Lecture 10 — Higher-order functions and decorators

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

Contents

1 Functions that take and return functions **2**
- 1.1 Higher-order functions . . . 2
- 1.2 Wrappers . . . 2
- 1.3 Forwarding of arguments . . . 4
- 1.4 Metadata of a wrapped function . . . 5

2 Decorators **6**
- 2.1 The decorator syntax . . . 6
- 2.2 Decorator factories . . . 7
- 2.3 Import time and call time . . . 8
- 2.4 Stacked decorators . . . 9

3 Counting and timing **10**
- 3.1 Counting calls . . . 10
- 3.2 Timers, overhead and profiles . . . 10

4 Caching **12**
- 4.1 Memoisation . . . 13
- 4.2 Cached objects and aliasing . . . 13

Troubleshooting **15**

Key points **15**

Exercises **16**

Solutions to the exercises **18**

References **21**

A Python function is an object. So a function can take functions as arguments and return new functions. The chapter has four sections:

- Section 1 treats higher-order functions, which take or return functions. Section 1 also treats wrappers. A wrapper is a function that forwards each call to a function stored in the wrapper.
- Section 2 treats decorators. Python calls a decorator on a function right after the definition of the function. Section 2 also shows which code runs at import and which code runs at each call.
- Section 3 counts and times calls with wrappers.
- Section 4 treats caching. A cache stores each result of a function under the arguments of the call.

A wrapper can count, time or cache the calls of a function. The source code of the wrapped function stays unchanged. The listings are at the tag `lecture-10` of the public repository

of the course project. Each session typed at the Python prompt is stored as a file in `sessions/lecture10/`. The command `python -m doctest` replays the session files.

1 Functions that take and return functions

Counting the calls of a function, or timing them, should not require editing the function. A second function can take the first function as an argument. The second function does the extra work and forwards each call to the first function. The program then calls the second function in place of the first function.

1.1 Higher-order functions

Definition 1.1: Higher-order function

A **higher-order function** is a function that takes a function as an argument, returns a function, or both.

In Python a function is an object like any other. A function can be passed, returned and stored in a dictionary, in the same way as a number or an array. A function object can also have attributes of its own.

A function stored in a dictionary is called like any other function. For example, `{"abs": abs}["abs"](-3)` is 3.

`sorted` takes a function as its `key` argument. `sorted` calls the `key` function once per element. As a result, `key=len` sorts words by their length (Listing 1).

`functools.partial` takes a function and returns a **callable**. A callable is an object that code can call like a function. The callable returned by `partial` fixes some arguments of the original function. For example, `partial(int, base=2)` converts binary numerals to integers.

A **factory**, a function that builds and returns another function, is higher-order too. A function that only calls other functions is not higher-order. For example, a function whose body calls `math.sqrt` is not higher-order. The function takes numbers and returns a number.

```
>>> 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)
```

Listing 1: Two higher-order functions of the standard library:
`sessions/lecture10/higher_order.txt`, lines 4–9, at `lecture-10`.

1.2 Wrappers

Definition 1.2: Wrapper

A **wrapper** is a function that callers call in place of a stored function. The stored function is bound when the wrapper is created.

The wrapper accepts any arguments. The wrapper forwards all these arguments to the stored function. The wrapper returns the result of the stored function. Before or after the forwarded call, the wrapper does work of its own.

A wrapper is a closure. The stored function is a free variable of the wrapper. A free variable is a name used in the body of a function but bound in the enclosing function. Python keeps the value of a free variable in a cell.

In Listing 2, `counted(f)` returns `wrapper`. Each call of `wrapper` adds one to a counter. The wrapper then calls `f` with the arguments that the wrapper received.

The counter is an attribute of the function object `wrapper`. So each wrapped function has a counter of its own. For example, after `c1, c2 = counted(abs), counted(len)` and one call `c1(-1)`, the pair `(c1.calls, c2.calls)` is `(1, 0)`.

The tuple `co_freevars` lists the names of the free variables of the wrapper code. The tuple contains `f`. The tuple also contains `wrapper` itself, because the body of the wrapper reads `wrapper.calls`. The first closure cell holds `math.hypot` (Figure 1).

```
>>> 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>)
```

Listing 2: A counting wrapper around `math.hypot`, and its closure: `sessions/lecture10/wrappers.txt`, lines 5–16, at `lecture-10`.

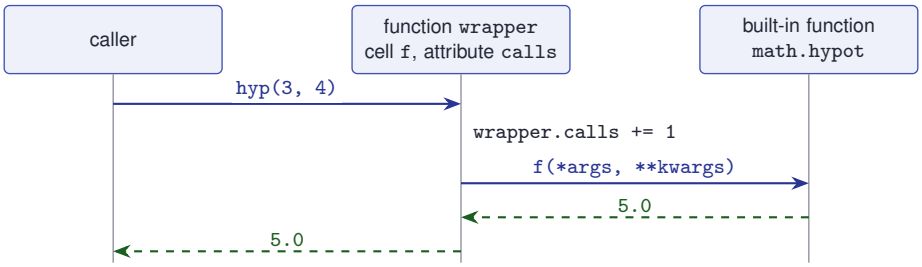


Figure 1: One call through the wrapper of Listing 2. The name `hyp` is bound to `wrapper`. The closure cell `f` of `wrapper` holds `math.hypot`. The wrapper adds one to its counter. The wrapper forwards the arguments that it received, `args = (3, 4)` and `kwargs = {}`. The wrapper returns the result unchanged.

