Programmazione Avanzata

Design Pattern: Prototype

Il pattern Prototype

- Il pattern Prototype è un design pattern creazionale usato per creare nuovi oggetti clonando un oggetto preesistente e poi modificando il clone così creato.

Il pattern Prototype: esempio

- Point è la classe preesistente

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```


65

Il pattern Prototype: esempio

- In questo codice cloniamo nuovi punti in diversi modi

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14) # Could have used any of point1 to point6
```


66

Il pattern Prototype: esempio

- point1 = Point(1, 2)
  - viene semplicemente invocato il costruttore della classe Point.
  - point 1 è creato in modo statico. Di seguito creeremo istanze di Point in modo dinamico.
- point2 = **eval**("{}({}, {})".format("Point", 2, 4))
  - usa eval() per creare istanze di Point
  - eval valuta l'espressione Python rappresentata dalla stringa ricevuta come argomento. In altre parole, esegue il codice passato come argomento

Il pattern Prototype: esempio

- point3 = **getattr**(sys.modules[__name__], "Point")(3, 6)
  - usa getattr() per creare un'istanza
  - **getattr(object,name, default)** restituisce il valore dell'attributo dell'oggetto
    - **object** : oggetto per il quale viene restituito il valore dell'attributo nominato
    - **name** : stringa che contiene il nome dell'attributo
    - **default (opzionale):** valore restituito quando l'attributo specificato non viene trovato
  - nel codice in alto
  - **sys.modules** è un dizionario che mappa i nomi dei moduli ai moduli che sono già stati caricati
  - l'espressione sys.modules[__name__] restituisce il modulo in cui essa si trova
  - l'espressione **getattr**(sys.modules[__name__], "Point") restituisce il valore dell'attributo Point del modulo

Il pattern Prototype: esempio

- point4 = **globals**()["Point"](4, 8)

  - La funzione built-in globals() restituisce un dizionario che rappresenta la tavola dei simboli globali. All'interno di una funzione o un metodo, il dizionario è quello relativo al modulo dove è definita la funzione (o il metodo), non quello in cui è invocata .
  - comportamento similie a **getattr(object,name, default)**

Il pattern Prototype: esempio

- point5 = **make_object**(Point, 3,9)

  - usa la funzione make-object

Il pattern Prototype: esempio

- point6 = copy.deepcopy(point5)
  - usa il classico approccio basato su Prototype:
    - prima clona un oggetto esitente
    - poi lo inizializza con le istruzioni successive

Python ha un support built-in per creare oggetti sulla base di prototipi: la funzione copy.deepcopy().

- point7 = point1.__class__(7, 14)
  - point7 è creato usando point1
  - instanza.__class__ contiene la classe a cui appartiene istanza

Programmazione Avanzata

Design Pattern: Flyweight

Il Design Pattern Flyweight

- Il pattern Flyweight è concepito per gestire un grande numero di oggetti relativamente piccoli dove molti degli oggetti sono duplicati l'uno dell'altro.
- Il pattern è implementato in modo da avere un'unica istanza per rappresentare tutti gli oggetti uguali tra loro. Ogni volta che è necessario, questa unica istanza viene condivisa.
- Python permette di implementare Flyweight in modo naturale grazie all'uso dei riferimenti.

Il Design Pattern Flyweight

- Probabilmente il modo più semplice per trarre vantaggio dal pattern Flyweight in Python è di usare un dict, in cui ciascun oggetto (unico) corrisponde ad un valore identificato da un'unica chiave.
  - Ciò assicura che ciascun oggetto distinto viene creato un'unica volta, indipendentemente da quante volte viene usato.
- In alcune situazioni si potrebbero avere molti oggetti non necessariamente piccoli dove gran parte di essi o tutti sono unici. Un facile modo per ridurre l'uso della memoria in questo è di usare __slots__

__slots__

- In Python ogni classe può avere attributi di istanza.
- Per default Python usa un dict per immagazzinare gli attributi di instanza di un oggetto. Ciò è molto utile perché consente di settare nuovi attributi durante l'esecuzione.
- Comunque per classi piccole con attributi noti questo comportamento potrebbe essere un collo di bottiglia in quanto il dict comporterebbe uno spreco di RAM nel caso in cui vengano creati molti oggetti.

__slots__

- Un modo per evitare questo spreco di RAM e di usare __slots__ per indicare a Python di non usate un dict, e di allocare spazio solo per un insieme fissato di attributi.

- __slots__ è una variabile di classe a cui può essere assegnata una stringa, un iterabile, o una sequenza di stringhe.
- __*slots*__ riserva spazio per le variabili dichiarate e previene la creazione automatica di __*dict*__ per ciascuna istanza.

__slots__

- Vediamo un esempio di implementazione della stessa classe con e senza __slots__ .
- Senza __slots__

