Programmazione Avanzata

Design Pattern: Decorator

1

Design Pattern

- Nel 1994, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides pubblicarono un libro intitolato **Design Patterns - Elements of Reusable Object-Oriented Software** in cui è stato introdotto il concetto di Design Pattern nell'Object Oriented Design (OOD).
- Il gruppo dei quattro autori è noto con il nome di **Gang of Four (GOF)**.
- Nel libro viene riportato il seguente pensiero dell'architetto Christopher Alexander: "Ciascun pattern descrive un problema che si presenta più e più volte nel nostro ambiente e poi descrive il nucleo della soluzione del problema, in modo tale che tu possa riutilizzare questa soluzione un milione di volte, senza mai applicarla alla stessa maniera."
- I quattro autori osservano che ciò che Christopher Alexander esprime riguardo ai pattern negli edifici e nelle città è vero anche quando si parla di object-oriented design pattern.
  - Solo che le soluzioni sono descritte in termini di interfacce e oggetti invece che di muri e porte.

2

Design Pattern

- Forniscono schemi generali per la soluzione di problematiche ricorrenti che si incontrano durante lo sviluppo del software
- Favoriscono il riutilizzo di tecniche di design di successo nello sviluppo di nuove soluzioni
- Evitano al progettista di riscoprire ogni volta le stesse cose
- Permettono di sviluppare un linguaggio comune che semplifica la comunicazione tra le persone coinvolte nello sviluppo del software

3

Design Pattern

- Per definire un design pattern occorre specificare:
- **Il nome del pattern.** Associare dei nomi ai design pattern consente un più elevato livello di astrazione nella fase di progettazione e facilita la comunicazione tra gli addetti ai lavori e la documentazione.
- **Il problema.** Il problema descrive in quali contesti ha senso applicare il pattern.
- **La soluzione.** La soluzione fornisce la descrizione astratta di un problema (nel nostro caso di OOD) e indica come utilizzare gli strumenti a disposizione (nel nostro caso classi e oggetti) per risolverlo.
- **Le conseguenze.** Le conseguenze descrivono i risultati dell'applicazione del design pattern. Esse sono fondamentali per valutare le diverse alternative e comprendere i costi e i benefici risultanti dall'applicazione del pattern

4

Design Pattern: elenco

- 1.Adapter
- 2.Facade
- 3.Composite
- 4.Decorator
- 5.Bridge
- 6.Singleton
- 7.Proxy
- 8.Flyweight
- 9.Strategy
- 10.State
- 11.Command
- 12.Observer

13. Memento
14.Interpreter
15.Iterator
16. Visitor
17.Mediator
18.Template Method
19.Chain of Responsibility
20.Builder
21.Prototype
22.Factory Method
23.Abstrac Factory

5

Design Pattern: classificazione

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

6

Design Pattern: classificazione

- I pattern creazionali riguardano il processo di creazione degli oggetti
- I pattern strutturali riguardano la composizione di classi ed oggetti
- I pattern comportamentali caratterizzano i modi in cui le classi e gli oggetti interagiscono tra di loro e si distribuiscono le responsabilità

7

Design Pattern Creazionali

- I design pattern creazionali astraggono il processo di creazione
- Aiutano a rendere il sistema indipendente da come i suoi oggetti sono creati, composti e rappresentati
- I design pattern di questo tipo diventano sempre più utili man mano che il sistema diventa sempre più dipendente dalla composizione di oggetti. Man mano che ciò accade, l'enfasi si sposta dalla codifica di un insieme fissato di comportamenti alla definizione di un insieme più piccolo di comportamenti fondamentali che possono essere composti per dar vita a comportamenti più complessi

8

Design Pattern strutturali

- I Design Pattern strutturali riguardano le relazioni tra entità quali classi e oggetti
  - forniscono metodologie semplici per comporre oggetti per creare nuove funzionalità

9

Design Pattern Comportamentali

- I pattern comportamentali riguardano il modo in cui le cose vengono fatte, in altre parole, gli algoritmi e le interazioni tra oggetti.

- Forniscono modi efficaci per pensare e organizzare la computazione.
- Alcuni di questi pattern sono built-in in Python.

