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

64

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

65

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 attraverso 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

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

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 __getattr__ definito in singleton

Il pattern Singleton: la classe Singleton

- nella slide 67 si usa questo assegnamento perche' se avessimo usato self._Singleton__instance, sarebbe stato invocato __setattr__ della slide precedente e avremmo aggiunto l'attributo all'oggetto self._instance

```
self.__dict__['_Singleton__instance'] = Singleton.__instance
```


__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.
- Per accedere a un attributo di self, l'implementazione di __getattribute__ () deve sempre invocare il metodo __getattribute__ della classe base per evitare la ricorsione infinita.

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.

74

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
```

75

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).

78

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.

79

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.

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
```

Programmazione Avanzata

Design Pattern: Proxy

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

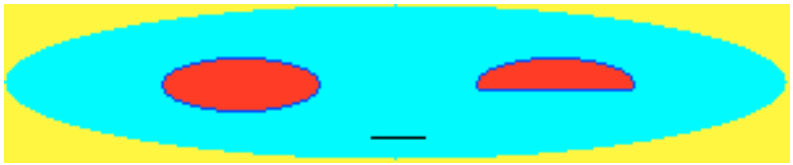
L'uso di **__getattr__( )** rende **Proxy2** completamente generica e non legata ad una particolare implementazione

```
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();
```

Design Pattern Proxy: esempio

- Abbiamo bisogno di creare immagini delle quali però una sola verrà usata realmente alla fine.
- Abbiamo un modulo Image e un modulo quasi equivalente più veloce cyImage. Entrambi i moduli creano le loro immagini in memoria.
- Siccome avremo bisogno solo di un'immagine tra quelle create, sarebbe meglio utilizzare dei proxy "leggeri" che permettano di creare una vera immagine solo quando sapremo di quale immagine avremo bisogno.
- L'interfaccia Image.Image consiste di 10 metodi in aggiunta al costruttore: load(), save(), pixel(), set_pixel(), line(), rectangle(), ellipse(), size(), subsample(), scale().
  - Non sono elencati alcuni metodi statici aggiuntivi, quali Image.Image.color_for_name() e Image.color_for_name().

Design Pattern Proxy: esempio



```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
    for color in ("yellow", "cyan", "blue", "red", "black"))
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```

Design Pattern Proxy: esempio

- La classe ImageProxy può essere usata al posto di Image.Image (o di qualsiasi altra classe immagine che supporta l'interfaccia di Image) a patto che l'interfaccia incompleta fornita da ImageProxy sia sufficiente.
- Un oggetto ImageProxy non salva un'immagine ma mantiene una lista di tuple di comandi dove il primo elemento in ciascuna tupla è una funzione o un metodo unbound (non legato ad una particolare istanza) e i rimanenti elementi sono gli argomenti da passare quando la funzione o il metodo è invocato.

```
class ImageProxy:

    def __init__(self, ImageClass, width=None, height=None, filename=None):
        assert (width is not None and height is not None) or \
                filename is not None
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
        else:
            self.commands = [(self.Image, width, height)]

    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```


90

Design Pattern Proxy: esempio

- Quando viene creato un ImageProxy, gli deve essere fornita l'altezza e la larghezza dell'immagine o il nome di un file. In caso contrario viene lanciata AssertionError.
- Se viene fornito il nome di un file, l'ImageProxy immagazzina una tupla con il costruttore Image.Image(), None e None (per la larghezza e l'altezza) e il nome del file da cui il metodo load di ImageClass carichera` le informazioni per costruire l'immagine.
- Se non viene fornito il nome di un file allora viene immagazzinato il costruttore Image.Image() insieme alla larghezza e l'altezza.

```
class ImageProxy:

    def __init__(self, ImageClass, width=None, height=None, filename=None):
        assert (width is not None and height is not None) or \
                filename is not None
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
        else:
            self.commands = [(self.Image, width, height)]

    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```


91

Design Pattern Proxy: esempio

- La classe Image.Image ha 4 metodi: line(), rectangle(), ellipse(), set_pixel().
- La classe ImageProxy supporta pienamente questa interfaccia solo che invece di eseguire questi comandi, semplicemente li aggiunge insieme ai loro argomenti alla lista dei comandi.
- Il metodo inserito all'inizio della tupla è unbound in quanto non è legato ad un'istanza di self.Image (self.Image è la classe che fornisce il metodo)