A wrapper must call its parameter and no other function. In Listing 3 the parameter is `g`. The body calls `f`, a module-level name bound to `double`.

Python looks up the name `f` by the LEGB rule of the previous chapter. The LEGB rule searches the local scope first, then the enclosing, global and built-in scopes. Python finds no local or enclosing `f`. As a result, Python uses the global `f`.

The script first prints `6 6`, because the wrapped `square` calls `double`. The script then prints `('wrapper',)`. The only free variable of the wrapper is the name `wrapper`. No closure cell holds the wrapped function.

```
f = double

def counted(g):
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        return f(*args, **kwargs)      # the global f, never the parameter g
    wrapper.calls = 0
    return wrapper

d, s = counted(double), counted(square)
print(d(3), s(3))
print(s.__code__.co_freevars)
```

Listing 3: A wrapper that names a global instead of its parameter, below the functions `double(x)` and `square(x)`: `experiments/wrong_name.py`, lines 12–25, at `lecture-10`.

Tools that measure scientific code, such as counters and timers, must not change what the code computes. A wrapper that calls the wrong function can make two different models give the same result.

1.3 Forwarding of arguments

In the definition `def wrapper(*args, **kwargs)`, `*args` collects the positional arguments of a call into a tuple. `**kwargs` collects the keyword arguments into a dictionary. In the call `f(*args, **kwargs)`, the same prefixes unpack the tuple and the dictionary again. So `f` receives exactly the arguments that the wrapper received.

The signature of a function is the list of parameters that the function accepts. As a result, one wrapper works for functions of any signature.

For example, with `args = (3, 4)` the call `add(*args)` is `add(3, 4)`. The call `add(args)` passes one tuple instead. With `kwargs = {"x": 3, "y": 4}`, the call `add(**kwargs)` is `add(x=3, y=4)`.

A wrapper with named parameters accepts the arguments of one signature only. In Listing 4, `wrapper(x)` passes its one argument to `abs`. Python raises `TypeError` when the same wrapper receives the two arguments of `math.hypot`.

```
>>> 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
```

Listing 4: A wrapper with one named parameter: `sessions/lecture10/wrappers.txt`, lines 17–29, at `lecture-10`.

A wrapper with named parameters raises `TypeError` on a function with another number of parameters. The wrapper also raises `TypeError` when a caller passes a keyword argument that the wrapper does not define.

For example, `counted_badly(int)("101", base=2)` raises `TypeError`. A wrapper defined as `def wrapper(*args, **kwargs)` forwards both positional and keyword arguments.

1.4 Metadata of a wrapped function

A function object stores its name, its qualified name, its module and its docstring. The function object also stores a dictionary of attributes. The function `help`, tracebacks and the test runner `pytest` read the name, the docstring and the attributes.

A wrapper is a new function object. The name of the wrapper is `wrapper`. The wrapper has no docstring. As a result, code that inspects the wrapper sees the name `wrapper`, no docstring and none of the attributes of the wrapped function. `inspect.signature` also reports `(*args, **kwargs)` for the wrapper (Listing 5).

The line `@functools.wraps(f)` above the `def` of the wrapper restores the metadata of `f` on the wrapper. The line uses the decorator syntax of Section 2.1. `functools.wraps` acts on the wrapper in three steps:

1. `wraps` copies the module, name, qualified name, docstring and annotations of `f` to the wrapper.
2. `wraps` merges the attribute dictionary of `f` into the attribute dictionary of the wrapper, once.
3. `wraps` sets `wrapper.__wrapped__` to `f`.

`inspect.signature` follows `__wrapped__` to report the original parameters. For example, suppose `area.unit` is `"m2"` before the decoration. Then `neat(area).unit` is `"m2"` too.

```
>>> 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)
```

Listing 5: A wrapper under `functools.wraps`, against `bare`, the same wrapper without it, both around `area(w, h)`, which has a docstring: `sessions/lecture10/wraps.txt`, lines 14–24, at `lecture-10`.

`functools.wraps` copies the metadata once, when Python applies the decorator. A later change to the docstring of `f` does not change the docstring of the wrapper. For example, after `area.__doc__ = "changed"`, `n.__doc__` is still `'Area of a w by h rectangle.'`.

`functools.wraps` does not change the code object of the wrapper. `n.__qualname__` is `'area'`. The code object of the wrapper stores its own qualified name, `n.__code__.co_qualname`. The value of `n.__code__.co_qualname` stays `'neat.<locals>.wrapper'`.

The line `@functools.wraps(f)` goes above the inner `def wrapper`. The name `f` in `@functools.wraps(f)` is the parameter of the enclosing function.

2 Decorators

A wrapper takes effect only when callers call the wrapper under the name of the original function. Some programs choose a function by a name read at run time. A program that chooses functions by name needs a table with an entry for each function.

A decorator can bind the name of a function to a wrapper. A decorator can also add the function to a table. The decorator does either job at the definition of the function.

2.1 The decorator syntax

Definition 2.1: Decorator

A **decorator** is a callable applied to a function object. Python calls the decorator right after the `def` statement has created the function.