```
class MyClass():
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
```


77

__slots__

- con __slots__

```
class MyClass():
    __slots__ = ['name', 'identifier']
    def __init__(self, name, identifier):
        self.name = name
        self.identifier=identifier
```

Non posso aggiungere altre variabili di istanza alle istanze di MyClass

78

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ("x", "y", "z", "color")

    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

- La classe Point mantiene una posizione nello spazio tridimensionale e un colore.
- Grazie a __slots__, nessun Point ha il suo dict (self.__dict__) privato.
- Ciò vuol dire che nessun attributo può essere aggiunto ai singoli punti.
- Un programma per creare una tupla di un milione di punti ha impiegato su una stessa macchina
- nella versione con slots, circa 2 secondi e il programma ha occupato 183 Mebibyte di RAM
- nella versione senza slots, una frazione di secondo in meno ma il programma ha occupato 312 Mebibyte di RAM.

- Per default Python sacrifica sempre la memoria a favore della velocità ma in questo caso è conveniente invertire queste priorità.

79

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

Questo è l'inizio di un'altra classe Point.
Essa utilizza un database DBM (chiave-valore) immagazzinato in un file su disco.
Un riferimento al DBM è mantenuto nella variabile Point.__dbm.
Tutti i punti condividono lo stesso file DBM.

Uno "shelf" è un oggetto persistente simile ad un dizionario. I valori (non le chiavi) in uno shelf possono essere arbitrari oggetti gestibili dal modulo pickle. Ciò include la maggior parte di istanze di classi, tipi di dati ricorsivi, e oggetti contenenti molti oggetti condivisi.
Le chiavi sono stringhe.
shelve.open apre un dizionario persistente.
Il filename specificato è il nome di base per il database sottostante.
Per default il file database è aperto in lettura e scrittura.

80

Il Design Pattern Flyweight

```
class Point:

    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

tempfile.gettempdir() restituisce il nome della directory usata per i file temporanei.

Il comportamento di default di open fa in modo che venga creato il file DBM se non esiste già .
Il modulo shelve serializza i valori immagazzinati e li deserializza quando i valori vengono recuperati dal database.
Il processo di deserializzazione in Python non è sicuro perché esegue dell'arbitrario codice Python e di conseguenza non dovrebbe mai essere effettuato su dati provenienti da fonti non affidabili,

Il Design Pattern Flyweight

```
    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

A differenza del metodo __init__() della prima classe Point, questo metodo assegna i valori delle variabili in un file DBM.

```
    def __getattr__(self, name):
        return Point.__dbm[self.__key(name)]
```

Questo metodo è invocato ogni volta che si accede ad un attributo della classe

Il Design Pattern Flyweight

```
def __key(self, name):
    return  "{:X}:{}".format(id(self), name)
```

Questo metodo fornisce una stringa chiave per ognuno degli attributi x, y, z e color.
La chiave è ottenuta dall'ID restituita da id(self) in esadecimale e dal nome dell'attributo. Per esempio se l'ID di un punto è 3954827, il suo attributo x avrà chiave "3C588B:x", il suo attributo y avrà chiave "3C588B:y", e così via.

Il design pattern Flyweight

- Le chiavi e i valori dei database DBM devono essere byte.
- Per fortuna, i moduli DBM Python accettano sia stringhe (str) che byte come chiavi convertendo le stringhe in byte.
- In particolare, il modulo shelve, qui usato, permette di immagazzinare un qualsiasi valore gestibile dal modulo pickle.
- Un valore recuperato dal database è convertito dalla rappresentazione sotto forma di sequenza di bytes nel tipo originario.

Il Design Pattern Flyweight

```
def __setattr__(self, name, value):
    Point.__dbm[self.__key(name)] = value
```

Ogni volta che un attributo di Point è settato (ad esempio, point.y = y), viene invocato questo metodo.
Il valore value immagazzinato è convertito in un flusso di byte.

Il Design Pattern Flyweight

Sulla macchina usata per i test, la creazione di un milione di punti ha richiesto circa un minuto ma il programma ha occupato solo 29 Mebibyte of RAM (più 361 Mebibyte di spazio su disco) mentre la prima versione di Point ha richiesto 183 Mebibyte di RAM.