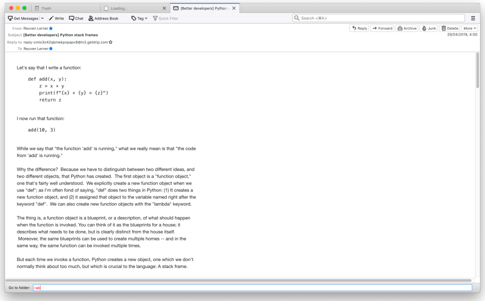
Practical Decorators

Reuven M. Lerner • PyCon 2019

reuven@lerner.co.il • @reuvenmlerner

Free “Better developers” weekly newsletter



From Reuven Lerner
Subject [Better developers] Python stack frames
Reply to reply-cmio3z42qbmekpopapv8@in2.getdrip.com
To Reuven Lerner

Let's say that I write a function:

```
def add(x, y):
    z = x + y
    print(f"{x} + {y} = {z}")
    return z
```

I now run that function:

```
add(10, 3)
```

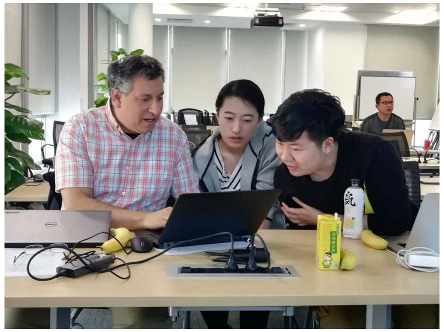
While we say that "the function 'add' is running," what we really mean is that "the code from 'add' is running."

Why the difference? Because we have to distinguish between two different ideas, and two different objects, that Python has created. The first object is a "function object," one that's fairly well understood. We explicitly create a new function object when we use "def"; as I'm often fond of saying, "def" does two things in Python: (1) It creates a new function object, and (2) it assigned that object to the variable named right after the keyword "def". We can also create new function objects with the "lambda" keyword.

The thing is, a function object is a blueprint, or a description, of what should happen when the function is invoked. You can think of it as the blueprints for a house; it describes what needs to be done, but is clearly distinct from the house itself. Moreover, the same blueprints can be used to create multiple homes -- and in the same way, the same function can be invoked multiple times.

But each time we invoke a function, Python creates a new object, one which we don't normally think about too much, but which is crucial to the language: A stack frame.

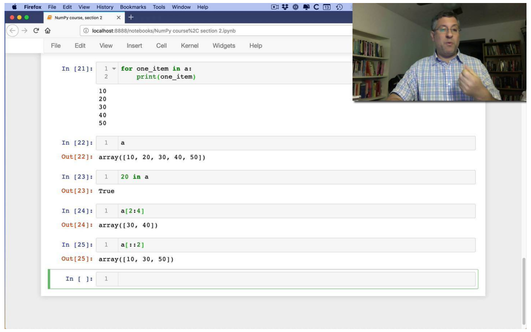
Corporate Python training



Weekly Python Exercise

Weekly Python Exercise

Online video courses



```
In [21]: for one_item in a:
             print(one_item)

         10
         20
         30
         40
         50

In [22]: a

Out[22]: array([10, 20, 30, 40, 50])

In [23]: 20 in a

Out[23]: True

In [24]: a[2:4]

Out[24]: array([30, 40])

In [25]: a[::2]

Out[25]: array([10, 30, 50])

In [ ]:
```

