Programmazione Avanzata

Design Pattern: Decorator (II parte)

Lezione VIII

30

Class Decorator

A differenza del codice in classdec2.py qui Spam.numInstances viene incrementata anche quando creiamo un'istanza di una delle sue sottoclassi. Perché?

classdec3.py

```
def count(aClass):
   aClass.numInstances = 0
   oldInit=aClass.__init__
   def __newInit__(self,*args,**kwargs):
      aClass.numInstances+=1
      oldInit(self,*args,**kwargs)
   aClass.__init__=__newInit__
   return aClass

@count
class Spam:
   pass
@count
class Sub(Spam):
   pass
@count
class Other(Spam):
   pass
```

```
>>> from classdec3.py import Spam, Sub, Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
3
>>> print(sub.numInstances)
1
>>> print(other.numInstances)
1
>>> other=Other()
>>> print(other.numInstances)
2
>>> print(spam.numInstances)
4
>>> print(sub.numInstances)
1
```


31

Class Decorator

```
def count(aClass):
   aClass.numInstances = 0
   oldInit=aClass.__init__
   def __newInit__(self,*args,**kwargs):
      aClass.numInstances+=1
      oldInit(self,*args,**kwargs)
   aClass.__init__=__newInit__
   return aClass

@count
class Spam:
   pass
@count
class Sub(Spam):
   pass
@count
class Other(Spam):
   pass
```

A differenza del codice in classdec2.py qui Spam.numInstances viene incrementata anche quando creiamo un'istanza di una delle sue sottoclassi. Perché?

Risposta:
Qui ogni classe ha la sua variabile numInstances e il suo metodo __init__, entrambi "attaccati" dal decorator count
__init__ di Sub e Other invocano oldInit che e` di fatto __init__ della classe spam gia` decorata e cioe` e` newInit.
__init__ di spam prima della decorazione e` quella di object.

→

- spam=Spam() viene eseguito __init__ di spam (__newInit__ di spam) che incrementa numInstances di spam e poi invoca __init__ di object
- Sub=Sub() viene eseguito prima __init__ di Sub che incrementa numInstances di Sub e poi invoca oldinit che e` __init__ di Spam (nella versione gia` decorate) che incrementa numInstances di Spam e poi invoca oldinit che e` __init__ di object.
- Stesso discorso per other=Other()

Alcune considerazioni sul codice nelle due slide precedenti

- Nei due ultimi esempi, count pone oldInit=aClass.__init__ e poi definisce la funzione __newInit__ in modo che invochi oldInit e non aClass.__init__ .
- Se __newInit__ avesse invocato aClass.__init__ allora, nel momento in cui avessimo creato un'istanza di una delle classi decorate con count, il metodo __init__ della classe (rimpiazzato nel frattempo da __newInit__) avrebbe lanciato l'eccezione RecursionError.
  - Questa eccezione indica che è stato ecceduto il limite al numero massimo di chiamate ricorsive possibili.
  - Questo limite evita un overflow dello stack e un conseguente crash di Python
- L'eccezione sarebbe stata causata da una ricorsione infinita innescata dall'invocazione di aClass.__init__ all'interno di __newInit__.
  - A causa del late binding, il valore di aClass.__init__ nella chiusura di __newInit__ è stabilito quando __newInit__ è eseguita. Siccome quando si esegue __newInit__ si ha che aClass.__init__ è stato sostituito dal metodo __newInit__ allora __newInit__ avrebbe invocato ricorsivamente se stesso.

Class decorator: Esercizio

Scrivere un decoratore di classe myDecorator che dota la classe decorata di un **metodo di istanza** contaVarClasse che prende in input un tipo t e restituisce il numero **di variabili di classe** di tipo t della classe.

34

Late binding

```
1.  listOfFunctions=[]
2.  for m in [1, 2, 3]:
3.      def f(n):
4.              return m*n
5.      listOfFunctions.append(f)

6.  for function in listOfFunctions:
7.      print(function (4))
```

Inaspettatamente il for alle linee 6 e 7 stampa
12
12
12
e non
4
8
12

Questo perché ciascuna funzione aggiunta alla lista computa m*n ed m assume come ultimo valore 3. Di conseguenza la funzione calcola sempre 3*n.

