Programmazione Avanzata

Design Pattern IV

Il Design Pattern State

Il Design Pattern State consente ad un oggetto di modificare il proprio comportamento quando il suo stato interno cambia

Utile nei seguenti casi:

- Il comportamento di un oggetto dipende dal suo stato e deve cambiare comportamento durante l'esecuzione del programma in base al suo stato
- Le operazioni contengono statement condizionali grandi che dipendono dallo stato dell'oggetto. Lo stato dell'oggetto è di solito rappresentato da una o più costanti numerate. Il pattern State inserisce ciascun caso dello statement condizionale in una classe separata.
  - **Ciò consente di trattare lo stato dell'oggetto come un vero e proprio oggetto che può cambiare indipendentemente da altri oggetti.**

Il Design Pattern State: un esempio

Consideriamo una classe multiplexer che ha due stati che influiscono sul comportamento dei metodi della classe.

Quando è **attivo**, il multiplexer accetta connessioni, cioè coppie (nome evento, callback), dove callback è un qualsiasi callable. Dopo che sono state stabilite le connessioni, ogni volta che viene inviato un evento al multiplexer, i callback associati vengono invocati.

Quanto il multiplexer è **dormiente**, l'invocazione dei suoi metodi non ha alcun effetto (comportamento safe)

Nell'esempio vengono create delle funzioni callback che contano il numero di eventi che ricevono. Queste funzioni vengono connesse ad un multiplexer attivo. Poi vengono inviati un certo numero di eventi random al multiplexer e stampati i conteggi tenuti dalle funzioni callback.

Il Design Pattern State: un esempio

- Dopo aver inviato 100 eventi random al multiplexer attivo, lo stato del multiplexer viene cambiato in dormiente e gli vengono inviati altri 100 eventi random, ciascuno dei quali deve essere ignorato.
- Il multiplexer è quindi riportato nello stato attivo e gli vengono inviati altri eventi ai quali il multiplexer deve rispondere invocando i callback associati.
- si veda anche il codice completo in multiplexer1.py

```
$ ./multiplexer1.py
After 100 active events:  cars=150 vans=42 trucks=14 total=206
After 100 dormant events: cars=150 vans=42 trucks=14 total=206
After 100 active events:  cars=303 vans=83 trucks=30 total=416
```

output del programma

Il Design Pattern State: un esempio

- Il main() comincia con il creare dei contatori. **Le istanze così create sono callable** e quindi possono essere usate come funzioni.
  - Le istanze di Counter mantengono contatori separati per ciascuno dei nomi passati come argomento o, in assenza di un nome (come totalCounter), matengono un contatore singolo.
- Viene quindi creato un multiplexer (che per default è attivo) e vengono connesse le funzioni callback agli eventi.
- I nomi degli eventi considerati sono "cars", "vans" e "trucks".
  - Nel for, la funzione carCounter() è connessa all'evento "cars", la funzione commercialCounter() è connessa agli eventi "vans" e "trucks" e totalCounter() è connessa a tutti e tre gli eventi.

```
totalCounter = Counter()
carCounter = Counter("cars")
commercialCounter = Counter("vans", "trucks")

multiplexer = Multiplexer()
for eventName, callback in (("cars", carCounter),
        ("vans", commercialCounter), ("trucks", commercialCounter)):
    multiplexer.connect(eventName, callback)
    multiplexer.connect(eventName, totalCounter)
```

36

Il Design Pattern State: un esempio

- Con il codice mostrato in basso, main() genera 100 eventi random e li invia al multiplexer.

- Per un evento "cars", il multiplexer invoca carCounter() e totalCounter(), passando l'evento come unico argomento a ciascuna chiamata. Se l'evento è invece "vans" o "trucks", il multiplexer invoca le funzioni commercialCounter() e totalCounter().

```
for event in generate_random_events(100):
    multiplexer.send(event)
print("After 100 active events:  cars={} vans={} trucks={} total={}"
        .format(carCounter.cars, commercialCounter.vans,
                commercialCounter.trucks, totalCounter.count))
```


37

Il Design Pattern State: un esempio

Il metodo __init__() della classe Counter:

- Se non vengono forniti nomi, viene creata un'istanza di un contatore anonimo il cui conteggio è mantenuto in self.count;
- in caso contrario, vengono mantenuti conteggi indipendenti mediante la funzione built-in setattr() per ciascuno dei nomi passati a __init__().
- Ad esempio, per l'istanza carCounter viene creato l'attributo self.cars.

```
class Counter:

    def __init__(self, *names):
        self.anonymous = not bool(names)
        if self.anonymous:
            self.count = 0
        else:
            for name in names:
                if not name.isidentifier():
                    raise ValueError("names must be valid identifiers")
                setattr(self, name, 0)
```