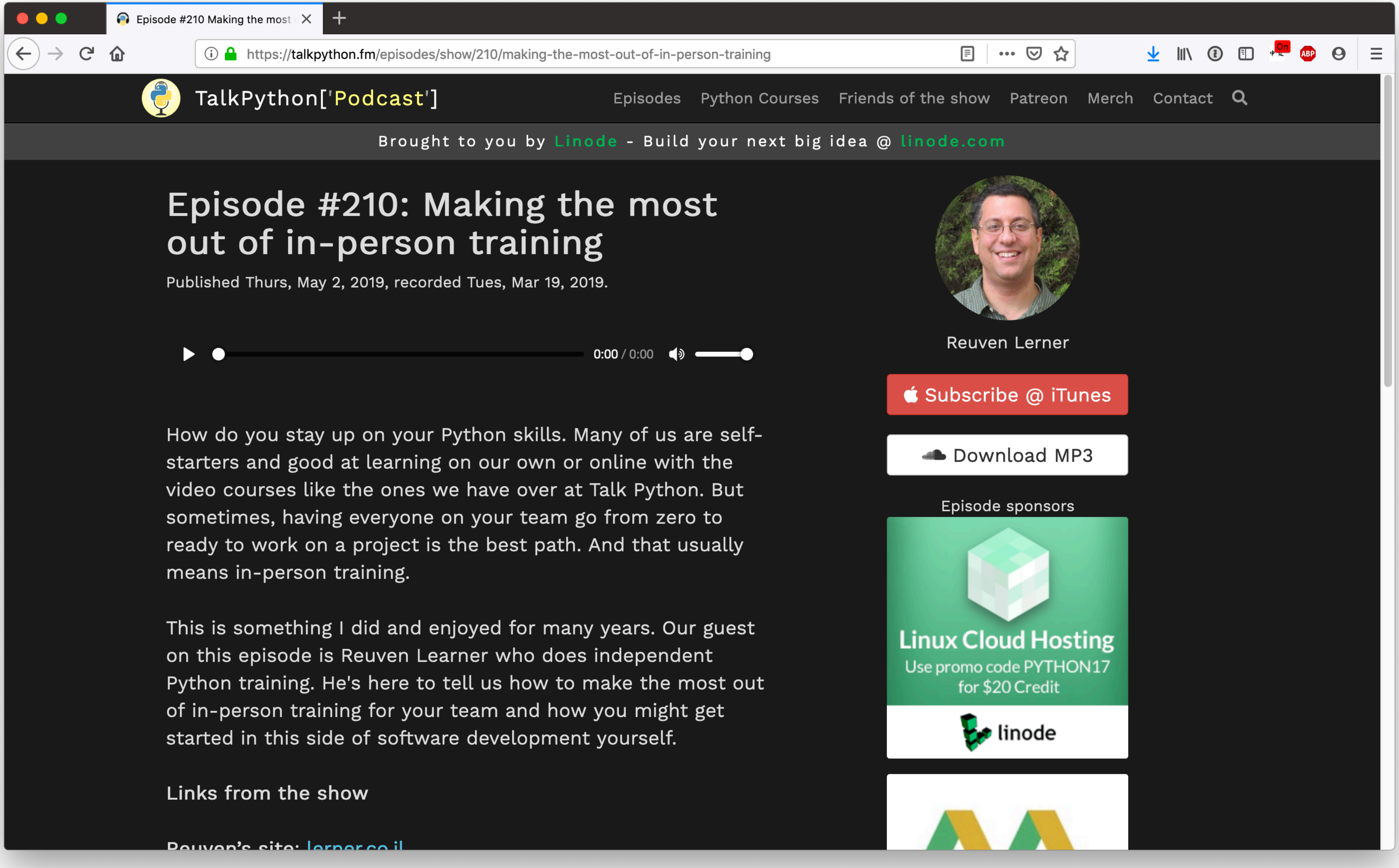
Solutions and discussions in text and video

Python Workout

Fifty short projects

Reuven M. Lerner

MANNING


Episode #210 Making the most
https://talkpython.fm/episodes/show/210/making-the-most-out-of-in-person-training
TalkPython['Podcast']
Episodes
Python Courses
Friends of the show
Patreon
Merch
Contact
Brought to you by Linode - Build your next big idea @ linode.com
0:00 / 0:00
Reuven Lerner
Subscribe @ iTunes
Download MP3
Episode sponsors
Linux Cloud Hosting
Use promo code PYTHON17 for $20 Credit
linode

Episode #210: Making the most out of in-person training

Published Thurs, May 2, 2019, recorded Tues, Mar 19, 2019.

