Lecture 10 — Higher-order functions and decorators

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

Boris Bolliet

University of Cambridge

Code and material

```
$ git clone https://github.com/borisbolliet/attractor.git ~/GitHub/attractor  # once
$ cd ~/GitHub/attractor
$ git fetch --tags
$ git switch --detach lecture-10
$ source setup_env.sh
$ python -m experiments.cost
$ python -m doctest sessions/lecture10/*.txt
```

- `git stash` first if `git status` lists local edits.
- `git diff lecture-09 lecture-10` shows what the lecture adds.
- `doctest` prints nothing when every session gives the output in the notes.

Outline

- Higher-order functions and wrappers: closures, forwarding with `*args` and `**kwargs`, `functools.wraps`
- `@deco` as `f = deco(f)`, decorator factories, import time versus call time, stacking
- Counting and timing calls: call counts, the overhead of a wrapper, a wrapper in a profile
- Caching with `functools.lru_cache`: hits, misses, hashable arguments, shared results
- Live demonstration: the lecture notebook, with a registry filled at import and a profile of a wrapper

A decorator is a function applied to a function at definition time. The wrapper it returns is a closure over the original.

Higher-order functions

```
>>> import functools
>>> sorted(["pear", "fig", "apple"], key=len)
['fig', 'pear', 'apple']
>>> from_binary = functools.partial(int, base=2)
>>> from_binary("101"), from_binary.func is int
(5, True)
```

- A **higher-order function** takes a function as an argument, returns one, or both.
- A Python function is an object. It can be passed, returned and stored in a dictionary.
- `sorted` calls its `key` once per element. `functools.partial` returns a **callable** with some arguments fixed.
- A **factory**, a function that builds and returns another function, is higher-order too.

A wrapper that counts

```
>>> def counted(f):
...     def wrapper(*args, **kwargs):
...         wrapper.calls += 1
...         return f(*args, **kwargs)
...     wrapper.calls = 0
...     return wrapper
...
>>> hyp = counted(math.hypot)
>>> hyp(3, 4), hyp(5, 12), hyp.calls
(5.0, 13.0, 2)
>>> hyp.__code__.co_freevars, hyp.__closure__[0].cell_contents
(('f', 'wrapper'), <built-in function hypot>)
```

- A **wrapper** forwards its arguments to a stored function and returns the result, with work of its own around the call.
- `wrapper` is a closure. Its cell `f` holds `math.hypot`.
- The counter is an attribute of the function object. Each wrapped function has its own.

Forwarding with *args and **kwargs

```
>>> def counted_badly(f):
...     def wrapper(x):
...         wrapper.calls += 1
...         return f(x)
...     wrapper.calls = 0
...     return wrapper
...
>>> counted_badly(abs)(-2)
2
>>> counted_badly(math.hypot)(3, 4)
TypeError: ...wrapper() takes 1 positional argument but 2 were given
```

- `*args` collects the positional arguments into a tuple. `**kwargs` collects the keyword arguments into a dictionary.
- `f(*args, **kwargs)` unpacks them again. So the wrapper serves every signature.
- Named parameters tie the wrapper to one signature.

Puzzle: a wrapper that reads the wrong name

```
f = double            # double(x) returns 2 * x, square(x) returns x * x
def counted(g):
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        return f(*args, **kwargs)
    wrapper.calls = 0
    return wrapper
d, s = counted(double), counted(square)
print(d(3), s(3))
print(s.__code__.co_freevars)
```

Puzzle: a wrapper that reads the wrong name

```
f = double            # double(x) returns 2 * x, square(x) returns x * x
def counted(g):
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        return f(*args, **kwargs)
    wrapper.calls = 0
    return wrapper
d, s = counted(double), counted(square)
print(d(3), s(3))
print(s.__code__.co_freevars)
```

```
$ python -m experiments.wrong_name
6 6
('wrapper',)
```

- The body names the global `f` (LEGB rule). So both wrappers double.
- The only free variable is `wrapper`. No cell holds the wrapped function.