35

Late binding

- Late binding: in Python i valori delle variabili usati nelle funzioni vengono osservati al momento della chiamata alla funzione.
  - Nell'esempio di prima quando vengono invocate le funzioni inserite in listOfFunctions, il valore di m è 3 perché il for (linee 2-4) è già terminato e il valore di m al termine del ciclo è 3

Chiusura

- Nella programmazione funzionale il termine chiusura indica la capacità di una funzione di ricordare valori presenti negli scope in cui essa è racchiusa a prescindere dal fatto che lo scope sia presente o meno in memoria quando la funzione è invocata.
- Scope delle funioni innestate (annidate):
  - Una funzione innestata è definita all'interno di un altra funzione.
  - Una funzione innestata puo` accedere allo scope della funzione che la racchiude, detto non-local scope.
    - Per default queste variabili sono di sola lettura e per modificarle occorle dichiararle non-local con la keyword nonlocal.
  - Una funzione **inner** definita all'interno di una funzione **outer** "ricorda" un valore dello scope di **outer** anche quando la variabile scompare dallo scope o la funzione **outer** viene rimossa dal namespace.

Chiusura

```
x = 24
y = 33
def outer():
  z = 100
  def inner():
      nonlocal z
      print("il valore di z stampato da inner è:", z)
      z=5
          def innerinner():
              print("il valore di z stampato da innerinner è ", z)
      return innerinner
  return inner

f=outer()       //f e` la funzione inner restituita da outer
g=f()           //g e` la funzione innerinner restituita da  f
g()            //stampa "il valore…"
```

```
l valore di z stampato da inner è: 100
il valore di z stampato da innerinner è  5
```


Chiusura

```
x = 24
y = 33
def outer():
  z = 100
  def inner():
      nonlocal z
      print("il valore di z stampato da inner è:", z)
      z=5
      def innerinner():
        print("il valore di z stampato da innerinner è ", z)
      return innerinner
      del z
  return inner

f=outer()
g=f()
del f
g()
```

```
l valore di z stampato da inner è: 100
il valore di z stampato da innerinner è  5
```

Chiusura e late binding

```
x = 24
y = 33
def outer():
  z = 100
  def inner():
      nonlocal z
      print("il valore di z stampato da inner è:", z)
      z=5
      def innerinner1():
        print("il valore di z stampato da innerinner1 è ", z)
      z=10
      def innerinner2():
        print("il valore di z stampato da innerinner2 è ", z)
      return (innerinner1,innerinner2)
  return inner
f=outer()  #questa e` la funzione inner
g=f()  #questa e` la tupla delle due funzioni innerinner1 e innerinner2
g[0]()
g[1]()
```

```
l valore di z stampato da inner è: 100
il valore di z stampato da innerinner1 è  10
il valore di z stampato da innerinner2 è  10
```


Proprietà

- Per capire il prossimo esempio di class decorator occorre parlare degli attributi property
- La funzione built-in property permette di associare operazioni di fetch e set ad attributi specifici
- property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*) restituisce un attributo property
  - *fget* è una funzione per ottenere il valore di un attributo
  - *fset* è una funzione per settare un attributo
  - *fdel* è una funzione per cancellare un attributo
  - *doc* crea una docstring dell'attributo.

Proprietà

- Se c è un'istanza di C, c.x =value invocherà il setter setx e del c.x invocherà il deleter delx.
- Se fornita, doc sarà la docstring dell'attributo property. In caso contrario, viene copiata la docstring di *fget* (se esiste)

```
class C:
    def __init__(self):
        self._x = None
    def getx(self): return self._x
    def setx(self, value): self._x = value
    def delx(self): del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```


42

Proprietà

- Nella classe Parrot in basso usiamo il decoratore @property per trasformare il metodo voltage() in un "getter" per l'attributo **read-only** voltage e settare la docstring di voltage a "Get the current voltage."

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```


43

Proprietà

- Un oggetto property ha i metodi getter, setter e deleter che possono essere usati come decoratori per creare una copia della proprietà con la corrispondente funzione accessoria uguale alla funzione decorata
- Questi due codici sono equivalenti
  - nel codice a sinistra dobbiamo stare attenti a dare alla funzioni aggiuntive lo stesso nome della proprietà originale (x, nel nostro esempio).

```
class C:
        def __init__(self):
                self._x = None
        @property
        def x(self):
                """I'm the 'x' property."""
                return self._x
        @x.setter
        def x(self, value):
                self._x = value
        @x.deleter
        def x(self):
                del self._x
```

```
class C:
        def __init__(self):
            self._x = None
        def getx(self): return self._x
        def setx(self, value): self._x = value
        def delx(self): del self._x

        x = property(getx, setx, delx, "I'm the 'x' property.")
```


Class Decorator

- È abbastanza comune creare classi che hanno molte proprietà read-write. Tali classi hanno molto codice duplicato o parzialmente duplicato per i getter e i setter.
- Esempio: Una classe Book che mantiene il titolo del libro, lo ISBN, il prezzo, e la quantità. Vorremmo
  - quattro decoratori @property, tutti fondamentalmente con lo stesso codice (ad esempio, @property def title(self): return title).
  - quattro metodi setter il cui codice differirebbe solo in parte
- I decoratori di classe consentono di evitare la duplicazione del codice

Class Decorator