How do you stay up on your Python skills. Many of us are self-starters and good at learning on our own or online with the video courses like the ones we have over at Talk Python. But sometimes, having everyone on your team go from zero to ready to work on a project is the best path. And that usually means in-person training.

This is something I did and enjoyed for many years. Our guest on this episode is Reuven Learner who does independent Python training. He's here to tell us how to make the most out of in-person training for your team and how you might get started in this side of software development yourself.

Links from the show

Reuven's site: lerner.co.il

Let’s decorate a function!

See this:

```
@mydeco
def add(a, b):
    return a + b
```

But think this:

```
def add(a, b):
    return a + b
add = mydeco(add)
```

Three callables!

(2) The decorator

(1) The decorated
function

```
@mydeco
def add(a, b):
    return a + b
```

(3) The return value
from mydeco(add),
assigned back to “add”

Defining a decorator

(2) The decorator

(1) The decorated function

```
def mydeco(func):
    def wrapper(*args, **kwargs):
        return f'{func(*args, **kwargs)}!!!'
    return wrapper
```

(3) The return value from mydeco(add), assigned back to “add”

Another perspective

```
def mydeco(func):
    def wrapper(*args, **kwargs):
        return f'{func(*args, **kwargs)}!!!'
    return wrapper
```

Executes once, when we decorate the function

Executes each time the decorated function runs

Wow, decorators are cool!

Better yet:
Decorators are useful

Example 1: Timing

How long does it take for a function to run?

