Programmazione Avanzata

Design Pattern: Singleton

Module-level singleton

Module-level 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 maggior parte dei casi i valori dei tassi verrebbero acquisiti una sola volta.
- Vediamo come usare il design pattern Singleton per ottenere quanto descritto.

Module-level 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)
```

esecuzione di module1.py

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


Module-level singleton

```
_URL = "http://www.bankofcanada.ca/stats/assets/csv/fx-seven-day.csv"
def get(refresh=False):
        if refresh:
                get.rates = {}              #reset della tabella dei cambi
        if get.rates:
                return get.rates
        with urllib.request.urlopen(_URL) as file:
           for line in file:
                ...                         #codice (non riportato) per estrarre informazioni da _URL
              get.rates[key] = value        #key e value ottenuti dalle linee di codice per estrarre informazioni
        return get.rates

get.rates = {}        #inizializzazione tabella dei cambi
```

Module-level singleton

- La funzione get è in un modulo Rates.py che viene importato in quello contenente il programma principale
- il valore singleton è la tabella dei tessi di cambio ed è memorizzato nella variabile globale get.rates di Rates.py
- Quando viene importato il modulo Rates.py, la tabella get.rates è inizializzata con dizionario vuoto
- La funzione Rate.get (di cui rates è un attributo) ricomputa i valori di get.rates solo se invocata per la prima volta o se è invocata con il parametro refresh uguale a true (refresh di default è false).
- Il programma invoca Rates.get con refresh uguale a false per ottenere la tabella dei tassi di cambio ma non accede direttamente a Rates.get.rates. In questo modo Rates.get.rates è una variabile privata del modulo.
  - Rates.get.rates viene inizializzato una sola volta fuori da Rates.get

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();
```

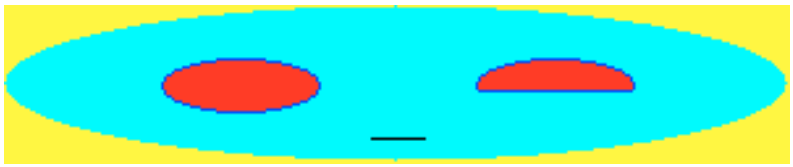

82

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().

83

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)]
```

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.
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)]
```


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))
```

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
```

88

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
```


89

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 possono essere delegati 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)
```

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.