Il Design Pattern State: un esempio

- Il metodo __call__() della classe Counter
- Quando un'istanza di Counter è invocata, la chiamata è passata a __call__()
- Se il contatore è anonimo, self.count viene incrementato; altrimenti si cerca di recuperare l'attributo corrispondente al nome dell'evento.
  - Ad esempio, se il nome dell'evento è "trucks", count viene settato con il valore di self.trucks. Viene quindi aggiornato il valore dell'attributo con il vecchio conteggio più il nuovo conteggio dell'evento.
- Siccome non è fornito un valore di default per la funzione built-in gettattr(), se l'attributo non esiste viene lanciato un AttributeError. Ciò assicura anche che non venga creato un attributo con un nome sbagliato perché in caso di errore la chiamata a setattr() non viene raggiunta.

```
    def __call__(self, event):
        if self.anonymous:
            self.count += event.count
        else:
            count = getattr(self, event.name)
            setattr(self, event.name, count + event.count)
```

Il Design Pattern State: un esempio

La classe Event è molto semplice perché serve esclusivamente come parte dell'infrastruttura per illustrare il Pattern State.

```
class Event:

    def __init__(self, name, count=1):
        if not name.isidentifier():
            raise ValueError("names must be valid identifiers")
        self.name = name
        self.count = count
```


Il Design Pattern State: un esempio

La classe Multiplexer:

- Ci sono due approcci principali che possiamo utilizzare per gestire lo stato interno ad una classe.
  - Approccio che usa **metodi state-sensitive** (comportamento dei metodi si adatta allo stato)
  - Approccio che usa **metodi state-specific** (progettati ad hoc per specifici stati)
- Consideriamo il primo dei due approcci:

```
class Multiplexer:

    ACTIVE, DORMANT = ("ACTIVE", "DORMANT")

    def __init__(self):
        self.callbacksForEvent = collections.defaultdict(list)
        self.state = Multiplexer.ACTIVE
```

Il Design Pattern State: un esempio

- La classe collections.defaultdict (https://docs.python.org/3.7/library/collections.html#collections.defaultdict):
- La classe defaultdict è una sottoclasse di dict.
- Il costruttore riceve come primo argomento un valore per il suo attributo default_factory (per default è None), usato per creare valori non presenti nel dizionario.
- I restanti argomenti corrispondono a quelli passati al costruttore di dict.
- Se default_factory non è None, esso viene invocato senza argomenti per fornire un valore di default per una data chiave k. Tale valore viene inserito nel dizionario (associato alla chiave k) e restituito.
- Questo metodo è invocato dal metodo __getitem__() quando la chiave richiesta non viene trovata.
- Gli altri metodi non invocano default_factory per cui get() restituisce None se la chiave non è nel dizionario.

42

Il Design Pattern State: un esempio

- La classe Multiplexer ha due possibili stati: ACTIVE e DORMANT.
  - ACTIVE: i metodi state-sensitive svolgono un lavoro utile
  - DORMANT: i metodi state-sensitive non fanno niente.
- Un nuovo Multiplexer è creato nello stato ACTIVE.
- self.callbacksForEvent è un dizionario (di tipo defaultdict) di coppie (nome evento, lista di callable)
- Il metodo connect è usato per creare un'associazione tra un evento con un certo nome e un callback.
- Se il nome dell'evento non è nel dizionario, il fatto che self.callbacksForEvent sia un defaultdict garantisce che venga creato un elemento con chiave uguale al nome dell'evento e con valore uguale ad una lista vuota che verrà poi restituita.
- Se il nome dell'evento è già nel dizionario, verrà restituita la lista associata.
- In entrambi i casi , con append() viene poi aggiunta alla lista il callback da associare all'evento

```
    def connect(self, eventName, callback):
        if self.state == Multiplexer.ACTIVE:
            self.callbacksForEvent[eventName].append(callback)
```

43

Il Design Pattern State: un esempio

Se invocato senza specificare un callback, questo metodo disconnette tutti i callback associati con il nome dell'evento dato; altrimenti rimuove solo il callback specificato dalla lista dei callback associata al nome dell'evento.

```
    def disconnect(self, eventName, callback=None):
        if self.state == Multiplexer.ACTIVE:
            if callback is None:
                del self.callbacksForEvent[eventName]
            else:
                self.callbacksForEvent[eventName].remove(callback)
```


Il Design Pattern State: un esempio

Se un evento è inviato al multiplexer e questo è attivo allora send itera su tutti i callback associati all'evento (se ve ne sono) e invoca ciascuno di questi callback passandogli l'evento come argomento.

```
    def send(self, event):
        if self.state == Multiplexer.ACTIVE:
            for callback in self.callbacksForEvent.get(event.name, ()):
                callback(event)
```

Il Design Pattern State: un esempio

Approccio basato su metodi state-specific:

- La classe Multiplexer ha gli stessi due stati di prima e lo stesso metodo __init__. Questa volta però l'attributo self.state è una proprietà.

