



Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

tracer

singleton

privateness

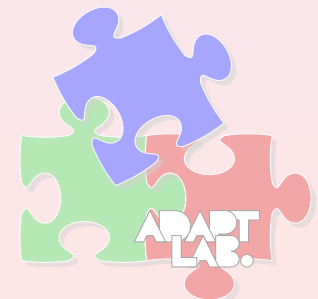
References

Decorators

How to Silently Extend Classes (Part 2)

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: [@w_cazzola](https://twitter.com/w_cazzola)





Class Extensions through Decorators

What's a Decorator?

Decorators

Walter Cazzola

Decorators

Definition

• decorators

class decorators

At Work

timing

tracer

singleton

privateness

References

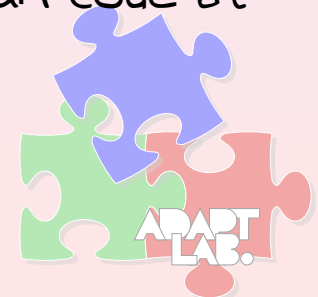
Decoration is a way to specify management code for functions and classes.

- Decorators themselves take the form of callable objects (e.g., functions) that process other callable objects.

Python decorators come in two related flavors:

- **Function decorators** do name rebinding at function definition time, providing a layer of logic that can manage functions and methods, or later calls to them.
- **Class decorators** do name rebinding at class definition time, providing a layer of logic that can manage classes, or the instances created by calling them later.

In short, decorators provide a way to automatically run code at the end of function and class definition statements.





Class Extensions through Decorators

Function Decorators

Decorators

Walter Cazzola

Decorators

Definition

↳ decorators

class decorators

At Work

timing

tracer

singleton

privateness

References

```
def decorator(F):          # on @ decoration
    def wrapper(*args):    # on wrapped function call
        # Use F and args and then call F(*args)
        print("I'm executing the call {0}{1} ...". \
              format(F.__name__, args))
        return F(*args)
    return wrapper
```

```
@decorator
def f(x,y):
    print("*** f({0}, {1})".format(x,y))
f(42, 7)
```

```
class wrapper:
    def __init__(self, func): # On @ decoration
        self.func = func
    def __call__(self, *args): # On wrapped calls
        # Use func and args and then call func(*args)
        print("I'm executing the call {0}{1} ...". \
              format(self.func.__name__, args))
        return self.func(*args)
```

```
@wrapper
def f2(x,y,z):
    print("*** f2({0}, {1}, {2})".format(x,y,z))
f2("abc",7, 'B')
```

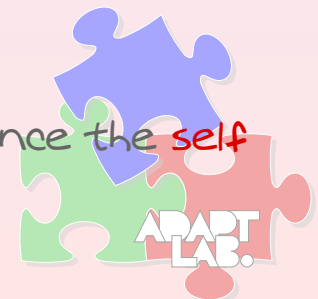
```
[23:30]cazzola@hymir:~/esercizi-pa>python3 fdecs.py
I'm executing the call f(42, 7) ...
*** f(42, 7)
```

```
[23:31]cazzola@hymir:~/esercizi-pa>python3 fdecs.py
I'm executing the call f2('abc', 7, 'B') ...
*** f2(abc, 7, B)
```

$@decorator\ f(5,7) \equiv decorator(f)(5,7)$

Note that,

- methods cannot be decorated by function decorators since the **self** would be associated to the decorator.





Class Extensions through Decorators

Class Decorators

Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

tracer

singleton

privateness

References

```
def decorator(cls):                                # On @ decoration
    class wrapper:
        def __init__(self, *args):                  # On instance creation
            print("I'm creating {0}{1} ...".format(cls.__name__, args))
            self.wrapped = cls(*args)
        def __getattr__(self, name):                 # On attribute fetch
            print("I'm fetching {0}.{1} ...".format(self.wrapped, name))
            return getattr(self.wrapped, name)
        def __setattr__(self, attribute, value):     # On attribute set
            print("I'm setting {0} to {1} ...".format(attribute, value))
            if attribute == 'wrapped':               # Allow my attrs
                self.__dict__[attribute] = value    # Avoid looping
            else:
                setattr(self.wrapped, attribute, value)
    return wrapper

@decorator
class C:                                            # C = decorator(C)
    def __init__(self, x, y): self.attr = 'spam'
    def f(self, a, b): print("*** f({0}, {1})".format(a,b))
```

```
[0:06]cazzola@hymir:~/esercizi-pa/decorators>python3
>>> from cdecorators import *
>>> x = C(6, 7)
I'm creating C(6, 7) ...
I'm setting wrapped to <cdecorators.C object at 0xb79eb26c> ...
>>> print(x.attr)
I'm fetching <cdecorators.C object at 0xb79eb26c>.attr ...
spam
>>> x.f(x.attr, 7)
I'm fetching <cdecorators.C object at 0xb79eb26c>.f ...
I'm fetching <cdecorators.C object at 0xb79eb26c>.attr ...
*** f(spam, 7)
```



