Programmazione Avanzata

Design Pattern (II parte)

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'acceso 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

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

Il pattern Singleton: la classe Singleton

```
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
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

47

Il pattern Singleton: la classe 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
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

48

Il pattern Singleton: la classe Singleton

```
    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

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 perche' a crearle è __init__ di Singleton

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(Borg, cls).__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 perche' 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.

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
```

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
```


Adapter

- L'Adapter è un design pattern strutturale che ci aiuta a rendere compatibili interfacce tra di loro incompatibili
- In altre parole l'Adapter crea un livello che permette di comunicare a due interfacce differenti che non sono in grado di comunicare tra di loro
- Esempio: Un sistema di e-commerce contiene una funzione calculate_total(order) in grado di calcolare l'ammontare di un ordine solo in Corone Danesi (DKK).
  - Vogliamo aggiungere il supporto per valute di uso più comune quali i Dollari USA (USD) e gli Euro (EUR).
  - Se possediamo il codice sorgente del sistema possiamo estenderlo in modo da incorporare nuove funzioni per effettuare le conversioni da DKK a EUR e USD.
  - Che accade però se non disponiamo del sorgente perchè l'applicazione ci è fornita da una libreria esterna? In questo caso, possiamo usare la libreria ma non modificarla o estenderla.
  - La soluzione fornita dall'Adapter consiste nel creare un livello extra (wrapper) che effettuala la conversione tra i formati delle valute.

Adapter: un semplice esempio

- La nostra applicazione ha una classe Computer che mostra l'informazione di base riguardo ad un computer.

```
class Computer:
	def __init__(self, name):
		self.name = name
	def __str__(self):
		return 'the {} computer'.format(self.name)
	def execute(self): return 'executes a program'
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

57

Adapter: un semplice esempio

- Decidiamo di arricchire la nostra applicazione con altre funzionalità e per nostra fortuna scopriamo due classi che potrebbero fare al nostro caso in due distinte librerie: la classe Synthesizer e la classe Human

```
class Synthesizer:
	def __init__(self, name):
		self.name = name
	def __str__(self): return 'the {} synthesizer'.format(self.name)
	def play(self): return 'is playing an electronic song’
class Human:
	def __init__(self, name): self.name = name
	def __str__(self):
		return '{} the human'.format(self.name)
	def speak(self):
		return 'says hello'
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

58

Adapter: un semplice esempio

- Poniamo le due classi in un modulo separato.
- Problema: il client sa solo che può invocare il metodo execute() e non ha alcuna idea dei metodi play() o speak().
- Come possiamo far funzionare il codice senza modificare le classi Synthesizer e Human?
- Creiamo una classe generica Adapter che ci permetta di unificare oggetti di diverse interfacce in un'unica interfaccia

Adapter: un semplice esempio

- L'argomento obj del metodo__init__() è l'oggetto che vogliamo adattare
- adapted_methods è un dizionario che contiene le coppie chiave/valore dove la chiave è il metodo che il client invoca e il valore è il metodo che dovrebbe essere invocato.

```
class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)
    def __str__(self):
        return str(self.obj)
```

Adapter: un semplice esempio

objects è la lista di tutti gli oggetti. L'istanza di computer viene aggiunta alla lista senza adattamenti. Gli oggetti incompatibili (istanze di Human o Synthesizer) sono prima adattate usando la classe Adapter. Il client può usare execute() su tutti gli oggetti senza essere a conoscenza delle differenza tra le classi usate.

```
def main():
    objects = [Computer('Asus')]
    synth = Synthesizer('moog')
    objects.append(Adapter(synth, dict(execute=synth.play) ))
    human = Human('Bob')
    objects.append(Adapter(human, dict(execute=human.speak)))
    for i in objects:
        print('{} {}'.format(str(i), i.execute()))
if __name__ == "__main__": main()
```

```
>>> python3 adapter.py
the Asus computer executes a program
the moog synthesizer is playing an electronic song
Bob the human says hello
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

61

Adapter: un'implementazione che usa l'ereditarietà (1)