Python binds the return value of the decorator to the name of the function, in place of the original function. So `@deco` above `def f` is `f = deco(f)`, run right after the definition.

The decorator may return a wrapper. The decorator may also return the original function after storing it in a table, or any other object.

Figure 2 follows one decoration in four steps:

1. Python evaluates the expression after `@`.
2. The `def` statement creates the function object.
3. Python calls the decorator with the function object.
4. Python binds the return value of the decorator to the name.

In Listing 6, `loud` prints the name of the function that `loud` receives. `loud` then returns the same function. The decorator prints the line `decorating area` when the `def` statement runs, before any call of `area`. The long spelling `area = loud(area)` prints the same line again.

```
>>> 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
```

Listing 6: A decorator that prints when it is applied, in the two spellings: `sessions/lecture10/decorator.txt`, lines 4–14, at `lecture-10`.

The return value of a decorator need not be a function. `staticmethod` and `property` are decorators in daily use inside a class body. Each of `staticmethod` and `property` returns an object that holds the function. The returned object is not a function. For example, `type(property(abs))` is `property`. The value of `callable(property(abs))` is `False`.

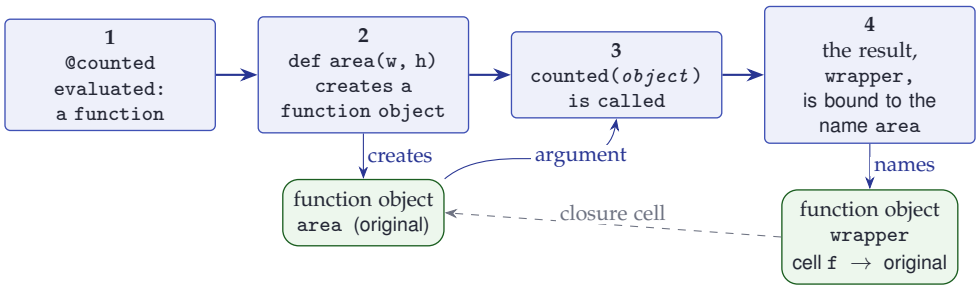


Figure 2: One decoration, `@counted` above `def area`, with the counter of Listing 2. Python evaluates the decorator expression. The `def` statement creates the original function object. Python calls `counted` with the original function object. Python binds the returned wrapper to the name `area`. After the binding, the original function is reachable only through the closure cell of the wrapper.

2.2 Decorator factories

Definition 2.2: Decorator factory

A **decorator factory** is a function that returns a decorator. Code calls a decorator factory to configure a decorator with arguments.

In `@factory(arg)`, Python first evaluates the call `factory(arg)`. The call returns a decorator. Python then applies the returned decorator to the function.

The argument of the factory is stored in a closure cell of the decorator. As a result, one factory returns a different decorator for each argument.

`functools.lru_cache`, the caching decorator of Section 4.1, is a decorator factory. The call `lru_cache(maxsize=32)` returns a decorator that keeps at most 32 results (Listing 7).

Since Python 3.8, the bare form `@functools.lru_cache` works as well. When `lru_cache` receives a function as its argument, `lru_cache` decorates the function with the default of 128 entries.

The call `functools.wraps(f)` of Section 1.4 is a factory call too. The call `functools.wraps(f)` returns the decorator that copies the metadata of `f`.

A factory written by hand nests three functions. `announce(message)` returns `decorator(f)`. In turn, `decorator(f)` returns `wrapper`.

Suppose `@announce("Bonjour!")` stands above `def double`. Each call of `double` then prints the message before the body of `double` runs. The tuple `double.__code__.co_freevars` is `('f', 'message')`. As a result, the wrapper keeps the function and the argument of the factory in two closure cells. The first exercise uses the same three nested functions.

```
>>> 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)
```

Listing 7: `lru_cache` with an argument and bare: `sessions/lecture10/lru_factory.txt`, lines 4–12, at `lecture-10`.

In `attractor`

`tools.py` defines a dictionary `REGISTRY` and a decorator factory `register(name)` (Listing 8). The decorator `deco` returned by `register` stores the decorated function in `REGISTRY` under the name from its closure. `deco` returns the function unchanged. As a result, no wrapper is created.

`make_model(name, **params)` looks up a model factory in `REGISTRY` by name. `make_model` then calls the model factory with the parameters (`tools.py, lines 47–49, at lecture-10`). For example, `make_model("goodwin")` raises `KeyError`. No factory is registered under the name `"goodwin"`.

```
REGISTRY = {}

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

Listing 8: A dictionary and a decorator factory that fills it: `tools.py, lines 36–44, at lecture-10`.

2.3 Import time and call time

The top-level code of a module is made of the statements outside any function or class. Python runs the top-level code once, at the first import of the module. Python then keeps the module object in `sys.modules`, the module cache of the previous chapter.

A second import returns the cached module without running the top-level code again. For example, after `import math`, `importlib.import_module("math") is math` is `True`.

A `def` and the decorator call attached to it are top-level code. As a result, the body of a decorator runs at **import time**, when Python imports the module. The body of the decorator runs once per decorated function. The body of a wrapper runs at **call time**, once per call of the wrapper.

In Listing 9, the decorator prints a line when Python applies the decorator. The wrapper prints a line at each call. The run of Listing 10 prints the line of the decorator once. The run prints the line of the wrapper at each of the two calls. The second import prints nothing.