Class Extensions through Decorators

Function Decorators at Work: Timing

Decorators

Walter Cazzola

Decorators

Definition

• decorators

class decorators

At Work

timing

tracer

singleton

privateness

References

```
import time

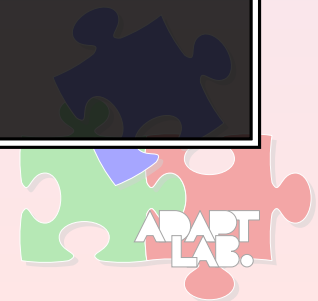
class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kwargs):
        start = time.clock()
        result = self.func(*args, **kwargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('{0}: {1:.5f}, {2:.5f}'.\
            format(self.func.__name__, elapsed, self.alltime))
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))

if __name__ == "__main__":
    result = listcomp(5)
    listcomp(50000)
    listcomp(500000)
    listcomp(1000000)
    print(result)
    print('allTime = {0}'.format(listcomp.alltime))
    print('')
    result = mapcall(5)
    mapcall(50000)
    mapcall(500000)
    mapcall(1000000)
    print(result)
    print('allTime = {0}'.format(mapcall.alltime))
    print('map/comp = {0}'.format(\
        round(mapcall.alltime / listcomp.alltime, 3)))
```

```
[21:06]cazzola@hymir:~/esercizi-pa>python3 timing.py
listcomp: 0.00000, 0.00000
listcomp: 0.03000, 0.03000
listcomp: 0.41000, 0.44000
listcomp: 0.85000, 1.29000
[0, 2, 4, 6, 8]
allTime = 1.29
mapcall: 0.00000, 0.00000
mapcall: 0.07000, 0.07000
mapcall: 0.71000, 0.78000
mapcall: 1.41000, 2.19000
[0, 2, 4, 6, 8]
allTime = 2.19
map/comp = 1.698
```





Class Extensions through Decorators

Class Decorators at Work: Tracer

Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

tracer

singleton

privateness

References

```
def Tracer(aClass):
    class Wrapper:
        def __init__(self, *args, **kwargs):
            self.fetches = 0
            self.wrapped = aClass(*args, **kwargs)
        def __getattr__(self, attrname):
            print('Trace: ' + attrname)
            self.fetches += 1
            return getattr(self.wrapped, attrname)
    return Wrapper

@Tracer
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate
```

On @ decorator
On instance creation
Use enclosing scope name
Catches all but own attrs
Delegate to wrapped obj
Person = Tracer(Person)
Wrapper remembers Person
Accesses outside class traced
In-method accesses not traced

```
[12:59]cazzola@hymir:~/esercizi-pa/decorators>python3 >>> print(sue.pay())
Trace: pay
6000
>>> from tracer import *
>>> bob = Person('Bob', 40, 50)
Trace: name
Bob
>>> print(bob.name) # bob is a Wrapper to a Person
Trace: name
Bob
>>> print(bob.pay())
Trace: pay
2000
>>> sue = Person('Sue', rate=100, hours=60)
Trace: name
Sue
>>> print(sue.name) # sue is a different Wrapper
Trace: name
Sue
>>> print(bob.name) # bob has a different state
Trace: name
Bob
>>> print(bob.pay())
Trace: pay
2000
>>> print([bob.fetches, sue.fetches])
[4, 2]
```



