Programmazione Avanzata

Design Pattern: Decorator (III parte), Singleton, Proxy (I parte)

Lezione VIII

Class Decorator

Una modifica per applicare un unico decoratore di classe

```
@do_ensure
class Book:

    title = Ensure(is_non_empty_str)
    isbn = Ensure(is_valid_isbn)
    price = Ensure(is_in_range(1, 10000))
    quantity = Ensure(is_in_range(0, 1000000))

    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
```

- Applicare molti decoratori in sequenza è una pratica che non è accettata da tutti i programmatori
- In questo esempio, le 4 proprietà vengono create come istanze della classe Ensure
- __init__ della classe Book associa le proprietà all'istanza di Book creata
- il decoratore di classe @do_ensure rimpiazza ciascuna delle 4 istanze di Ensure con una proprietà con lo stesso nome della corrispondente istanza di Ensure. La proprietà avrà come funzione di validazione quella passata ad Ensure()

Class Decorator

- La classe Ensure è usata per memorizzare
  - la funzione di validazione che sarà usata dal setter della proprietà
  - l'eventuale docstring della proprietà
- Ad esempio, l'attributo title di Book è inizialmente creato come un'istanza di Ensure ma dopo la creazione della classe Book il decoratore @do_ensure rimpiazza ogni istanza di Ensure con una proprietà. Il setter usa la funzione di validazione con cui l'istanza è sta creata.

```
class Ensure:

    def __init__(self, validate, doc=None):
        self.validate = validate
        self.doc = doc
```


Class Decorator

- Il decoratore di classe do_ensure consiste di tre parti:
  - La prima parte definisce la funzione innestata make_property(). La funzione make_property() prende come parametro name (ad esempio, title) e un attributo di tipo Ensure e crea una proprietà il cui valore viene memorizzato in un attributo privato (ad esempio, "_title"). Il setter al suo interno invoca la funzione di validazione.

```
def do_ensure(Class):
    def make_property(name, attribute):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            attribute.validate(name, value)
            setattr(self, privateName, value)
        return property(getter, setter, doc=attribute.doc)
    for name, attribute in Class.__dict__.items():
        if isinstance(attribute, Ensure):
            setattr(Class, name, make_property(name, attribute))
    return Class
```

Class Decorator

- La seconda parte itera sugli attributi della classe e rimpiazza ciascun attributo di tipo Ensure con una nuova proprietà con lo stesso nome dell'attributo rimpiazzato.
- La terza parte restituisce la classe modificata

```
def do_ensure(Class):
    def make_property(name, attribute):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            attribute.validate(name, value)
            setattr(self, privateName, value)
        return property(getter, setter, doc=attribute.doc)
    for name, attribute in Class.__dict__.items():
        if isinstance(attribute, Ensure):
            setattr(Class, name, make_property(name, attribute))
    return Class
```


56

Class Decorator

- In teoria avremmo potuto evitare la funzione innestata e porre il codice di quella funzione dopo il test isinstance().
- Ciò non avrebbe però funzionato in pratica a causa di problemi con il binding ritardato.
- Questo problema si presenta abbastanza frequentemente quando si creano decoratori o decorator factory.
  - In genere per risolvere il problema è sufficiente usare una funzione separata (eventualmente innestata)

57

Class Decorator nella derivazioni di classi

- A volte creiamo una classe di base con metodi o dati al solo scopo di poterla derivare più volte.
- Ciò evita di dover duplicare i metodi o i dati nelle sottoclassi ma se i metodi o i dati ereditati non vengono mai modificati nelle sottoclassi, è possibile usare un decoratore di classe per raggiungere lo stesso obiettivo.

Class Decorator nella derivazioni di classi

- Questa è la classe base che verrà estesa da classi che non modificano il metodo on_change() e l'attributo mediator.

```
class Mediated:
    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```

Class Decorator nella derivazioni di classi

```
def mediated(Class):
    setattr(Class, "mediator", None)
    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
    setattr(Class, "on_change", on_change)
    return Class
```

Possiamo applicare il decoratore di classe mediated in questo modo:

@mediated
class Button: ...

La classe Button avrà esattamente lo stesso comportamento che avrebbe avuto se l'avessimo definita come sottoclasse di Mediated con

class Button(Mediated): ...

60

Class decorator: esercizio

- Scrivere un decoratore di classe che, se applicato ad una classe, la modifica in modo che funzioni come se fosse stata derivata dalla seguente classe base. N.B. le classi derivate da ClasseBase non hanno bisogno di modificare i metodi f() e g() e la variabile varC. Inoltre quando vengono create le istanze di una classe derivata queste ''nascono'' con lo stesso valore di varI settato da __init__ di ClasseBase.

