Slide 1

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

Slide 2

Class Extensions through Decorators
What's a Decorator?

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 <u>automatically</u> run code at the end of function and class definition statements.

Slide 3

Class Extensions through Decorators
Function Decorators

```python
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)
```

```python
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, 'ß')
```

```
[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, 'ß') ...
*** f2(abc, 7, ß)
```

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

Note that,

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

Slide 4

Class Extensions through Decorators
Class Decorators

```python
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)
```

Decorators

Walter Cazzola

Decorators
Definition
& decorators
class decorators
At Work
timing
tracer
singleton
privateness
References

```python
import time
class timer:
  def __init__(self, func):
    self.func    = func
    self.alltime = 0
  def __call__(self, *args, **kargs):
    start   = time.clock()
    result = self.func(*args, **kargs)
    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      mapcall: 0.00000, 0.00000
listcomp: 0.00000, 0.00000                                mapcall: 0.07000, 0.07000
listcomp: 0.03000, 0.03000                                mapcall: 0.71000, 0.78000
listcomp: 0.41000, 0.44000                                mapcall: 1.41000, 2.19000
listcomp: 0.85000, 1.29000                                [0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]                                           allTime = 2.19
allTime = 1.29                                            map/comp = 1.698
```

Decorators

Walter Cazzola

Decorators
Definition
& decorators
class decorators
At Work
timing
tracer
singleton
privateness
References

```python
def Tracer(aClass):                                    # On @ decorator
  class Wrapper:
    def __init__(self, *args, **kargs):                # On instance creation
      self.fetches = 0
      self.wrapped = aClass(*args, **kargs)            # Use enclosing scope name
    def __getattr__(self, attrname):
      print('Trace: ' + attrname)                      # Catches all but own attrs
      self.fetches += 1
      return getattr(self.wrapped, attrname)           # Delegate to wrapped obj
  return Wrapper

@Tracer
class Person:                                          # Person = Tracer(Person)
  def __init__(self, name, hours, rate):               # Wrapper remembers Person
    self.name = name
    self.hours = hours
    self.rate = rate
  def pay(self):                                       # Accesses outside class traced
    return self.hours * self.rate                      # In-method accesses not traced
```

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

Decorators

Walter Cazzola

Decorators
Definition
& decorators
class decorators
At Work
timing
tracer
singleton
privateness
References

```python
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                                    # Person = singleton(Person)
class Person:                                  # Rebinds Person to onCall
    def __init__(self, name, hours, rate): # onCall remembers Person
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton                                    # Spam = singleton(Spam)
class Spam:                                    # Rebinds Spam to onCall
    def __init__(self, val):                   # onCall remembers Spam
        self.attr = val
```

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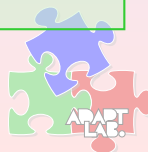

Decorators

Walter Cazzola

Decorators
Definition
& decorators
class decorators
At Work
timing
tracer
singleton
privateness
References

```python
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')
def Private(*privates):                                  # privates in enclosing scope
    def onDecorator(aClass):                             # aClass in enclosing scope
        class onInstance:                                # wrapped in instance attribute
            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)
            def __getattr__(self, attr):                 # My attrs don't call getattr
                trace('get:', attr)                      # Others assumed in wrapped
                if attr in privates:
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.wrapped, attr)
            def __setattr__(self, attr, value):          # Outside accesses
                trace('set:', attr, value)               # Others run normally
                if attr == 'wrapped':                    # Allow my attrs
                    self.__dict__[attr] = value          # Avoid looping
                elif attr in privates:
                    raise TypeError('private attribute change: ' + attr)
                else:
                    setattr(self.wrapped, attr, value)   # Wrapped obj attrs
        return onInstance                                # Or use __dict__
    return onDecorator
```

```
[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
```
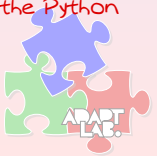
▶ Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
Practical Programming: An Introduction to Computer Science Using Python.
The Pragmatic Bookshelf, second edition, 2009.

▶ 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.