# Programmazione Avanzata

## Design Pattern: Decorator (II parte)





## 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
```



28

# 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
```



29

# 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.



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
```

classdec3.py

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.

Di conseguenza,
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()



32

# 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.



33

# 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.





# Programmazione Avanzata

## Design Pattern: Decorator (II parte)

# 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

- 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 funzioni innestate (annidate):
  - Una funzione innestata è definita all'interno di un altra funzione.
  - Una funzione innestata puo` accedere allo scopo della funzione che la racchiude, detto non-local scope.
    - Per default queste variabili sono di sola lettura e per modificarle occorre 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.



38

# 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 è inner
g=f()     #g è innnerinner
g()
```

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



39

# 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()
```

```
Traceback (most recent call last):
  File "/Users/adb/Documents/pop.py", line 16, in
<module>
    g=f()
  File "/Users/adb/Documents/pop.py", line 7, in inner
    print("il valore di z stampato da inner è:", z)
NameError: free variable 'z' referenced before
assignment in enclosing scope
```

quando viene invocato inner, z è stato 'distrutto' e si ha errore





# 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.")
```

# 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
```





# 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

# 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: ...))))
```



48

# 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



49

# 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
```

50

# 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



51

# 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

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