Class Extensions through Decorators

Class Decorators at Work: Singleton

Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

tracer

singleton

privateness

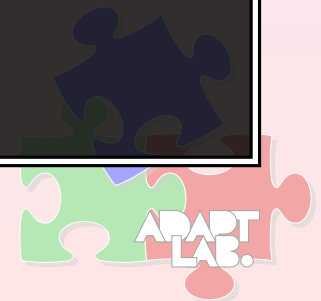
References

```
class singleton:
    def __init__(self, aClass):
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args):
        if self.instance == None:
            self.instance = self.aClass(*args) # One instance per class
        return self.instance

@singleton
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton
class Spam:
    def __init__(self, val):
        self.attr = val
```

```
[21:29]cazzola@hymir:~/esercizi-pa/decorators>python3
>>> sue = Person('Sue', 50, 20)
>>> from singleton import *
>>> bob = Person('Bob', 40, 10)
>>> print(bob.name, bob.pay())
Bob 400
>>> X = Spam(42)
>>> Y = Spam(99)
>>> print(X.attr, Y.attr)
42 42
```





Class Extensions through Decorators

Class Decorators at Work: Privateness

Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

tracer

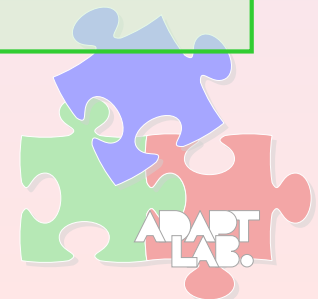
singleton

privateness

References

```
traceMe = False
def trace(*args):
    if traceMe: print('[ ' + ' '.join(map(str, args)) + ' ]')
def Private(*privates):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kwargs):
                self.wrapped = aClass(*args, **kwargs)
            def __getattr__(self, attr):
                trace('get:', attr)
                if attr in privates:
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.wrapped, attr)
            def __setattr__(self, attr, value):
                trace('set:', attr, value)
                if attr == 'wrapped':
                    self.__dict__[attr] = value
                elif attr in privates:
                    raise TypeError('private attribute change: ' + attr)
                else:
                    setattr(self.wrapped, attr, value)
        return onInstance
    return onDecorator
```

privates in enclosing scope
aClass in enclosing scope
wrapped in instance attribute
My attrs don't call getattr
Others assumed in wrapped
Outside accesses
Others run normally
Allow my attrs
Avoid looping
Wrapped obj attrs
Or use __dict__





Class Extensions through Decorators

Class Decorators at Work: Privateness (Cont'd)

Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

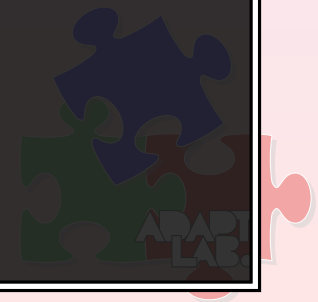
tracer

singleton

privateness

References

```
[22:05]cazzola@hymir:~/esercizi-pa/decorators>python3
>>> from private import *
>>> traceMe = True
>>> @Private('data', 'size')
... class Doubler:
...     def __init__(self, label, start):
...         self.label = label                # Accesses inside the subject class
...         self.data = start                # Not intercepted: run normally
...     def size(self):
...         return len(self.data)             # Methods run with no checking
...     def double(self):                     # Because privacy not inherited
...         for i in range(self.size()):
...             self.data[i] = self.data[i] * 2
...     def display(self):
...         print('{0} => {1}'.format(self.label, self.data))
>>> X = Doubler('X', [1, 2, 3])
>>> print(X.label)                          # Accesses outside subject class
X
>>> X.display(); X.double(); X.display()      # Intercepted: validated, delegated
X => [1, 2, 3]
X => [2, 4, 6]
>>> print(X.size())                         # prints "TypeError: private attribute fetch: size"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "private.py", line 19, in __getattr__
    raise TypeError('private attribute fetch: ' + attr)
TypeError: private attribute fetch: size
>>> X.data = [1, 1, 1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "private.py", line 27, in __setattr__
    raise TypeError('private attribute change: ' + attr)
TypeError: private attribute change: data
```





References

Decorators

Walter Cazzola

Decorators

Definition

f. decorators

class decorators

At Work

timing

tracer

singleton

privateness

References

- ▶ Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
Practical Programming: An Introduction to Computer Science Using Python.
The Pragmatic Bookshelf, second edition, 2009.
- ▶ Mark Lutz.
Learning Python.
O'Reilly, fourth edition, November 2009.
- ▶ Mark Pilgrim.
Dive into Python 3.
Apress*, 2009.
- ▶ Mark Summerfield.
Programming in Python 3: A Complete Introduction to the Python Language.
Addison-Wesley, October 2009.