```
def set_pixel(self, x, y, color):
    self.commands.append((self.Image.set_pixel, x, y, color))

def line(self, x0, y0, x1, y1, color):
    self.commands.append((self.Image.line, x0, y0, x1, y1, color))

def rectangle(self, x0, y0, x1, y1, outline=None, fill=None):
    self.commands.append((self.Image.rectangle, x0, y0, x1, y1,
            outline, fill))

def ellipse(self, x0, y0, x1, y1, outline=None, fill=None):
    self.commands.append((self.Image.ellipse, x0, y0, x1, y1,
            outline, fill))
```


92

Design Pattern Proxy: esempio

- Solo quando si sceglie di salvare l'immagine, essa viene effettivamente creata e viene quindi pagato il prezzo relativo alla sua creazione, in termini di computazione e uso di memoria.
- Il primo comando della lista self.commands è sempre quello che crea una nuova immagine. Quindi il primo comando viene trattato in modo speciale salvando il suo valore di ritorno (che è un Image.Image o un cyImage.Image) in image.
- Poi vengono invocati nel for i restanti comandi passando image come argomento insieme agli altri argomenti.
- Alla fine, si salva l'immagine con il metodo Image.Image.save().

```
def save(self, filename=None):
    command = self.commands.pop(0)
    function, *args = command
    image = function(*args)
    for command in self.commands:
        function, *args = command
        function(image, *args)
    image.save(filename)
    return image
```

93

Design Pattern Proxy: esempio

- Il metodo Image.Image.save() non ha un valore di ritorno (sebbene possa lanciare un'eccezione se accade un errore).
- L'interfaccia è stata modificata leggermente per ImageProxy per consentire a save() di restituire l'immagine Image.Image creata per eventuali ulteriori usi dell'immagine.
- Si tratta di una modifica innocua in quanto se il valore di ritorno è ignorato, esso viene scartato.

```
    def save(self, filename=None):
        command = self.commands.pop(0)
        function, *args = command
        image = function(*args)
        for command in self.commands:
            function, *args = command
            function(image, *args)
        image.save(filename)
        return image
```


Design Pattern Proxy: esempio

- Se un metodo non supportato viene invocato (ad esempio, pixel()), Python lancia un AttributeError.
- Un approccio alternativo per gestire i metodi che non sono supportati dal proxy è di creare una vera immagine non appena uno di questi metodi è invocato e da quel momento usare la vera immagine.

Design Pattern Proxy: esempio

Questo codice prima crea alcune costanti colore con la funzione color_for_name del modulo Image e poi crea un oggetto ImageProxy passando come argomento a __init__ la classe che si vuole usare. L'ImageProxy creato è usato quindi per disegnare e infine salvare l'immagine risultante.

```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
    for color in ("yellow", "cyan", "blue", "red", "black"))
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```


96

Design Pattern Proxy: esempio

- Il codice alla pagina precedente avrebbe funzionato allo stesso modo se avessimo usato Image.image() al posto di ImageProxy().
- Usando un image proxy, la vera immagine non viene creata fino a che il metodo save non viene invocato. In questo modo il costo per creare un'immagine prima di salvarlo è estremamente basso (sia in termini di memoria che di computazione) e se alla fine scartiamo l'immagine senza salvarla perdiamo veramente poco.
- Se usassimo Image.Image, verrebbe effettivamente creato un array di dimensioni width × height di colori e si farebbe un costoso lavoro di elaborazione per disegnare (ad esempio, per settare ogni pixel del rettangolo) che verrebbe sprecato se alla fine scartassimo l'immagine.

97

Esercizio 22 ottobre -1

1. Scrivere una classe MyProxy che è il proxy della classe MyClass. Ogni volta che viene invocato un metodo di istanza della classe MyProxy, di fatto viene invocato l'omonimo metodo di istanza di MyClass. **NON deve essere usata l'ereditarietà.**
   - **Si assuma che __init__ di MyClass prenda in input un argomento x e che il comportamento dei suoi metodi di istanza dipenda dal valori di x passati a __init__.**

Esercizio 22 ottobre-2

- Scrivere un decorator factory che prende in input una classe ClasseConFF e due stringhe funz e ff e restituisce un decoratore di classe che decora una classe in modo tale che se viene invocata funz di fatto al posto di funz viene invocata la funzione ff della classe ClasseConFF.