```
class WhatIHave:
        def g(self): pass
        def h(self): pass
class WhatIWant:
        def f(self): pass
class Adapter(WhatIWant):
        def __init__(self, whatIHave):
                self.whatIHave = whatIHave
        def f(self):
                # Implement behavior
                #using  methods in WhatIHave:

                self.whatIHave.g()
                self.whatIHave.h()
```

```
class WhatIUse:
        def op(self, whatIWant):
                whatIWant.f()


whatIUse = WhatIUse()
whatIHave = WhatIHave()
adapt= Adapter(whatIHave)
whatIUse.op(adapt)          #riceve la classe derivata
                            #da WhatIwant
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

62

Adapter: un'implementazione che usa l'ereditarietà (1) – esempio Computer

```
class Human:
        ...
class Synthesizer:
       ...
class Computer:
        ...
class Adapter(Computer):
        def __init__(self, wih):
                self.wih=wih
        def execute(self):
                if isinstance(self.wih, Synthesizer):
                        return self.wih.play()
                if isinstance(self.wih, Human):
                        return self.wih.speak()
```

```
class WhatIUse:
        def op(self, comp):
                return comp.execute()

whatIUse = WhatIUse()
human = Human('Bob')
adapt= Adapter(human)
print(whatIUse.op(adapt))
```


Adapter: un'altra implementazione che usa l'ereditarietà (2)

```
class WhatIHave:
        def g(self): pass
        def h(self): pass
class WhatIWant:
        def f(self): pass
class Adapter(WhatIWant):
        def __init__(self, whatIHave):
                self.whatIHave = whatIHave
        def f(self):
                # Implement behavior
                #using  methods in WhatIHave:

                self.whatIHave.g()
                self.whatIHave.h()
```

```
class WhatIUse:
        def op(self, whatIWant):
                whatIWant.f()


# build adapter use into op():
class WhatIUse2(WhatIUse):
        def op(self, whatIHave):
                Adapter(whatIhave).f()

whatIUse2 = WhatIUse2()
whatIHave = WhatIHave()
whatIUse2.op(whatIHave)
```

prime 4 classi stesse di prima

Adapter: un'implementazione che usa l'ereditarietà (2) - esempio Computer

```
class Human:
    ...
class Synthesizer:
    ...
class Computer:
    ...
class Adapter(Computer):
    def __init__(self, wih):
        self.wih=wih
    def execute(self):
        if isinstance(self.wih, Synthesizer):
            return self.wih.play()
        if isinstance(self.wih, Human):
            return self.wih.speak()
```

```
class WhatIUse:
    def op(self, comp):
        return comp.execute()

class WhatIUse2(WhatIUse):
    def op(self, wih):
        return Adapter(wih).execute()


whatIUse2 = WhatIUse2()
human = Human('Bob')
print(whatIUse2.op(human))
```


Adapter: ancora un'altra implementazione che usa l'ereditarietà (3)

```
class WhatIHave:
    def g(self): pass
    def h(self): pass
class WhatIWant:
    def f(self): pass
```

```
class WhatIUse:
    def op(self, whatIWant):
        whatIWant.f()

# build adapter into WhatIHave:
class WhatIHave2(WhatIHave, WhatIWant):
    def f(self):
        self.g()
        self.h()


whatIUse = WhatIUse()
whatIhave2=WhatIHave2()
whatIUse.op(whatIHave2)
```

prime 3 classi stesse di prima

Adapter: un'implementazione che usa l'ereditarietà (3) - esempio Computer

```
class Human:
        ...
class Synthesizer:
        ...
class Computer:
        ...
```

```
class WhatIUse:
        def op(self, comp):
                return comp.execute()

class Human2(Human, Computer):
        def execute(self):
                return self.speak()


class Synthesizer2(Synthesizer, Computer):
        def execute(self):
                return self.play()

whatIUse = WhatIUse()
human2 = Human2('Bob')
print(whatIUse.op(human2))
```