- Questa versione di multiplexer non immagazzina lo stato come tale ma lo computa controllando se uno dei metodi pubblici è stato settato ad un metodo privato attivo o passivo.

```
@property
def state(self):
    return (Multiplexer.ACTIVE if self.send == self.__active_send
            else Multiplexer.DORMANT)
```


Il Design Pattern State: un esempio

- Ogni volta che viene cambiato lo stato, il setter della proprietà associa il multiplexer ad un insieme di metodi appropriati al suo stato
- Se, ad esempio, lo stato è DORMANT, ai metodi pubblici viene assegnata la versione lambda dei metodi

```
@state.setter
def state(self, state):
    if state == Multiplexer.ACTIVE:
        self.connect = self.__active_connect
        self.disconnect = self.__active_disconnect
        self.send = self.__active_send
    else:
        self.connect = lambda *args: None
        self.disconnect = lambda *args: None
        self.send = lambda *args: None
```

Il Design Pattern State: un esempio

- _active_connect è un metodo privato che può essere assegnato al corrispondente metodo pubblico self.connect se lo stato del multiplexer è ACTIVE. I metodi _active_disconnect e _active_ send sono simili.
  - **Nessuno di questi tre metodi controlla lo stato dell'istanza.**

```
def __active_connect(self, eventName, callback):
    self.callbacksForEvent[eventName].append(callback)
```


Il Design Pattern State: un ulteriore esempio

State è simile a Proxy ma a differenza di Proxy utilizza più implementazioni ed un metodo per passare da un'implementazione all'altra durante la vita del surrogato.

```
class State_d:
    def __init__(self, imp):
        self.__implementation = imp
    def changeImp(self, newImp):
        self.__implementation = newImp
    # Delegate calls to the implementation:
    def __getattr__(self, name):
        return getattr(self.__implementation, name)

class Implementation1:
    def f(self):
        print("Fiddle de dum, Fiddle de dee,")
    def g(self):
        print("Eric the half a bee.")
    def h(self):
        print("Ho ho ho, tee hee hee,")

class Implementation2:
    def f(self):
        print("We're Knights of the Round Table.")
    def g(self):
        print("We dance whene'er we're able.")
    def h(self):
        print("We do routines and chorus scenes")
```


#Uso di State_d

```
def run(b):
    b.f()
    b.g()
    b.h()
    b.g()

b = State_d(Implementation1())
run(b)
b.changeImp(Implementation2())
run(b)
```

Il Design Pattern Mediator

- Il Design Pattern Mediator e` un design pattern comportamentale. Mediator fornisce un mezzo per creare un oggetto che incapsula le interazioni tra altri oggetti.
- Cio` consente di stabilire relazioni tra oggetti che non hanno conoscenza diretta l'uno dell'altro.
- Per esempio se si verifica un evento che richiede l'attenzione di alcuni oggetti, tale evento sara` comunicato al mediatore che mandera` una notifica agli oggetti interessati.

Il Design Pattern Mediator: un esempio

- Vogliamo creare delle form contenenti text widget e button widget
- Cio` e` di grande utilita` nella programmazione GUI.
- L'interazione tra i widget della form sara` gestita da un mediator
- La classe Form fornisce i metodi create_widgets() e create_mediator()

```
class Form:

    def __init__(self):
        self.create_widgets()
        self.create_mediator()
```

Il Design Pattern Mediator: un esempio

- La form ha
  - due widget per inserire testo: una per il nome dell'utente, l'altra per l'indirizzo email
  - due bottoni: OK e CANCEL

```
    def create_widgets(self):
        self.nameText = Text()
        self.emailText = Text()
        self.okButton = Button("OK")
        self.cancelButton = Button("Cancel")
```