```
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
```

Listing 9: A decorator factory with a print at import and one at each call, applied in the same file as `@loud_register("area")` to `area(w, h)`: `experiments/import_time.py, lines 11–21, at lecture-10`.

```
$ 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
```

Listing 10: One import, two calls, and a second import that runs nothing: `experiments/import_time.py` at `lecture-10`.

Decorators add counting, timing or caching to many functions without editing the functions. The scientific code then stays separate from the tools that measure it.

A **registry** is a dictionary from names to functions. A program uses a registry to call a function chosen by a name read at run time. A decorator adds each function to the registry when the function is defined.

`LOUD_REGISTRY` of Listing 9 is a registry. After the import, `LOUD_REGISTRY` holds the original `area` under the key `"area"`. For example, `m.LOUD_REGISTRY["area"](2, 3)` returns `6`. The call prints no wrapper line, because the registry holds the original function.

> Python fills a registry when Python imports the module that defines the decorated functions. An import of the module that defines only the dictionary leaves the dictionary empty.

In `attractor`

The two model factories of `models.py` are decorated with `@register("lotka_volterra")` and `@register("rosenzweig_macarthur")` (`models.py, lines 11–21, at lecture-10`). As a result, `import models` fills `REGISTRY` (Listing 11). The one dictionary is bound to two names, `tools.REGISTRY` and `models.REGISTRY`.

The function that `make_model` builds by name is the closure `rhs` defined inside the factory. The qualified name `'make_lotka_volterra.<locals>.rhs'` shows that `rhs` is defined inside `make_lotka_volterra`.

```
>>> from tools import REGISTRY, make_model
>>> import models
>>> sorted(REGISTRY)
['lotka_volterra', 'rosenzweig_macarthur']
>>> REGISTRY is models.REGISTRY
True
>>> rhs = make_model("lotka_volterra", **models.CHEMOSTAT)
>>> rhs.__qualname__
'make_lotka_volterra.<locals>.rhs'
```

Listing 11: The registry after the import of `models`, and a right-hand side built by name: `sessions/lecture10/registry.txt, lines 4–12, at lecture-10`.

2.4 Stacked decorators

Several decorators written above one `def` are **stacked** decorators. Python applies stacked decorators from the bottom up. The decorator nearest the `def` receives the original function. Each higher decorator receives the result of the decorator directly below. So `@A` over `@B` over `def g` is `g = A(B(g))`.

Listing 12 prints `B applied to g` before `A applied to g`. At call time, the order is reversed. The name `g` refers to the outermost wrapper. The outermost wrapper calls the next wrapper

inwards.

```
def A(f):
    print("A applied to", f.__name__)
    return f
[...]
@A
@B
def g(x):
    return x
```

Listing 12: Two stacked decorators that print when applied: `experiments/stack_order.py`, lines 8–21, at `lecture-10`.

3 Counting and timing

Two methods that compute the same result are compared by the work that each method needs for a given accuracy. A count of the calls of the expensive function measures the work of a method. The call count is the same on every machine. A clock measures the time a user waits.

3.1 Counting calls

A count of calls depends on the algorithm and its input only. A decorator can apply the counter of Section 1.2 to a recursive function.

The recursive calls look up the function by its module-level name. The decoration has bound the module-level name to the wrapper. As a result, the counter also counts the recursive calls. For example, `fib(2)` gives `fib.calls` the value 3. The three calls are `fib(2)`, `fib(1)` and `fib(0)`.

In Listing 13, the naive Fibonacci recursion makes 177 calls for `fib(10)`. The same recursion makes 21 891 calls for `fib(20)`. So the count grows by a factor of about 1.6 for each unit of n. For example, `21891 / 177` is about 124, and `1.62**10` is about 124 too. Storing each value of `fib` once removes the exponential growth of the count (Section 4.1).