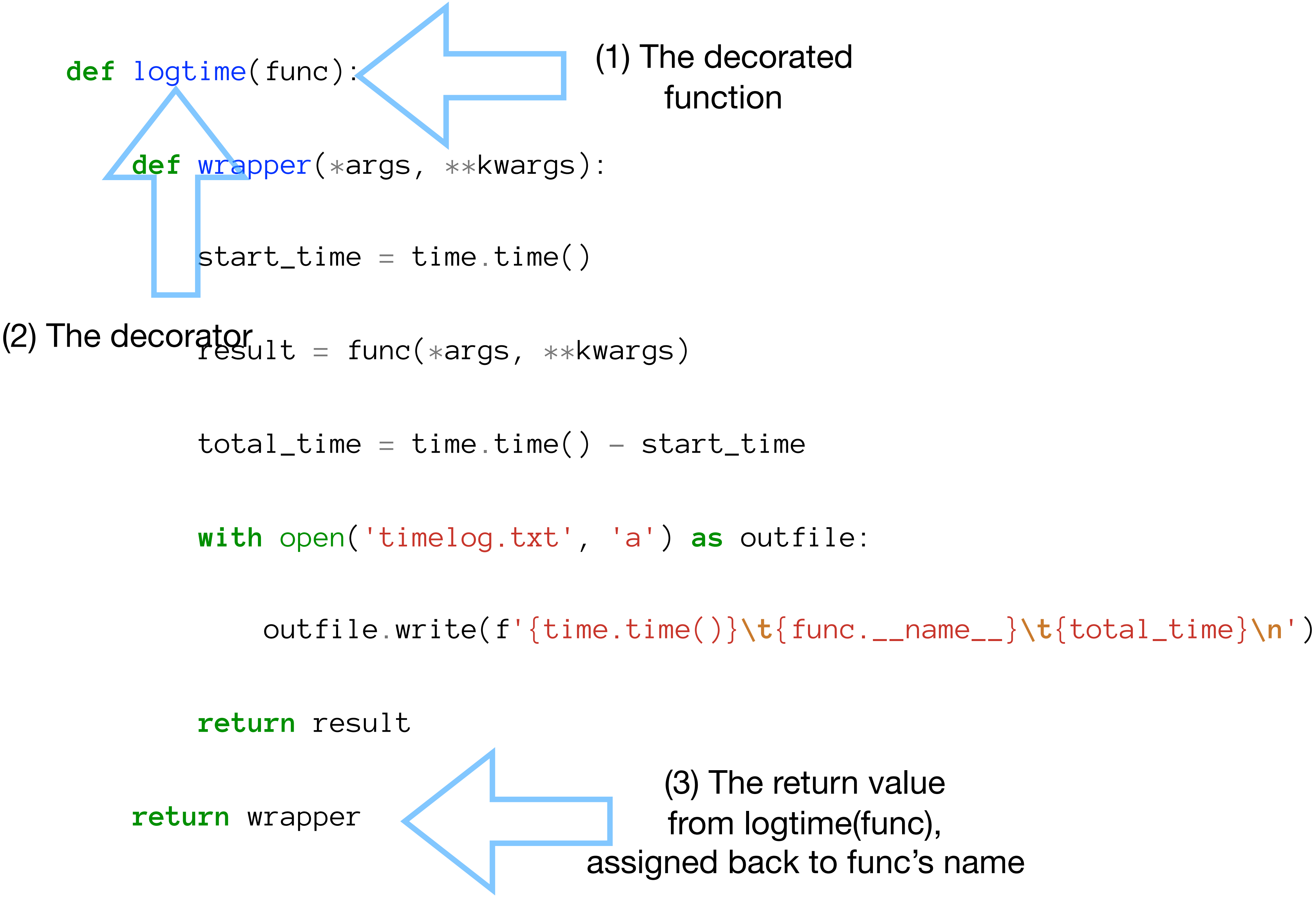
My plan

- The inner function (“wrapper”) will run the original function
- But it’ll keep track of the time before and after doing so
- Before returning the result to the user, we’ll write the timing information to a logfile

```python
def logtime(func):

    def wrapper(*args, **kwargs):
        start_time = time.time()

        result = func(*args, **kwargs)

        total_time = time.time() - start_time

        with open('timelog.txt', 'a') as outfile:

            outfile.write(f'{time.time()}\t{func.__name__}\t{total_time}\n')

        return result

    return wrapper
```

```python
@logtime
def slow_add(a, b):
    time.sleep(2)
    return a + b


@logtime
def slow_mul(a, b):
    time.sleep(3)
    return a * b
```

```
1556147289.666728  slow_add   2.00215220451355
1556147292.670324  slow_mul   3.0029208660125732
1556147294.6720388 slow_add   2.0013420581817627
1556147297.675552  slow_mul   3.0031981468200684
1556147299.679569  slow_add   2.00363206863403 3
1556147302.680939  slow_mul   3.0009829998016357
1556147304.682554  slow_add   2.001215934753418
```



```
def logtime(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        total_time = time.time() - start_time
        with open('timelog.txt', 'a') as outfile:
            outfile.write(f'{time.time()}\t{func.__name__}\t{total_time}\n')
        return result
    return wrapper
```

(1) The decorated
function

(2) The decorator

(3) The return value
from logtime(func),
assigned back to func’s name

Example 2: Once per min

Raise an exception if we try to run
a function more than once in 60 seconds

Limit

```
def once_per_minute(func):

    def wrapper(*args, **kwargs):

        # What goes here?


        return func(*args, **kwargs)

    return wrapper
```

(1) The decorated function

(2) The decorator

(3) The return value from once_per_minute(func), assigned back to func's name

We need “nonlocal”!

```
def once_per_minute(func):
    last_invoked = 0

    def wrapper(*args, **kwargs):
        nonlocal last_invoked
        elapsed_time = time.time() - last_invoked
        if elapsed_time < 60:
            raise CalledTooOftenError(f"Only {elapsed_time} has passed")
        last_invoked = time.time()
        return func(*args, **kwargs)
    return wrapper
```