```
class ClasseBase:
        varC=1000
        def __init__(self):
                self.varI=10
        def f(self,v):
                print(v*self.varI)
        @staticmethod
        def g(x):
                print(x*varC)
```


61

Il pattern Singleton

- Il pattern Singleton è un pattern **creazionale** ed è usato quando abbiamo bisogno di una classe che ha un'unica istanza che è la sola ad essere utilizzata dal programma.
- In particolare, è utile nelle seguenti situazioni:
  - Controllare l'accesso concorrente ad una risorsa condivisa
  - Se si ha bisogno di un punto globale di accesso per la risorsa da parti differenti del sistema.
  - Quando si ha bisogno di un unico oggetto di una certa classe

Il pattern Singleton

Alcuni usi comuni:

- Lo spooler della stampante: vogliamo una singola istanza dello spooler per evitare il conflitto tra richieste per la stessa risorsa
- Gestire la connessione ad un database
- Trovare e memorizzare informazioni su un file di configurazione esterno

Il pattern Singleton

- Il pattern Singleton è usato quando abbiamo bisogno di una classe che ha un'unica istanza che è la sola ad essere utilizzata dal programma
- In python creare un singleton è un' operazione molto semplice
- Il Python Cookbook (trasferito presso GitHub.com/activestate/code) fornisce
  - una classe Singleton di facile uso. Ogni classe che discende da essa diventa un singleton
  - una classe Borg che ottiene la stessa cosa in modo differente

Il pattern Singleton: la classe Singleton

- L'implementazione della classe è in realta` contenuta nella classe __impl
- la variabile __instance fara` riferimento all'unica istanza della classe Singleton che di fatto da un punto di vista implementativo sara` un'istanza della classe __impl

```
class Singleton:

           class __impl:
               """ Implementation of the singleton interface """

               def spam(self):
                   """ Test method, return singleton id """
                   return id(self)


           # storage for the instance reference
           __instance = None
```

Il pattern Singleton: la classe Singleton

- Quando viene creata un'istanza di Singleton, __init__() verifica che non esista gia` un'istanza andando a controllare che __instance sia None.
- Se non esiste gia` un'istanza questa viene creata. Nell'implementazione viene di fatto creata un'istanza di __impl alla quale si accede attrraverso la variabile Singleton.__instance.
- In __dict__ della ''vera'' istanza di Singleton si aggiunge l'attributo _Singleton_instance il cui valore è l'istanza di __impl contenuta in Singleton.__instance (unica per tutte le istanze di Singleton)

```
def __init__(self):

        """ Create singleton instance """

        # Check whether we already have an instance
        if Singleton.__instance is None:
            # Create and remember instance
            Singleton.__instance = Singleton.__impl()

        # Store instance reference as the only member in the handle
        self.__dict__['_Singleton__instance'] = Singleton.__instance
```

per capire perche' la variabile di istanza venga creata cosi` , guardate come continua l'implementazione della classe

66

Il pattern Singleton: la classe Singleton

NB: se creiamo una nuova classe che è sottoclasse di Singleton allora

- se __init__ della nuova classe invoca __init__ di Singleton allora __init__ di Singleton non crea una nuova istanza (non invoca Singleton._impl() nell'if)

- se __init__ della nuova classe non invoca __init__ di Singleton allora è evidente che non viene creata alcuna nuova istanza perché a crearle è __init__ di Singleton

67

Il pattern Singleton: la classe Singleton

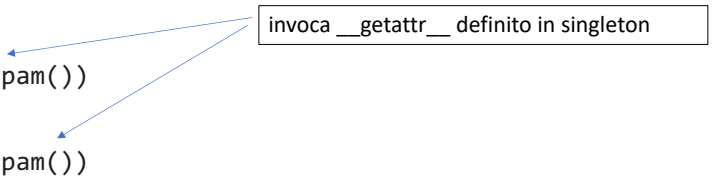
- Ridefinisce __getattr__ e __setattr__ in modo che quando si accede a o si modifica un attributo di un'istanza di Singleton, di fatto si accede a o si modifica l'attributo omonimo di Singleton.__instance

```
def __getattr__(self, attr):
     """ Delegate access to implementation """
     return getattr(self.__instance, attr)

def __setattr__(self, attr, value):
     """ Delegate access to implementation """
     return setattr(self.__instance, attr, value)
```