```
>>> 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)
```

Listing 13: Calls of the naive Fibonacci recursion under the counter of `tools.py`: `sessions/lecture10/counting.txt`, lines 4–12, at `lecture-10`.

3.2 Timers, overhead and profiles

A measured time depends on the machine as well as on the algorithm. `time.perf_counter()` returns the value of the clock with the highest resolution for measuring an interval.

The `perf_counter` clock is monotonic. A monotonic clock never goes backwards. So a difference of two readings is never negative. The zero point of the clock is undefined. So one reading

means nothing on its own (Listing 14).

`time.time()` returns the wall-clock time. The wall-clock time is the number of seconds since 1 January 1970, as the system clock shows it. The operating system may adjust the system clock while a program runs.

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

Listing 14: Two readings of the interval clock around a sum: `sessions/lecture10/timer.txt`, lines 4–9, at `lecture-10`.

A timing wrapper reads the clock before and after the forwarded call. The timing wrapper adds the difference to an attribute, as the counter adds one.

Each call runs in a **frame**. A frame is the record that Python keeps on the call stack for the local names of one call. The wrapper calls the wrapped function. So the frame of the wrapped function lies above the frame of the wrapper on the call stack (Figure 3).

The two clock readings also bracket the cost of the extra call through the wrapper. So the measured time includes part of the **overhead**. The overhead is the work that the wrapper does itself, plus the cost of the extra call. For a timing wrapper, the overhead is about 150 ns per call on the lecture laptop (the fourth exercise).

On a function that runs for milliseconds, an overhead of 150 ns is negligible. A function of about 30 ns called a million times is a different case. The function spends about 0.15 s in its wrapper. The work of the function itself takes about 0.03 s. As a result, a timer belongs on the outer function of a computation.

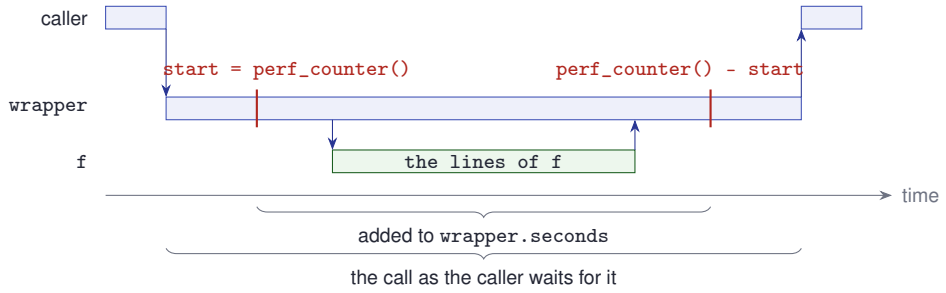


Figure 3: One call through a timing wrapper. The wrapper calls `f`, so the frame of `f` lies above the frame of the wrapper. The two clock readings bracket the forwarded call. So the measured interval holds the time of `f` and part of the work of the wrapper. A profiler records the two frames as two rows.

A **profiler** is a tool that records each function call of a run with its time. A programmer runs a profiler to find which functions take the time of a run. `cProfile` is the profiler of the standard library. `cProfile` reports four values per function:

- `ncalls`, the number of calls.
- `tottime`, the time in the function's own lines.
- `cumtime`, the time in the function, including the functions that the function calls.
- `percall`, `tottime` or `cumtime` divided by `ncalls`.

An operator on arrays, such as `*` or `**`, runs in compiled code without a Python function call. As a result, `cProfile` adds the time of the operator to the `tottime` of the function whose line uses the operator.

A wrapper is a function. So the wrapper has a row of its own. `cProfile` labels the row by the file, the line and the name in the `def` of the wrapper. `functools.wraps` does not change the label. The `tottime` of the wrapper row is the overhead of the wrapper.

The bookkeeping of a profiler slows each call that the profiler records. A benchmark measures how long one operation takes. A benchmark repeats the operation alone and times the repetitions with `perf_counter`. The fourth exercise is a benchmark.

In `attractor`

The decorator `timed` of `tools.py` has the same structure as `counted`. The wrapper of `timed` reads the clock before and after the forwarded call (`tools.py`, lines 21–33, at `lecture-10`). The wrapper adds each interval to `wrapper.seconds`. The wrapper counts the calls in `wrapper.calls`. The wrapper prints nothing.

One run of 800 RK4 steps through `timed(integrate)` takes about 5 ms on the lecture laptop. The overhead of the wrapper is paid once per run.

Listing 15 shows the profile of one call of `integrate` without `timed`. The right-hand side is decorated with `counted`. The wrapper has its own row, `tools.py:8(wrapper)`.

The wrapper row shows 3 200 calls, the same number as the row of `rhs`. Each RK4 step evaluates the right-hand side four times. For example, `800 * 4` is 3 200. The `cumtime` of the wrapper includes the time of `rhs`.

The array arithmetic of the RK4 step has no row. The time of the array arithmetic is in the `tottime` of `rk4_step`. The `tottime` of `rk4_step` is the largest of the run.