We need “nonlocal”!

```
def once_per_minute(func):
    last_invoked = 0
```

Executes once, when we decorate the function

```
    def wrapper(*args, **kwargs):
        nonlocal last_invoked
        elapsed_time = time.time() - last_invoked
        if elapsed_time < 60:
            raise CalledTooOftenError(f"Only {elapsed_time} has passed")
        last_invoked = time.time()
        return func(*args, **kwargs)
    return wrapper
```

Executes each time the decorated function is executed

```python
print(add(2, 2))

print(add(3, 3))
```

```
4

__main__.CalledTooOftenError: Only 4.410743713378906e-05 has passed
```

Example 3: Once per n

Raise an exception if we try to run
a function more than once in n seconds

Remember

When we see this:

```python
@once_per_minute

def add(a, b):

    return a + b
```

We should think this:

```python
def add(a, b):

    return a + b

add = once_per_minute(add)
```

So what do we do now?

This code:

```
@once_per_n(5)

def add(a, b):

    return a + b
```

Becomes this:

```
def add(a, b):

    return a + b

add = once_per_n(5)(add)
```

That’s right: 4 callables!

```
def add(a, b):
    return a + b
add = once_per_n(5)(add)
```

(1) The decorated function

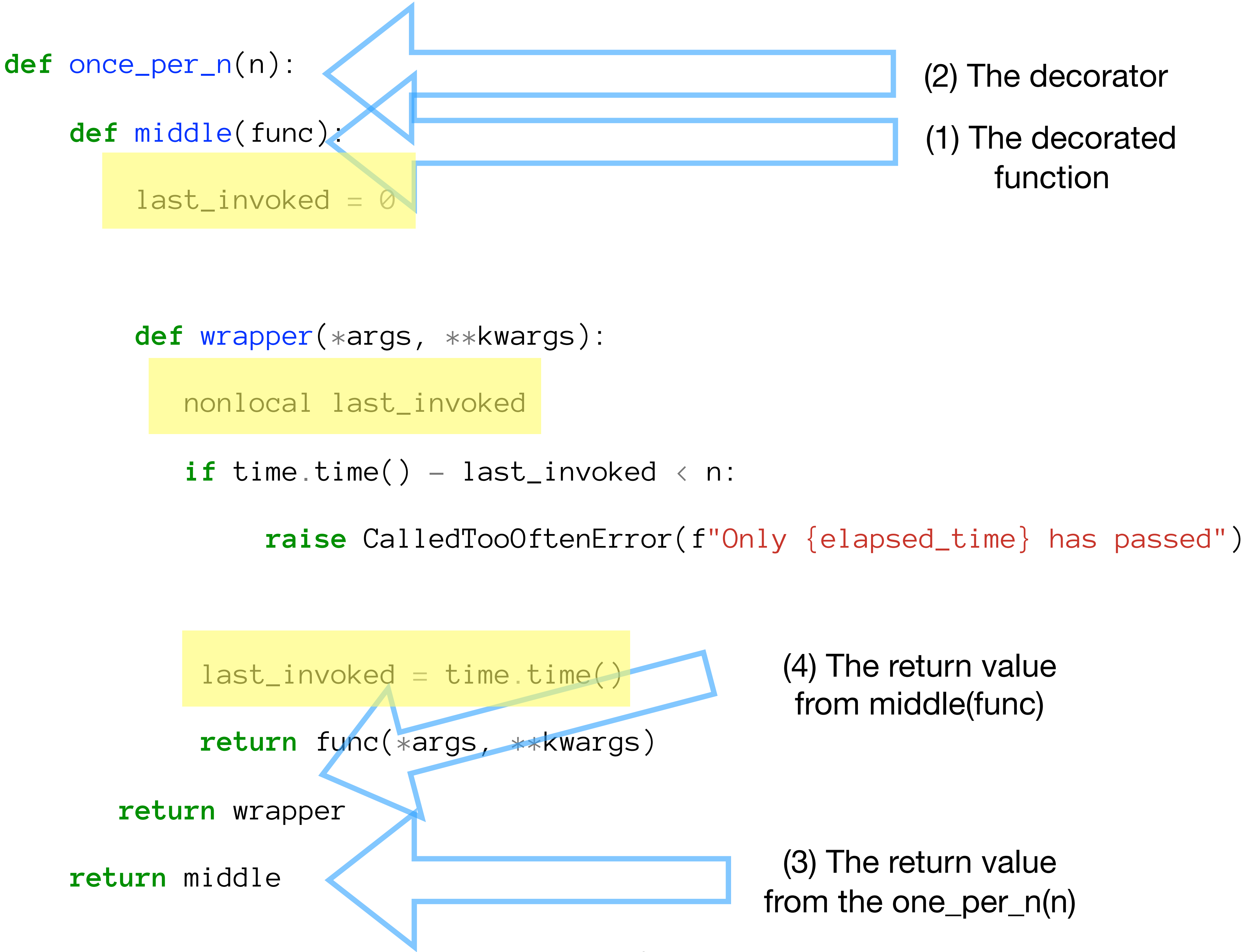
(2) The decorator

(3) The return value from once_per_n(5), itself a callable, invoked on “add”

(4) The return value from once_per_n(5)(add), assigned back to “add”

How does this look in code?

For four callables,
we need *three* levels of function!



```
def once_per_n(n):
    def middle(func):
        last_invoked = 0


        def wrapper(*args, **kwargs):
           nonlocal last_invoked
           if time.time() - last_invoked < n:
                raise CalledTooOftenError(f"Only {elapsed_time} has passed")

            last_invoked = time.time()
            return func(*args, **kwargs)
       return wrapper
    return middle
```

(2) The decorator

(1) The decorated
function

(4) The return value
from middle(func)

(3) The return value
from the one_per_n(n)

```
def once_per_n(n):
    def middle(func):
        last_invoked = 0
        def wrapper(*args, **kwargs):
            nonlocal last_invoked
            if time.time() - last_invoked < n:
                raise CalledTooOftenError(f"Only {elapsed_time} has passed")

             last_invoked = time.time()
             return func(*args, **kwargs)
        return wrapper
    return middle
```