Adapter: ancora un'altra implementazione che usa l'ereditarietà (4)

```
class WhatIHave:
        def g(self): pass
        def h(self): pass
class WhatIWant:
        def f(self): pass
```

prime 3 classi stesse di prima

```
class WhatIUse:
        def op(self, whatIWant):
                whatIWant.f()
# inner adapter:
class WhatIHave3(WhatIHave):
        class InnerAdapter(WhatIWant):
                def __init__(self, outer):
                        self.outer = outer
                def f(self):
                        self.outer.g()
                        self.outer.h()
        def whatIWant(self):
                return WhatIHave3.InnerAdapter(self)

whatIUse = WhatIUse()
human=WhatIHave3()
whatIUse.op(whatIHave3.whatIWant())
```

Adapter: un'implementazione che usa l'ereditarietà (4) - esempio Computer

```
class Human:
        ...
class Synthesizer:
        ...
class Computer:
        ...
class WhatIUse:
        def op(self, comp):
                return comp.execute()
```

```
class Human3(Human):
    #adattatore interno
    class InnerAdapter(Computer):
        def __init__(self, outer):
            self.outer = outer
        def execute(self):
            return self.outer.speak()
    def whatIWant(self):
        return Human3.InnerAdapter(self)

class Synthesizer3(Synthesizer):
        ...

whatIUse = WhatIUse()
human3=Human3('Bob')
print(whatIUse.op(human3.whatIWant()))
```


Adapter: un'implementazione che usa l'ereditarietà (4 bis) - esempio Computer

```
class Human:
        ...
class Synthesizer:
        ...
class Computer:
        ...
class WhatIUse:
        def op(self, comp):
                return comp.execute()
```

```
def createClass(cls):
        class hs(cls):
                def execute(self):
                        if isinstance(self,Human) :
                                return self.speak()
                        else: return self.play()

        return hs

whatIUse = WhatIUse()
myclass=createClass(Human) #myclass estende Human e Computer
human2 = myclass('Bob') #instanza di myclass di nome Bob
print(whatIUse.op(human2)) #invoca human2.speak()
```

Esercizio: posso rendere createClass un decoratore di classe? Se si` , modificate il codice.

Design Pattern Proxy

- Proxy e` 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.
- Proxy è un caso particolare del design pattern state
- Quando un oggetto surrogato è creato, viene fornita un'implementazione alla quale vengono inviate tutte le chiamate dei metodi

Design Pattern Proxy

- La differenza tra Proxy e State (la vedremo poi) consiste semplicemente nel fatto che Proxy ha un'unica implementazione mentre State ne ha più di una.
- Da un punto di vista applicativo, Proxy è usato per controllare l'accesso alla sua implementazione mentre State permette di cambiare dinamicamente la sua implementazione

Design Pattern Proxy

Gli usi di Proxy sono:

1. **Remote proxy**. E` 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**. E` un proxy che fornisce una “lazy initialization” per creare oggetti costosi su richiesta solo se se sono realmente necessari.

3. **Protection proxy**. E` un proxy usato quando vogliamo che il programmatore lato client non abbia pieno accesso all’oggetto.

4. **Smart reference**. E` 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.** E` comunque conveniente avere una interfaccia comune in modo che **Implementation** sia forzata a fornire tutti i metodi che **Proxy** ha bisogno di invocare.
- Python fornisce un meccanismo di delega built-in (prossimo esempio).

Design Pattern Proxy

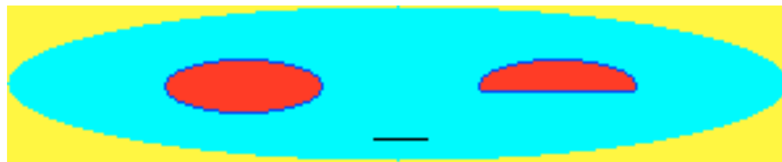
L'uso di **__getattr__( )** rende **Proxy2** completamente generica e non legata ad una particolare implementazione

```
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 passata a __init__ che supporta l'interfaccia 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 e i rimanti 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.
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.

```
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
```

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 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.