```
$ 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)
[...]
```

Listing 15: One run of `integrate` under `cProfile`, sorted by own time: `experiments/timing.py` at `lecture-10`.

Computing time is limited, on a laptop as on a cluster. A measurement before an optimisation directs the effort to the part of the code that takes the time.

4 Caching

A function called again with the same arguments repeats its work. The work may be a computation or the reading of a file. A cache stores each result under the arguments of the call. With a cache, the function does the work once per distinct argument. The cache costs memory. The cache also makes callers share the stored results.

4.1 Memoisation

Definition 4.1: Memoisation

Memoisation stores the result of a function call under the arguments of the call. A memoised function returns the stored result when a caller passes the same arguments again.

Memoisation is correct for a function whose result depends on its arguments alone. For a function of this kind, a repeated call would return the stored value anyway.

`functools.lru_cache` is the memoising decorator of the standard library. A **cache** is the store of results that a memoised function reads before computing. The cache of `lru_cache` is a dictionary keyed by the arguments. The argument `maxsize` sets the size of the cache:

- with an integer `maxsize`, the cache holds at most `maxsize` entries, and when the cache is full, `lru_cache` evicts the least recently used entry first;
- with `maxsize=None`, the cache is unbounded.

The line `@functools.lru_cache(maxsize=None)` above the `def` turns the naive Fibonacci recursion of Section 3.1 into a chain of stored values (Listing 16). `fib(80)` makes 81 **misses**. A miss is a call whose arguments are not yet in the cache. The wrapper computes and stores the result of a miss.

`fib(80)` has one miss per distinct argument, from 0 to 80. `fib(80)` also makes 78 **hits**. A hit is a call that the wrapper answers from the cache (Figure 4). As with the counter, each recursive call goes through the wrapper.

A bounded cache can miss on an argument it has already seen. For example, under `lru_cache(maxsize=2)`, the arguments 1, 2, 3 and 1 make four misses. The call with 3 evicted the entry for 1.

The arguments are the keys of the cache dictionary. As a result, the arguments must be **hashable**. A hashable object is an object whose hash value never changes. A dictionary uses the hash value to find a key.

Numbers, strings and tuples of numbers and strings are hashable. Python raises `TypeError` when code hashes a list or an array. For example, `hash(np.zeros(3))` raises `TypeError: unhashable type: 'numpy.ndarray'`.

A cache pays off when a call is slow. Suppose the body of a cached `square` calls `time.sleep(0.5)`. Then `square(30)` takes 0.5 s at its first call. The second call of `square(30)` returns at once.

The dictionary finds a key by its hash and then compares keys with `==`. The dictionary does not compare identities. For example, `tuple([1, 2]) is tuple([1, 2])` is `False`. The two tuples have equal hashes and compare equal. As a result, each tuple finds the entry stored under the other tuple.

> A list or a NumPy array as the argument of a cached function raises `TypeError: unhashable type`. A path, a size or a tuple of parameters can be the argument instead. The cached function then builds the array inside its body.

A cache can save hours in an analysis that repeats the same computations. A cached result is correct only while everything it depends on is unchanged.

4.2 Cached objects and aliasing

A cache returns the stored object itself. The cache makes no copy. As a result, the aliasing rule of the previous chapter applies. Each caller that receives a cached result holds a name for the

```
>>> 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)
>>> fib([80])
[...]
TypeError: unhashable type: 'list'
```

Listing 16: The memoised Fibonacci recursion, 81 misses and 78 hits for `fib(80)`, and an unhashable argument rejected: `sessions/lecture10/lru_cache.txt`, lines 4–16, at `lecture-10`.

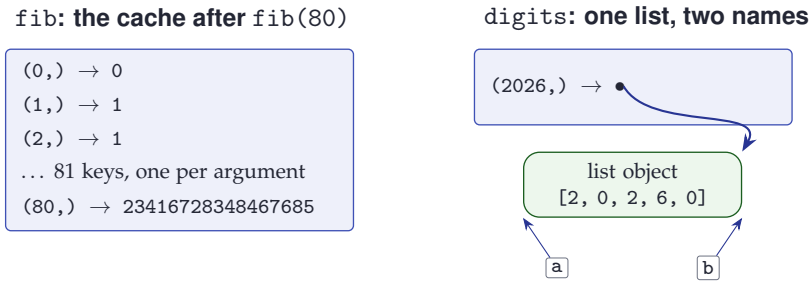


Figure 4: The cache is a dictionary keyed by the arguments. Left: after `fib(80)`, one entry per distinct argument. Right: the cached `digits` of Listing 17 holds one list for the argument 2026, and the two callers that asked for it hold that list, not copies.

same stored object.

In Listing 17, a caller appends to the list returned by the cached `digits`. The next call with the same argument returns the changed list from the cache.

```
>>> 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)
```

Listing 17: A cached list changed by a caller: `sessions/lecture10/shared.txt`, lines 4–13, at `lecture-10`.

> A caller that writes into a cached result changes what every later caller receives. Python raises no error.

A cache that returns a copy prevents the shared change. The fifth exercise implements a cache that returns copies. A read-only stored array also prevents the shared change. Setting `a.flags.writeable` to `False` makes the array `a` read-only. For example, a write `a[0] = 1` then raises `ValueError: assignment destination is read-only`.

Troubleshooting

symptom	cause	remedy
`help` on a decorated function shows `wrapper` and no docstring.	The decorator does not use `functools.wraps` (Section 1.4).	`@functools.wraps(f)` above the inner `def wrapper`.
A counter or timer on a stacked decorator stays at its initial value.	`wraps` copied the attributes of the inner wrapper once, at decoration. The outer wrapper holds that copy (Section 1.4).	The attribute read through `__wrapped__`.
A profile shows a row named `wrapper`.	Each wrapper runs in a frame of its own (Section 3.2).	The wrapped function's row for its time, the wrapper's row for the overhead.

Key points

- `@deco` above `def f` is `f = deco(f)`. The body of the decorator runs once, at import. The body of the wrapper runs at every call (Definition 2.1, Listing 10).
- A wrapper is a closure over the function it wraps. The wrapper forwards `*args` and `**kwargs`. `functools.wraps` copies the metadata once and sets `__wrapped__` (Definition 1.2, Listing 5).
- A decorator factory returns a decorator. The argument of the factory is stored in a closure cell. Importing the module with the decorated definitions fills a registry (Definition 2.2, Listing 11).
- A call count measures cost in the same way on every machine (Listing 13). A profiler shows a wrapper as a row of its own (Listing 15).
- `lru_cache` keys results by hashable arguments. `lru_cache` returns the stored object itself. Every caller shares the stored object (Definition 4.1, Listing 17).

Table 1: Decorators and tools of this chapter, with the section that explains each.

object	effect	section
`counted(f)`, `timed(f)`	wrappers with `.calls`, and `.seconds`, in `tools.py`	1.2, 3.2
`register(name)`, `REGISTRY`, `make_model(name, **params)`	decorator factory storing a function in the dictionary, filled by `import models`, and the lookup by name	2.2, 2.3
`functools.wraps(f)`, `f.__wrapped__`	copies name, docstring and attributes of `f` to the wrapper, and the function under it	1.4
`functools.lru_cache(maxsize)`, `f.cache_info()`	memoising decorator keyed by the arguments, and its counters	4.1
`time.perf_counter()`, `cProfile.run(statement)`	monotonic clock for intervals, and call counts and times per function	3.2

Exercises

Solutions are given from page 18.

Order of application

The decorator returned by the factory below prints a line when Python applies the decorator. The wrapper prints a line when the wrapper is called.

Three decorators made by the factory are stacked on a function `g`. From the top, the three decorators are `outer`, `middle` and `inner`. The function `g` prints `g runs`. The script then prints `--- one call` and calls `g()` once.

(a) Give the eight lines printed. (b) Give, as one bullet point, the rule for the order of application and the rule for the order of the calls.