**Executes once,
when we get an argument**

**Executes once,
when we decorate
the function**

**Executes each time
the function is run**

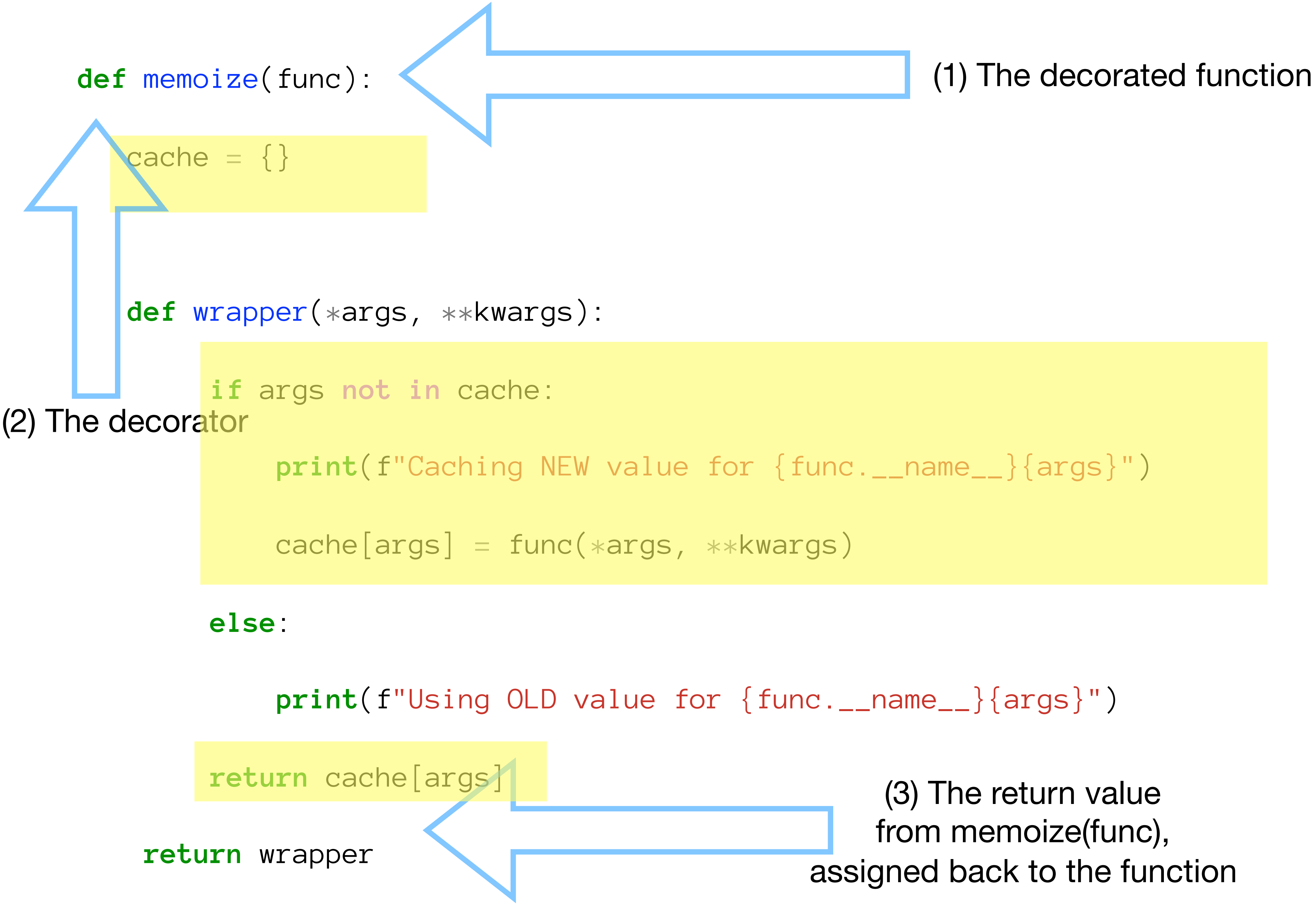
Does it work?

```
print(slow_add(2, 2))

print(slow_add(3, 3))
```

```
4

__main__.CalledTooOftenError: Only 3.0025641918182373 has passed
```

Example 4: Memoization

Cache the results of function calls,
so we don’t need to call them again



```python
def memoize(func):
    cache = {}

    def wrapper(*args, **kwargs):
        if args not in cache:
            print(f"Caching NEW value for {func.__name__}{args}")
            cache[args] = func(*args, **kwargs)
        else:
            print(f"Using OLD value for {func.__name__}{args}")
        return cache[args]
    return wrapper
```

(1) The decorated function

(2) The decorator

(3) The return value
from memoize(func),
assigned back to the function

Executes once, when we decorate the function

```
def memoize(func):
    cache = {}
```

Executes each time the decorated function is executed

```
    def wrapper(*args, **kwargs):
        if args not in cache:
            print(f"Caching NEW value for {func.__name__}{args}")
            cache[args] = func(*args, **kwargs)
        else:
            print(f"Using OLD value for {func.__name__}{args}")
        return cache[args]
    return wrapper
```

Does it work?

```
@memoize
def add(a, b):
    print("Running add!")
    return a + b


@memoize
def mul(a, b):
    print("Running mul!")
    return a * b
```

```python
print(add(3, 7))

print(mul(3, 7))

print(add(3, 7))

print(mul(3, 7))
```

```
Caching NEW value for add(3, 7)

Running add!

10

Caching NEW value for mul(3, 7)

Running mul!

21

Using OLD value for add(3, 7)

10

Using OLD value for mul(3, 7)

21
```

Wait a second…

- What if *args contains a non-hashable value?
- What about **kwargs?