10

Riferimenti

- Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides **"Design Patterns - Elements of Reusable Object-Oriented Software",** Addison-Wesley

- Mark Summerfield, **"Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns ",** Addison-Wesley Professional

11

Il pattern Decorator

- È un design pattern **strutturale**.
- Serve quando vogliamo estendere le funzionalità di singoli oggetti dinamicamente
- Ad esempio un sistema GUI dovrebbe consentire di aggiungere proprietà (ad esempio, i bordi) o comportamenti (ad esempio, lo scrolling) ad ogni componente dell'interfaccia utente.
- Un modo per far questo è l'ereditarietà: ereditare un bordo da un'altra classe mette un bordo intorno ad ogni istanza della sottoclasse.
- Ciò è poco flessibile perché la scelta di un bordo è fatta in modo statico. Non è possibile controllare come e quando decorare la componente con un bordo.
- Un approccio più flessibile consiste nel racchiudere la componente in un altro oggetto che si occupa di aggiungere il bordo.
- Tale oggetto è chiamato decoratore.
- Il decoratore inoltra richieste alla componente e può svolgere azioni aggiuntive, come aggiungere un bordo o altre proprietà.

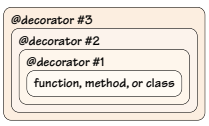
12

Il pattern Decorator

- Un *decoratore di funzione* è una funzione che ha come unico argomento una funzione e restituisce una funzione con lo stesso nome della funzione originale ma con ulteriori funzionalità
- Un *decoratore di classe* è una funzione che ha come unico argomento una classe e restituisce una classe con lo stesso nome della classe originale ma con funzionalità aggiuntive.
  - I decoratori di classe possono a volte essere utilizzati come alternativa alla creazione di sottoclassi
- In python c'è un supporto built-in per i decoratori di funzioni (e di metodi) e per i decoratori di classe.

Function Decorator

- Tutti i decoratori di funzioni o di metodi hanno la stessa struttura
  - Creazione della funzione wrapper:
    - All'interno del wrapper invochiamo la funzione originale.
    - Prima di invocare la funzione originale possiamo effettuare qualsiasi lavoro di preprocessing
    - Dopo la chiamata siamo liberi di acquisire il risultato, di fare qualsiasi lavoro di postprocessing e di restituire qualsiasi valore vogliamo.
    - Alla fine restituiamo la funzione wrapper come risultato del decoratore e questa funzione sostituisce la funzione originale acquisendo il suo nome.
  - Applicazione di un decoratore:
    - Si scrive il simbolo @, allo stesso livello di indentazione dello statement def seguito immediatamente dal nome del decoratore.
    - è possibile applicare un decoratore ad una funzione decorata.



Function Decorator

- La funzione mean() senza decoratore ha due o più argomenti numerici e restituisce la loro media come un float.
- Senza il decoratore la chiamata mean(5, "6", "7.5") genera un TypeError perché non è possibile sommare int e str.

```
@float_args_and_return
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
```


Function Decorator

- La funzione mean() decorata con il decoratore float_args_and_return può accettare due o più argomenti di qualsiasi tipo che convertirà in un float.
- Con la versione decorata, mean(5, "6", "7.5") non genera errore dal momento che float("6") and float("7.5") producono numeri validi.

.

```
def float_args_and_return(function):
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

Function Decorator

- Nel codice in basso, è stata creata la funzione senza il decoratore e poi sostituita con il decoratore invocando il decoratore
- A volte è necessario invocare i decoratori direttamente
  - vedremo in seguito degli esempi

