/3/library/functions.html#id. accessed 24 October 2026.

Luciano Ramalho. *Fluent Python*. O'Reilly Media, Sebastopol, CA, 2nd edition, 2022. Chapter 6, Object references, mutability and recycling; Chapter 9, Decorators and closures.

Python Software Foundation. What's new in python 3.8. Python documentation, https://docs.python.org/3/whatsnew/3.8.html. The compiler warns on `is` with a literal operand. Accessed 24 October 2026.

NumPy developers. Copies and views. NumPy 2.5 user guide, https://numpy.org/doc/stable/user/basics.copies.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.

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

Python Software Foundation. Expressions: calls. Python 3.13 language reference, https://docs.python.org/3/reference/expressions.html#calls. accessed 24 October 2026.

Python Software Foundation. Compound statements: function definitions. Python 3.13 language reference, https://docs.python.org/3/reference/compound_stmts.html#function-definitions. accessed 24 October 2026.

Python Software Foundation. Programming FAQ: why are default values shared between objects? Python 3.13 documentation, https://docs.python.org/3/faq/programming.html#why-are-default-values-shared-between-objects. accessed 24 October 2026.

Python Software Foundation. Execution model: naming and binding. Python 3.13 language reference, https://docs.python.org/3/reference/executionmodel.html#naming-and-binding. accessed 24 October 2026.

Python Software Foundation. Simple statements: the `global` statement. Python 3.13 language reference, https://docs.python.org/3/reference/simple_stmts.html#the-global-statement. accessed 24 October 2026.

Python Software Foundation. Simple statements: the `nonlocal` statement. Python 3.13 language reference, https://docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement. accessed 24 October 2026.

Python Software Foundation. Programming FAQ: why do lambdas defined in a loop with different values all return the same result? Python 3.13 documentation, https://docs.python.org/3/faq/programming.html#why-do-lambdas-defined-in-a-loop-with-different-values-all-return-the-same-result. accessed 24 October 2026.

Python Software Foundation. `functools.partial`. Python 3.13 documentation, https://docs.python.org/3/library/functools.html#functools.partial. accessed 24 October 2026.

Python Software Foundation. The import system: the module cache. Python 3.13 language reference, https://docs.python.org/3/reference/import.html#the-module-cache. accessed 24 October 2026.

Python Software Foundation. `os.environ`. Python 3.13 documentation, https://docs.python.org/3/library/os.html#os.environ. accessed 24 October 2026.

Python Software Foundation. `importlib.reload`. Python 3.13 documentation, https://docs.python.org/3/library/importlib.html#importlib.reload. accessed 24 October 2026.

Python Software Foundation. `time.perf_counter`. Python 3.13 documentation, https://docs.python.org/3/library/time.html#time.perf_counter. accessed 24 October 2026.

Index

alias, 4

binding, 2
built-in, 10
bytecode, 3

cell, 12
child process, 15
closure, 12
code object, 11

enclosing, 9
environment variables, 15

factory, 12
free variable, 12

global, 9

identity, 2
immutable, 4
interning, 4

keyword-only, 8

late binding, 13
LEGB rule, 10
literal, 4
local, 9

module, 15
module cache, 15
mutable, 4

name, 2
namespace, 2

object, 2

partial, 13

qualified name, 11

scope, 9
signature, 8

type, 2

view, 7