```
@ensure("title", is_non_empty_str)
@ensure("isbn", is_valid_isbn)
@ensure("price", is_in_range(1, 10000))
@ensure("quantity", is_in_range(0, 1000000))
class Book:

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

self.title, self.isbn, self.price, self.quantity sono proprietà per cui gli assegnamenti che avvengono in __init__() sono tutti effettuati dai setter delle proprietà

Invece di scrivere il codice per creare le proprietà con i loro getter e setter, si usa un decoratore di classe

La funzione ensure() è un **decorator factory,** cioè una funzione che restituisce un decoratore. La funzione ensure() accetta due parametri, il nome di una proprietà e una funzione di validazione, e restituisce un decoratore di classe

Nel codice applico 4 volte @ensure per creare le 4 proprietà in questo ordine: quantity, price, isbn, title

46

Class Decorator

- Possiamo applicare i decoratori anche nel modo illustrato in figura.
- In questo modo è più evidente l'ordine in cui vengono applicati i decoratori.
- Lo statement class Book deve essere eseguito per primo perché la classe Book serve come parametro di ensure("quantity",...).
- La classe ottenuta applicando il decoratore restituito da ensure("quantity",...) è passata come argomento in ensure("price",...) e così via.

```
ensure("title", is_non_empty_str)( # Pseudo-code
    ensure("isbn", is_valid_isbn)(
        ensure("price", is_in_range(1, 10000))(
            ensure("quantity", is_in_range(0, 1000000))(class Book: ...))))
```


47

Class Decorator

- La funzione ensure() è parametrizzata dal nome della proprietà (name), dalla funzione di validazione (validate) e da una docstring opzionale (doc).
- ensure() crea un decoratore di classe che se applicato ad una classe, dota quella classe della proprietà il cui nome è specificato dal primo parametro di ensure()

```
def ensure(name, validate, doc=None):
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
        return Class
    return decorator
```

la funzione decorator()

- riceve una classe come unico argomento e crea un nome "privato" e lo assegna privateName;
- crea una funzione getter che restituisce il valore associato alla property;
- crea una funzione setter che, nel caso in cui validate() non lanci un'eccezione, modifica il valore della property con il nuovo valore value, eventualmente creando l'attributo property se non esiste

48

Class Decorator

- Una volta che sono stati creati getter e setter, essi vengono usati per creare una nuova proprietà che viene aggiunta come attributo alla classe passata come argomento a decorator().
- La proprietà viene creata invocando property() nell'istruzione evidenziata:
  - in questa istruzione viene invocata la funzione built-in setattr() per associare la proprietà alla classe
  - La proprietà così creata avrà nella classe il nome *pubblico* corrispondente al parametro name di ensure()

```
def ensure(name, validate, doc=None):
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
        return Class
    return decorator
```

49

Class Decorator

- Qualche considerazione sulle funzioni di validazione:
- la funzione di validazione is_in_range() usata per price e per quantity è una factory function che restituisce una nuova funzione is_in_range() che ha i valori minimo e massimo codificati al suo interno e prende in input il nome dell'attributo e un valore

```
def is_in_range(minimum=None, maximum=None):
    assert minimum is not None or maximum is not None
    def is_in_range(name, value):
        if not isinstance(value, numbers.Number):
            raise ValueError("{} must be a number".format(name))
        if minimum is not None and value < minimum:
            raise ValueError("{} {} is too small".format(name, value))
        if maximum is not None and value > maximum:
            raise ValueError("{} {} is too big".format(name, value))
    return is_in_range
```

- AssertionError se minimum o maximum sono entrambi None
- ValueError se value non è un numero, se minimum è diverso da None e value < minimum, oppure se maximum è diverso da None e value>maximum

50

Class Decorator

- Questa funzione di validazione è usata per la proprietà title e ci assicura che il titolo sia una stringa e che la stringa non sia vuota.
  - Il nome di una proprietà è utile nei messaggi di errore: nell'esempio viene sollevata l'eccezione ValueError se name non è una stringa o se è una stringa vuota e il nome della proprietà compare nel messaggio di errore.

```
def is_non_empty_str(name, value):
    if not isinstance(value, str):
        raise ValueError("{} must be of type str".format(name))
    if not bool(value):
        raise ValueError("{} may not be empty".format(name))
```


51