```
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
mean = float_args_and_return(mean)
```


Function Decorator

- La funzione float_args_and_return() è un decoratore di funzione per cui ha come argomento una singola funzione
- Per convenzione, le funzioni wrapper hanno come argomenti un parametro che indica un numero variabile di parametri (*args, nell'esempio) e un parametro di tipo keyword (**kwargs, nell'esempio)
- Eventuali vincoli sugli argomenti sono gestiti dalla funzione originale. Nel creare il decoratore, dobbiamo solo assicurarci che alla funzione originale vengano passati tutti gli argomenti.

Function Decorator

- Per come è stato scritto il decoratore float_args_and_return ,
  - la funzione decorata avrà il valore dell'attributo __name__ settato a "wrapper" invece che con il nome originale della funzione
  - non ha docstring anche nel caso in cui la funzione originale abbia una docstring
- Per ovviare a questo inconveniente, la libreria standard di Python include il decoratore **@functools.wraps** che può essere usato per decorare una funzione wrapper dentro il decoratore e assicurare che gli attributi __name__ and __doc__ della funzione decorata contengano rispettivamente il nome e la docstring della funzione originale.

Function Decorator

- La versione in basso del decoratore float_args_and_return usa il decoratore @functools.wraps per garantire che la funzione wrapper()
  - abbia il suo attributo __name__ correttamente settato con il nome della funzione passata come argomento al decoratore (mean, nel nostro esempio)
  - abbia la docstring della funzione originale (non presente, nel nostro esempio)
- è sempre consigliabile usare il decoratore @functools.wraps dal momento che il suo uso ci assicura che
  - nei traceback vengano visualizzati i nomi corretti delle funzioni
  - si possa accedere alle docstring delle funzioni originali

```
def float_args_and_return(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

Function Decorator: esercizio

- Scrivere il decoratore di funzione decf che fa in modo che venga lanciata l ’eccezione TypeError se il numero di argomenti è diverso da due. Altrimenti, se la funzione decorata restituisce un risultato, questo viene aggiunto insieme al valore del primo argomento in un file di nome “**risultato.txt**”.
- Suggerimento: Ricordatevi di convertire a stringa il valore del primo argomento e il risultato quando li scrivete nel file e di aprire il file in modo da non cancellare quanto scritto precedentemente nel file.

Function Decorator: soluzione esercizio

```
from functools import wraps

def decf(f):
    @wraps(f)
    def wrapper(*args,**kwargs):
        if len(args)+len(kwargs)!=2:
            raise TypeError
        else:
            f_o=open("risultato.txt",'a')
            res=f(*args,**kwargs)
            if res!=None:
                f_o.write(res)
            if args:
                f_o.write(str(args[0]))
            else :
                f_o.write(str(next(iter(kwargs.values()))))
            f_o.write("\n")
            f_o.close()
    return wrapper
```

Function Decorator: esercizio

- Scrivere il decoratore di funzione **decora** che trasforma la funzione decorata in una funzione che lancia l'eccezione **TypeError** se uno o più argomenti non sono di tipo str. La funzione deve restituire una stringa formata dagli argomenti ricevuti in input e dal risultato intervallati da uno spazio. Non dimenticate di convertire il risultato in stringa quando lo inserite nella stringa output.
- Esempio: se la funzione riceve in input **"il" , "risultato", "è" ,** la funzione non lancia l'eccezione e stampa "Il risultato è …" dove al posto dei puntini deve apparire il risultato della funzione.

Class Decorator

- I decoratori di classe sono simili ai decoratori di funzioni ma sono eseguiti al termine di uno statement class
- I decoratori di classe sono funzioni che ricevono una classe come unico argomento e restituiscono una nuova classe con lo stesso nome della classe originale ma con funzionalità aggiuntive.
  - I decoratori di classe possono essere usati sia per gestire le classi dopo che esse sono state create sia per inserire un livello di logica extra (wrapper) per gestire le istanze della classe quando sono create.

```
def decorator(aClass): ...

@decorator
class C: ...
```

è equivalente a

```
def decorator(aClass): ...

class C: ...
C = decorator(C)
```

Class Decorator

- è possibile usare questo decoratore per dotare automaticamente le classi con una variabile numInstances per contare le istanze.
- è possibile usare lo stesso approccio per aggiungere altri dati

classdec0.py

```
def count(aClass):
   aClass.numInstances = 0
   return aClass

@count
class Spam:
   def __init__(self):
      Spam.numInstances =Spam.numInstances + 1

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

```
>>> from classdec0.py import Spam, Sub,
Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
3
>>> print(sub.numInstances)
3
>>> print(other.numInstances)
3
```


Class Decorator

- Per come è stato definito nella slide precedente, il decoratore count può essere applicato sia a classi che a funzioni

```
@count                          #equivalente a f=count(f)
def f(): pass

@count                          #equivalente a Other=count(Other)
class Other: pass

spam.numInstances               #entrambi settati a 0
Other.numInstances
```

Class Decorator