```
def announce(name):
    def deco(f):
        print(f"{name} applied")
        def wrapper(*args, **kwargs):
            print(f"{name} called")
            return f(*args, **kwargs)
        return wrapper
    return deco
```

Listing 18: The announcing decorator factory: `exercises/lecture10/e1_order.py`, lines 1–8, at `lecture-10`.

A counter shared by mistake

A version of `counted` keeps its count in a module-level list `CALLS`. This version of `counted` binds the list `CALLS` to `wrapper.calls`. The decorator is applied to two functions with empty bodies, `ping` and `pong`.

(a) Give the line printed. (b) Give, as one bullet point, why the two counts are equal. (c) Change the decorator so that each function has its own count. The last two lines stay unchanged.

```
CALLS = [0]
def counted(f):
    def wrapper(*args, **kwargs):
        CALLS[0] += 1
        return f(*args, **kwargs)
    wrapper.calls = CALLS
    return wrapper
[...]
ping(); pong(); pong()
print(ping.calls, pong.calls, ping.calls is pong.calls)
```

Listing 19: A counter kept in a module-level list: `exercises/lecture10/e2_shared_counter.py`, lines 1–13, at `lecture-10`.

A registry that is empty

The script below runs as `python -m exercises.lecture10.e3_registry` from the project root. (a) Give the three lines that the script prints. (b) Give, as one bullet point, why the first dictionary is empty. (c) Give, as one bullet point, why the reload does not duplicate the entries.

```
import importlib
from tools import REGISTRY
print(REGISTRY)
import models
print(sorted(REGISTRY),
      id(REGISTRY["lotka_volterra"]) == id(models.make_lotka_volterra))
m = importlib.reload(models)
print(sorted(REGISTRY), REGISTRY["lotka_volterra"] is models.make_lotka_volterra)
```

Listing 20: The registry printed before and after `import models`, and once more after `importlib.reload(models)`: `exercises/lecture10/e3_registry.py`, lines 2–9, at `lecture-10`.

The overhead of a wrapper

Time 10^6 calls of a function with an empty body. Time the bare function, the function under `counted` and the function under `timed`. Place `time.perf_counter` around the loop.

(a) Report the three times and the time per call in nanoseconds. (b) Give the overhead per call of each decorator. Then consider `timed` on the right-hand side, called 3 200 times in an integration of about 5 ms. Say whether `timed` would change the time of the integration by a measurable amount.

```
def empty(): pass
N = 10**6
PAIRS = (("bare", empty), ("counted", counted(empty)), ("timed", timed(empty)))
for name, f in PAIRS:
    start = time.perf_counter()
    for _ in range(N):
        f()
    dt = time.perf_counter() - start
    print(f"{name:8s} {dt:.3f} s  {dt / N * 1e9:.0f} ns per call")
```

Listing 21: The timing loop: `exercises/lecture10/e4_overhead.py`, lines 4–12, at `lecture-10`.

A cache that must not share

Write a decorator `cached_copy` that caches like `lru_cache`. The wrapper returns `copy.deepcopy` of the stored value. Apply `cached_copy` to `grid(n)`, which returns `np.linspace(0.0, 1.0, n)`. Call `grid` twice with the same argument. Show with `is`, `np.shares_memory` and `cache_info` that the second call was a hit. Show also that the two results are distinct arrays.

Solutions to the exercises

Order of application

(a) Python prints the three "applied" lines at the definition. The order of the "applied" lines is inner, middle, outer. The reason is that `g = outer(middle(inner(g)))` applies the decorator nearest the `def` first.

The call `g()` prints outer, middle, inner. Each wrapper calls the next wrapper inwards. Then `g` prints `g runs`.

(b)

- Decorators apply from the bottom up at the `def`. The wrappers run from the top down at the call. The reason is that the name `g` refers to the outermost wrapper.

Answer check