```
# Test it
s1 = Singleton()
print (id(s1), s1.spam())

s2 = Singleton()
print (id(s2), s2.spam())
```



__getattr__ e __getattribute__

- object.__getattr__(*self, name*) restituisce il valore dell'attributo di nome *name* o lancia un'eccezione AttributeError.
- Quando si accede ad un attributo di un'istanza di una classe viene invocato il metodo object.__getattribute__(*self*, *name*).
- Se la classe definisce anche __getattr__() allora quest'ultimo metodo viene invocato nel caso in cui __getattribute__() lo invochi esplicitamente o lanci un'eccezione AttributeError.
- __getattribute__() deve restituire il valore dell'attributo o lanciare un'eccezione AttributeError.
- L'implementazione di __getattribute__ () deve sempre invocare il metodo della classe base usando lo stesso nome per evitare la ricorsione infinita. Ad esempio, se si vuole invocare __getattribute__( ) di object occorre scrivere object.__getattribute__(self, name).

Il pattern Singleton: la classe Borg

- Nella classe Borg tutte le istanze sono diverse ma condividono lo stesso stato.
- Nel codice in basso, lo stato condiviso è nell'attributo _shared_state e tutte le nuove istanze di Borg avranno lo stesso stato così come è definito dal metodo __new__ .
- In genere lo stato di un'istanza è memorizzato nel dizionario *__dict__* proprio dell'istanza. Nel codice in basso assegnamo la variabile di classe *_shared_state* a tutte le istanze create

```
class Borg():
        _shared_state = {}
        def __new__(cls, *args, **kwargs):
                obj = super().__new__(cls, *args, **kwargs)
                obj.__dict__ = cls._shared_state
                return obj
```


__new__ e __init__

- __new__ crea un oggetto
- __init__ inizializza le variabili dell'istanza
- quando viene creata un'istanza di una classe viene invocato prima __new_ e poi _init__
- __new__ accetta cls come primo parametro perché quando viene invocato di fatto l'istanza deve essere ancora creata
- __init_ accetta self come primo parametro

__new__ e __init__

- tipiche implementazioni di __new_ creano una nuova istanza della classe cls invocando il metodo __new__ della superclasse con **super(currentclass, cls).__new__(cls,...)** . Tipicamente prima di restituire l'istanza __new__ modifica l'istanza appena creata.
- Se __new__ restituisce un'istanza di cls allora viene invocato il metodo __init__(self,...), dove self è l'istanza creata e i restanti argomenti sono gli stessi passati a __new__
- Se __new__ non restituisce un'istanza allora __init__ non viene invocato.
- __new__ viene utilizzato soprattutto per consentire a sottoclassi di tipi immutabili (come ad esempio str, int e tuple) di modificare la creazione delle proprie istanze.

72

Il pattern Singleton: la classe Borg

- creiamo istanze diverse di Borg: borg e another_borg
- creiamo un'istanza della sottoclasse Child di Borg
- aggiungiamo la variabile di istanza only_one_var a borg
- siccome lo stato è condiviso da tutte le istanze di Borg, anche child avrà la variabile di istanza only_one_var

```
class Child(Borg):
        pass
>>> borg = Borg()
>>> another_borg = Borg()
>>> borg is another_borg
False
>>> child = Child()
>>> borg.only_one_var = "I'm the only one var"
>>> child.only_one_var
I'm the only one var
```

73

Il pattern Singleton: la classe Borg

- Se vogliamo definire una sottoclasse di Borg con un altro stato condiviso dobbiamo resettare _shared_state nella sottoclasse come segue