```
def count(aClass):
   aClass.numInstances = 0
   return aClass

@count
class Spam:
   def __init__(self):
      Spam.numInstances =Spam.numInstances + 1

@count
class Sub(Spam):
   pass

@count
class Other(Spam):
   def __init__(self):
      Other.numInstances =Other.numInstances + 1
```

classdec1.py

- In questo esempio ogni classe ha la sua variabile numInstances.
- Quando viene creato un oggetto di tipo Sub viene invocato __init__ della classe base Spam e viene incrementato numInstances di Spam
- Quando viene creato un oggetto di tipo Other viene invocato __init__ di Other e incrementato numInstances di Other

```
>>> from classdec1.py import Spam, Sub, Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
2
>>> print(sub.numInstances)
0
>>> print(other.numInstances)
1
```


Class Decorator

```
def count(aClass):
   aClass.numInstances = 0
   return aClass

@count
class Spam:
   @classmethod
   def count(cls):
      cls.numInstances+=1
   def __init__(self):
      self.count()

@count
class Sub(Spam):
   pass

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

classdec2.py

```
>>> from classdec2.py import Spam, Sub, Other
>>> spam=Spam()
>>> sub=Sub()
>>> other=Other()
>>> print(spam.numInstances)
1
>>> print(sub.numInstances)
1
>>> print(other.numInstances)
1
>>> other=Other()
>>> print(other.numInstances)
2
>>> print(spam.numInstances)
1
>>> print(sub.numInstances)
1
```

Class Decorator

- Nell'ultimo esempio, ogni volta che viene creato un oggetto di tipo Other o di tipo Sub viene eseguito __init__ della classe base Spam che invoca il metodo di classe count passandogli come argomento self.
  - Di conseguenza, count incrementa la variabile numInstances di Other se si sta creando un'istanza di Other e di Sub se si sta creando un'istanza di Sub.
- Non ha molto senso aver dotato le classi della variabile numInstances mediante un decoratore di classe e aver inserito il codice per aggiornare questa variabile direttamente nelle classi
  - Le classi non potrebbero funzionare correttamente se non fossero decorate con count (direttamente o decorando la classe base)
- Nel prossimo esempio vediamo come aggiungere ad una classe la funzionalità per contare le istanze mediante il decoratore.

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
```

Class Decorator

stesso comportamento del codice in classdec2.py

classdec3.py

```
def count(aClass):
   aClass.numInstances = 0
   oldInit=aClass.__init__
   def __newInit__(self,*args,**kwargs):
      if(aClass==type(self)):   #questo if evita che venga incrementato
                                # anche numInstance della classe base
         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)
1
>>> print(sub.numInstances)
1
>>> print(other.numInstances)
1
>>> other=Other()
>>> print(other.numInstances)
2
>>> print(spam.numInstances)
1
>>> print(sub.numInstances)
1
```


31

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.

32

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.

Late binding

- Nella programmazione funzionale il termine chiusura indica la capacità di un oggetto 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.
- Late binding: in Python i valori delle variabili usati nelle chiusure 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

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.")
```

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
```


37

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.")
```


38

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

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",...) serve come parametro 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: ...))))
```


41

Class Decorator

- La funzione ensure() crea un decoratore di classe parametrizzato dal nome della proprietà (name), dalla funzione di validazione (validate) e da una docstring opzionale (doc).
- Ogni volta che un decoratore di classe restituito da ensure() è usato per una particolare classe, quella classe viene dotata 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 cas 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

42

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
```

43

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

```
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 nessuno tra minimum o maximum è diverso da 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

44

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))
```


Class Decorator

```
@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 dell'istanza. 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
```


47

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
```


48

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
```


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)

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): ...

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)
```

Programmazione Avanzata

Design Pattern: Singleton

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

57

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

58

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
```


59

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
```


60

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

61

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())
```

invoca __get__attr__ definito in singleton

62

__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 il nome della classe base 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

65

__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.

66

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
```

67

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
```


68

Il pattern Singleton

- Secondo 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).

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 resituisce il modulo inizializzato per cui l'inzializzazione non viene fatta.

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)
```