```
$ python exercises/lecture10/e1_order.py
inner applied
middle applied
outer applied
--- one call
outer called
middle called
inner called
g runs
```

Listing 22: The run of `exercises/lecture10/e1_order.py` at `lecture-10`.

A counter shared by mistake

(a) `[3] [3] True`. (b)

- Both wrappers add to the one module-level list. Both `calls` attributes refer to the list `CALLS`. So the three calls are counted together. The identity test is `True`.

(c) Each wrapper needs its own counter. The corrected `counted` sets `wrapper.calls = 0`. The wrapper runs `wrapper.calls += 1`, as in Listing 2. Python then creates a new counter at each call of `counted`, once per decorated function. The last line prints `1 2 False`.

Answer check

```
$ python exercises/lecture10/e2_shared_counter.py
[3] [3] True
```

Listing 23: The run of `exercises/lecture10/e2_shared_counter.py` at `lecture-10`.

A registry that is empty

(a) The script prints the empty dictionary first. The script then prints the two names with `True`. The script prints the two names with `True` again. (b)

- `from tools import REGISTRY` runs `tools.py`. The file `tools.py` creates the dictionary with no entries. The decorated definitions of `models.py` add the entries. Python runs the decorated definitions of `models.py` at `import models`.

(c)

- `reload` runs `models.py` again in the same module object. The two decorated definitions create new function objects. The decorators store the new function objects under the same two keys of the one dictionary in `tools`. The module `tools` is not reloaded. As a result, the new function objects replace the two entries. The dictionary keeps two keys. The identity test is `True` again.

Answer check

```
$ python -m exercises.lecture10.e3_registry
{}
['lotka_volterra', 'rosenzweig_macarthur'] True
['lotka_volterra', 'rosenzweig_macarthur'] True
```

Listing 24: The run of `exercises/lecture10/e3_registry.py` at `lecture-10`.

The overhead of a wrapper

(a) On the lecture laptop, the bare call takes about 30 ns. The counted call takes about 100 ns. The timed call takes about 180 ns.

(b) The overhead is about 75 ns per call for `counted`. The overhead is about 150 ns per call for `timed`. The overhead of `timed` is larger because of the two clock readings.

The right-hand side is called 3 200 times in one integration. On the right-hand side, `timed` would add about 0.5 ms to a run of about 5 ms. An addition of 0.5 ms is a tenth of the run. A tenth of the run is measurable. As a result, a timer belongs on `integrate`, and a counter belongs on `rhs`.

Answer check

```
$ python -m exercises.lecture10.e4_overhead
bare     0.030 s  30 ns per call
counted  0.103 s  103 ns per call
timed    0.182 s  182 ns per call
```

Listing 25: A run of `exercises/lecture10/e4_overhead.py` at `lecture-10`; the times vary between runs.

A cache that must not share

`cached_copy` applies `lru_cache` to the function. `cached_copy` then wraps the cached function in a second wrapper. The second wrapper returns a deep copy of each value that the cache returns (Listing 26). The second wrapper exposes the counters of the cache through the attribute `cache_info`.

A shallow copy of an object that holds arrays would share the arrays with the stored object. A deep copy also copies the arrays. As a result, `copy.deepcopy` works for an object that holds arrays as well as for a bare array.

Answer check

```
def cached_copy(f):
    cached = functools.lru_cache(maxsize=None)(f)
    @functools.wraps(f)
    def wrapper(*args):
        return copy.deepcopy(cached(*args))
    wrapper.cache_info = cached.cache_info
    return wrapper
```

Listing 26: A caching decorator that returns copies: `exercises/lecture10/e5_cached_copy.py`, lines 4–10, at `lecture-10`.

The script applies `cached_copy` to `grid`. The script calls `grid` twice with 5. The script prints `False False` for `a is b` and `np.shares_memory(a, b)`. The script then prints `CacheInfo(hits=1, misses=1, maxsize=None, currsize=1)`. So the two results are distinct arrays with no shared memory. The second call was a hit.

References

Python Software Foundation. Command line and environment: the `-m` option. Python 3.13 documentation, https://docs.python.org/3/using/cmdline.html#cmdoption-m. accessed 24 October 2026.

Boris Bolliet. `attractor`: cycle periods of the chemostat runs. GitHub, https://github.com/borisbolliet/attractor. accessed 24 October 2026.

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

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

Kevin D. Smith, Jim J. Jewett, Skip Montanaro, and Anthony Baxter. PEP 318 – decorators for functions and methods. Python Enhancement Proposals, https://peps.python.org/pep-0318/, 2003. accessed 24 October 2026.

Python Software Foundation. Function definitions. The Python Language Reference, 3.13, https://docs.python.org/3/reference/compound_stmts.html#function-definitions. accessed 24 October 2026.

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

Python Software Foundation. The import system. The Python Language Reference, 3.13, https://docs.python.org/3/reference/import.html. 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.

Python Software Foundation. The python profilers. Python 3.13 documentation, https://docs.python.org/3/library/profile.html. accessed 24 October 2026.

Python Software Foundation. Glossary: hashable. Python 3.13 documentation, https://docs.python.org/3/glossary.html#term-hashable. 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.flags`. NumPy 2.5 reference, https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flags.html. accessed 24 October 2026.

Index

cache, 13
call time, 8
callable, 2

decorator, 6
decorator factory, 7

factory, 2
frame, 11

hashable, 13
higher-order function, 2
hits, 13

import time, 8

memoisation, 13
misses, 13

overhead, 11

profiler, 11

registry, 9

stacked, 9

wrapper, 2