Pickle to the rescue!

- Strings (and bytestrings) are hashable
- And just about anything can be pickled
- So use a tuple of bytestrings as your dict keys, and you’ll be fine for most purposes.
- If all this doesn’t work, you can always call the function!

```python
def memoize(func):
    cache = {}
    def wrapper(*args, **kwargs):
        t = (pickle.dumps(args),  pickle.dumps(kwargs))
        if t not in cache:
            print(f"Caching NEW value for {func.__name__}{args}")
            cache[t] = func(*args, **kwargs)
        else:
            print(f"Using OLD value for {func.__name__}{args}")
        return cache[t]
    return wrapper
```

Example 5: Attributes

**Give many objects the same attributes,
but without using inheritance**

Setting class attributes

- I want to have a bunch of attributes consistently set across several classes
- These classes aren’t related, so I no inheritance
- (And no, I don’t want multiple inheritance.)

Let’s improve __repr__

```python
def fancy_repr(self):
    return f"I'm a {type(self)}, with vars {vars(self)}"
```

Our implementation

(2) The decorator

(1) The decorated class

```
def better_repr(c):
    c.__repr__ = fancy_repr
    def wrapper(*args, **kwargs):
        o = c(*args, **kwargs)
        return o
    return wrapper
```

(3) Return a callable

Our 2nd implementation

(2) The decorator

```
def better_repr(c):
    c.__repr__ = fancy_repr
    return c
```

(1) The decorated class

(3) Return a callable —
here, it’s just the class!

Does it work?

```
@better_repr
class Foo():
    def __init__(self, x, y):
        self.x = x
        self.y = y
f = Foo(10, [10, 20, 30])
print(f)
```

```
I'm a Foo, with vars {'x': 10, 'y': [10, 20, 30]}
```

Wait a moment!
We set a class attribute.
Can we also change object attributes?

Of course.

Let’s give every object its own birthday

- The @object_birthday decorator, when applied to a class, will add a new _created_at attribute to new objects
- This will contain the timestamp at which each instance was created

Our implementation

(2) The decorator

(1) The decorated class

```
def object_birthday(c):
    def wrapper(*args, **kwargs):
        o = c(*args, **kwargs)
        o._created_at = time.time()
        return o
    return wrapper
```

(3) The returned object — what we get when we invoke a class, after all

Does it work?

```python
@object_birthday
class Foo():
    def __init__(self, x, y):
        self.x = x
        self.y = y

f = Foo(10, [10, 20, 30])
print(f)
print(f._created_at)
```

```
<__main__.Foo object at 0x106c82f98>
1556536616.5308428
```

Let’s do both!

```
def object_birthday(c):
    c.__repr__ = fancy_repr
    def wrapper(*args, **kwargs):
        o = c(*args, **kwargs)
        o._created_at = time.time()
        return o
    return wrapper
```

Add a method
to the class

Add an attribute
to the instance

Conclusions

- Decorators let you DRY up your callables
- Understanding how many callables are involved makes it easier to see what problems can be solved, and how
- Decorators make it dramatically easier to do many things
- Of course, much of this depends on the fact that in Python, callables (functions and classes) are objects like any other — and can be passed and returned easily.

Questions?

- Get the code + slides from this talk:
  - http://PracticalDecorators.com/
- Or: Chat with me at the WPE booth!
- Or contact me:
  - reuven@lerner.co.il
  - Twitter: @reuvenmlerner

Weekly Python Exercise