```
class AnotherChild(Borg):
        _shared_state = {}

>>> another_child = AnotherChild()
>>> another_child.only_one_var
AttributeError:  AnotherChild instance has no attribute
'shared_staté
```


Il pattern Singleton

- Il libro di Summerfield “Python in Practice …”, il modo piu` semplice per realizzare le funzionalita` del singleton in Python è di creare un modulo con lo stato globale di cui si ha bisogno mantenuto in variabili ‘’private’’ e l’accesso fornito da funzioni ‘’pubbliche’’.
- Immaginiamo di avere bisogno di una funzione che restituisca un dizionario di quotazioni di valute dove ogni entrata è della forma (nome chiave, tasso di cambio).
- La funzione potrebbe essere invocata piu` volte ma nella paggior parte dei casi i valori dei tassi verrebbero acquisiti una sola volta.
- Vediamo come usare il design pattern Singleton per ottenere quanto descritto.

Il pattern Singleton

- All'interno di un modulo Rates.py possiamo definire una funzione get(), che è la funzione pubblica che ci permette di accedere ai tassi di cambio.
- La funzione get() ha un attributo rates che è il dizionario contenente i tassi di cambio della valute.
- I tassi vengono prelevati da get(), ad esempio accedendo ad un file pubblicato sul Web, solo la prima volta che viene invocata o quando i tassi devono essere aggiornati.
  - L'aggiornamento dei tassi potrebbe essere richiesto a get() mediante un parametro booleano, settato per default a False (aggiornamento non richiesto).

Esercizio

- Scrivere una classe C per cui accade che ogni volta che si aggiunge una variabile di istanza ad una delle istanze di C in realta` la variabile viene aggiunta alla classe come variabile di classe.
- Modificare la classe al punto precedente in modo tale che le istanze abbiano al piu` due variabili di istanza: varA e varB e non deve essere possibile aggiungere altre variabili di istanza oltre a queste due. Se il programma avesse bisogno di aggiungere altre variabili oltre a quelle sopra indicate, queste altre variabili verrebbero create come variabili di classe e non di istanza.

Module-level singleton

- Tutti i moduli sono per loro natura dei singleton per il modo in cui vengono importati in Python
- Passi per importare un modulo:

1. Se il modulo è già stato importato, questo viene restituito; altrimenti dopo aver trovato il modulo, questo viene inizializzato e restituito.
2. Inizializzare un modulo significa eseguire un codice includendo tutti gli assegnamenti a livello del modulo
3. Quando si importa un modulo per la prima volta, vengono fatte tutte le inizializzazioni. Quando si importa il modulo una seconda volta, Python restituisce il modulo inizializzato per cui l'inzializzazione non viene fatta.

78

Module-level singleton

- Per realizzare velocemente il pattern singleton, eseguiamo i seguenti passi e mantieniamo i dati condivisi nell'attributo del modulo.

singleton.py:

```
only_one_var = "I'm only one var"
```

module1.py:

```
import singleton
print (singleton.only_one_var )
singleton.only_one_var += " after modification"  #una nuova variabile only_one_var
import module2  # import singleton in module2 non inizializza singleton perche'
                #singleton è gia` stato importato in module1
```

module2.py:

```
import singleton
print (singleton.only_one_var)
```

Esecuzione di module1

```
I'm only one var
I'm only one var after modification
```


79

Design Pattern Proxy

- Proxy è un design pattern strutturale
  - fornisce una classe surrogato che nasconde la classe che svolge effettivamente il lavoro
- Quando si invoca un metodo del surrogato, di fatto viene utilizzato il metodo della classe che lo implementa.
- Quando un oggetto surrogato è creato, viene fornita un'implementazione alla quale vengono inviate tutte le chiamate dei metodi

Design Pattern Proxy

Usi di Proxy :

1. **Remote proxy** è un proxy per un oggetto in un diverso spazio di indirizzi.
   - Il libro "Python in Practice" descrive nel capitolo 6 la libreria RPyC (Remote Python Call) che permette di creare oggetti su un server e di avere proxy di questi oggetti su uno o più client

2. **Virtual proxy** è un proxy che fornisce una "lazy initialization" per creare oggetti costosi su richiesta solo se se sono realmente necessari.

3. **Protection proxy** è un proxy usato quando vogliamo che il programmatore lato client non abbia pieno accesso all'oggetto.

4. **Smart reference** è un proxy usato per aggiungere azioni aggiuntive quando si accede all'oggetto. Per esempio, per mantenere traccia del numero di riferimenti ad un certo oggetto

Design Pattern Proxy

```
class Implementation:
        def f(self):
                print("Implementation.f()")
        def g(self):
                print("Implementation.g()")
        def h(self):
                print("Implementation.h()")
class Proxy:
        def __init__(self):
                self.__implementation = Implementation()
        # Passa le chiamate ai metodi all'implementazione:
        def f(self): self.__implementation.f()
        def g(self): self.__implementation.g()
        def h(self): self.__implementation.h()
p = Proxy()
p.f(); p.g(); p.h()
```


Design Pattern Proxy

- Non è necessario che **Implementation** abbia la stessa interfaccia di **Proxy** ma è comunque conveniente avere un'interfaccia comune in modo che **Implementation** fornisca tutti i metodi che **Proxy** ha bisogno di invocare.

Design Pattern Proxy

L'uso di **__getattr__( )** rende **Proxy2** completamente generica e non legata ad una particolare implementazione

```
class Implementation2:
        def f(self):
                print("Implementation.f()")
        def g(self):
                print("Implementation.g()")
        def h(self):
                print("Implementation.h()")
class Proxy2:
        def __init__(self):
                self.__implementation = Implementation2()
        def __getattr__(self, name):
                return getattr(self.__implementation, name)
p = Proxy2()
p.f(); p.g(); p.h();
```