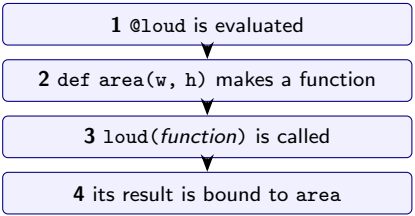
functools.wraps and the metadata of a wrapped function

```
>>> def neat(f):
...     @functools.wraps(f)
...     def wrapper(*args, **kwargs):
...         return f(*args, **kwargs)
...     return wrapper
...
>>> b, n = bare(area), neat(area)
>>> b.__name__, b.__doc__, inspect.signature(b)
('wrapper', None, <Signature (*args, **kwargs)>)
>>> n.__name__, n.__doc__, inspect.signature(n), n.__wrapped__ is area
('area', 'Area of a w by h rectangle.', <Signature (w, h)>, True)
```

- `bare` is `neat` without `wraps`. The wrapped `area` is named `wrapper` and has no docstring.
- `wraps` copies name, module, docstring and `__dict__` once, at decoration.
- It sets `__wrapped__`. `inspect.signature` follows it.

@deco as f = deco(f)

```
>>> def loud(f):
...     print("decorating", f.__name__)
...     return f
...
>>> @loud
... def area(w, h): return w * h
...
decorating area
>>> area = loud(area); area(2, 3)
decorating area
6
```



- The **decorator** runs at the definition, before any call.
- Its return value takes the name. Here it is the function itself.

Decorator factories

```
>>> import functools
>>> @functools.lru_cache(maxsize=32)
... def square(n): return n * n
...
>>> @functools.lru_cache
... def cube(n): return n ** 3
...
>>> square.cache_info().maxsize, cube.cache_info().maxsize
(32, 128)
```

- A **decorator factory** returns a decorator. `lru_cache(maxsize=32)` is evaluated first. Its result is applied to the function.
- The argument lives in a closure cell of the decorator. So each argument makes another decorator.
- Bare `@functools.lru_cache` keeps 128 entries.

A decorator factory written by hand

```
def announce(message):                # factory
    def decorator(f):                 # decorator
        def wrapper(*args, **kwargs):
            print(message)
            return f(*args, **kwargs)
        return wrapper
    return decorator

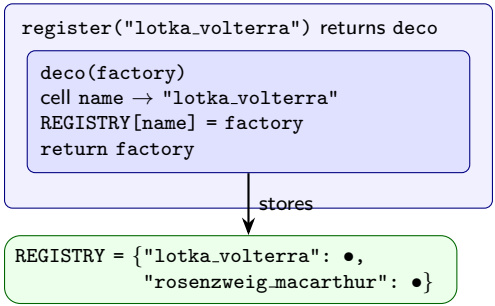
@announce("Bonjour!")
def double(x):
    return 2 * x
```

- `double(3)` prints `Bonjour!` and returns 6.
- `double.__code__.co_freevars` is `('f', 'message')`.
- So the wrapper keeps the function and the argument in two cells.

A decorator factory that fills a dictionary

```
REGISTRY = {}

def register(name):
    [...]
    def deco(factory):
        REGISTRY[name] = factory
        return factory
    return deco
```



- `register("lotka_volterra")` returns `deco`. The string sits in a closure cell.
- `deco` stores the factory and returns it unchanged. No wrapper is made.
- `make_model(name, **params)` looks a factory up by name. `make_model("goodwin")` raises `KeyError`.

Import time versus call time

```
def loud_register(name):
    def deco(f):
        print(f"decorator: registering {name}")  # once, at import
        LOUD_REGISTRY[name] = f
        @functools.wraps(f)
        def wrapper(*args, **kwargs):
            print(f"wrapper: calling {name}")    # at every call
            return f(*args, **kwargs)
        return wrapper
    return deco
```

- The top-level code of a module runs once, at its first import. A second import takes the module from `sys.modules`.
- A `def` and its decorator call are top-level code.
- The decorator's body runs at **import time**. It fills a **registry**, a dictionary from names to functions.
- The wrapper's body runs at **call time**.

✎ attractor A registry filled at import

```
$ python -c "import experiments.import_time as m; m.area(2, 3); \
    m.area(4, 5); import experiments.import_time"
decorator: registering area
wrapper: calling area
wrapper: calling area
$ python -c "import models; print(sorted(models.REGISTRY))"
['lotka_volterra', 'rosenzweig_macarthur']
$ python -c "import models, models; print(len(models.REGISTRY))"
2
```

- One decorator line, one wrapper line per call, nothing for the second import.
- The factories of `models.py` carry `@register(...)`. So importing `models` fills `REGISTRY`.
- After `from tools import REGISTRY` alone, the dictionary is empty.

✎ Stacking decorators

```
@A              # A(f) prints "A applied to", f.__name__ and returns f
@B              # B(f) likewise
def g(x):
    return x
```

```
$ python -m experiments.stack_order
B applied to g
A applied to g
[...]
```

- `@A` over `@B` over `def g` is `g = A(B(g))`.
- The decorator nearest the `def` receives the original function.
- At call time the order reverses. The name refers to the outermost wrapper.

Counting calls

```
>>> from tools import counted
>>> @counted
... def fib(n):
...     return n if n < 2 else fib(n - 1) + fib(n - 2)
...
>>> fib(10), fib.calls
(55, 177)
>>> fib.reset(); fib(20), fib.calls
(6765, 21891)
```

- A call count depends on the algorithm and its input only. So it is the same on every machine.
- The recursion calls `fib` through the rebound name. So the counter sees every inner call.
- The count grows by a factor of about 1.6 per unit of n.

Timers and overhead

```
>>> import time
>>> start = time.perf_counter()
>>> total = sum(range(10**6))
>>> elapsed = time.perf_counter() - start      # about 6 ms
>>> total, elapsed > 0
(499999500000, True)
```

- `time.perf_counter` is the monotonic clock for intervals. `time.time` is the wall clock, which the system may adjust.
- A timing wrapper reads the clock around the forwarded call. The measured time includes its **overhead**, about 150 ns per call.
- So a timer belongs on the outer function of a computation, such as one integration of about 5 ms.

✎ attractor A wrapper in a profile

```
$ python -m experiments.timing
[...]
         12008 function calls in 0.006 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      800    0.003    0.000    0.005    0.000 integrators.py:10(rk4_step)
     3200    0.002    0.000    0.002    0.000 models.py:14(rhs)
     3203    0.001    0.000    0.001    0.000 {built-in method numpy.array}
     3200    0.000    0.000    0.003    0.000 tools.py:8(wrapper)
        1    0.000    0.000    0.006    0.006 integrators.py:20(integrate)
[...]
```

- `cProfile` records every call. `tottime` is the time in a function's own lines, array operators included.
- `cumtime` adds the time of the functions it calls.
- The wrapper of `counted` has its own row, with 3 200 calls like `rhs`. Its `tottime` is the overhead.

Caching with `lru_cache`

```
>>> import functools
>>> @functools.lru_cache(maxsize=None)
... def fib(n):
...     return n if n < 2 else fib(n - 1) + fib(n - 2)
...
>>> fib(80)
23416728348467685
>>> fib.cache_info()
CacheInfo(hits=78, misses=81, maxsize=None, currsize=81)
```

- **Memoisation** stores a result under its arguments. It returns the result when they come again.
- 81 **misses** are computed and stored. 78 **hits** are read back.
- Arguments are dictionary keys. So they must be **hashable**: numbers, strings, tuples.
- `fib([80])` raises `TypeError`.

A cache returns the stored object

```
>>> import functools
>>> @functools.lru_cache
... def digits(n):
...     return [int(c) for c in str(n)]
...
>>> a = digits(2026)
>>> a.append(0)
>>> b = digits(2026)
>>> b, b is a, digits.cache_info().hits
([2, 0, 2, 6, 0], True, 1)
```

- The second call is a hit. It returns the list that the first caller changed.
- Every caller holds a name for the one stored object.
- Returning a copy prevents the change.

Decorators in libraries

- `@dataclass` rewrites a class at definition time.
- `@property` turns a method into an attribute computed on access.
- `@pytest.fixture` registers a function with the test runner at import.
- `@app.get("/")` registers a function as a route at import.
- `@jax.jit` returns a wrapper that compiles the function on its first call.

Each is a callable applied to a definition. The name is rebound to the result, as with `counted` and `register`.

Summary

- A wrapper is a closure over the function it wraps. It forwards `*args` and `**kwargs`.
- `functools.wraps` keeps the name and docstring.
- `@deco` above `def f` is `f = deco(f)`. The decorator's body runs at import time. The wrapper's body runs at every call.
- A call count is the same on every machine. A profiler shows the wrapper as a row.
- `lru_cache` needs hashable arguments. It returns the stored object itself.