Il Design Pattern Mediator: un esempio

- **Ogni form ha un singolo mediator associato**.
- Il metodo __init__() di Mediator riceve come argomenti una o piu` coppie (widget, callable), ciascuna delle quali descrive una relazione che il mediatore deve supportare.
- Nel codice riportato di seguito, le coppie passate al mediatore fanno in modo che se cambia il testo di uno dei widget per l'inserimento di testo allora viene invocato il metodo Form.update_ui(); mentre se viene cliccato uno dei bottoni allora viene invocato il metodo Form.clicked().
- Dopo aver creato il mediatore, viene invocato il metodo update_ui() per inizializzare la form.

```
    def create_mediator(self):
        self.mediator = Mediator(((self.nameText, self.update_ui),
                (self.emailText, self.update_ui),
                (self.okButton, self.clicked),
                (self.cancelButton, self.clicked)))
        self.update_ui()
```

Il Design Pattern Mediator: un esempio

- Questo metodo abilita il bottone OK se entrambi i widget per inserire testo contengono del testo; altrimenti disabilita il bottone.
- Questo metodo viene invocato ogni volta che cambia il testo in uno dei due widget.

```
    def update_ui(self, widget=None):
        self.okButton.enabled = (bool(self.nameText.text) and
                                 bool(self.emailText.text))
```

- Questo altro metodo di Form viene invocato ogni volta che viene cliccato un bottone.
- In questo esempio il metodo si limita a stampare OK o Cancel ma nelle applicazioni reali ovviamente compie azioni piu` interessanti.

```
    def clicked(self, widget):
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
```


Il Design Pattern Mediator: un esempio

- Il metodo __init__() della classe Mediator
  - crea un dizionario di tipo defaultdict le cui chiavi sono widget e i cui valori sono liste di uno o piu` callable
  - Il for considera le coppie (widget, callable) presenti nella tupla passata come secondo argomento. Per ciascuno dei widget in queste coppie, quando si cerca di accedere per la prima volta all' item del dizionario con chiave uguale a quel widget, siccome il widget non e` ancora presente nel dizionario, viene inserito nel dizionario un elemento con chiave uguale al widget e valore uguale ad una lista vuota che viene poi restituita in output. Se si accede successivamente a quell'item, viene semplicemente restituita la lista associata al widget nel dizionario.
  - Il metodo append aggiunge poi alla lista il caller associato al widget nella coppia considerata.
  - Alla fine viene settato (ed eventualmente creato) l'attributo mediator del widget in modo che contenga il mediator appena creato.

```
class Mediator:

    def __init__(self, widgetCallablePairs):
        self.callablesForWidget = collections.defaultdict(list)
        for widget, caller in widgetCallablePairs:
            self.callablesForWidget[widget].append(caller)
            widget.mediator = self
```

Il Design Pattern Mediator: un esempio

- Ogni volta che un oggetto mediato (cioe` un widget passato a Mediator) cambia stato esso invoca il seguente metodo di Mediator che si occupa di invocare ogni metodo associato al widget.

```
    def on_change(self, widget):
        callables = self.callablesForWidget.get(widget)
        if callables is not None:
            for caller in callables:
                caller(widget)
        else:
            raise AttributeError("No on_change() method registered for {}"
                    .format(widget))
```


56

Il Design Pattern Mediator: un esempio

- Questa e` una classe usata come classe base per le classi mediate.
- Le istanze della classe mantengono un riferimento all'oggetto mediatore
- Il metodo Mediated.on_change() invoca il metodo on_change() del mediatore passandogli il widget mediato su cui e` stato invocato il metodo Mediated.on_change.
- Siccome questa classe non e` modificata dalle sue sottoclassi, essa rappresenta un esempio in cui e` possibile rimpiazzare la classe base con un decoratore di classe

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```


57

Il Design Pattern Mediator: un esempio

- La classe Button estende Mediated e di conseguenza un oggetto bottone ha l'attributo self.mediator e il metodo on_change che viene invocato quando il bottone cambia stato (ad esempio quando viene cliccato.
- In questo esempio, un'invocazione di Button.click() provoca un'invocazione di Button.on_change() (ereditato da Mediated), che a sua volta causa un'invocazione del metodo on_change() del mediatore.
  - Il metodo on_change() del mediatore invochera` i metodi associati al bottone. In questo caso, viene invocato il metodo Form.clicked() con il bottone stesso come argomento di tipo widget.

```
class Button(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.enabled = True
        self.text = text

    def click(self):
        if self.enabled:
            self.on_change()
```


58

Il Design Pattern Mediator: un esempio

- La classe Text ha la stessa struttura di Button.
- Per ogni widget (button widget, text widget, ecc.), il fatto di definire la classe corrispondente come sottoclasse di Mediated permette di lasciare al mediatore il compito di occuparsi delle azioni legate ad un cambio di stato del widget.
- Ovviamente quando si crea il mediatore occorre stabilire le associazioni tra i widget e i metodi che vogliamo vengano invocati

```
class Text(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.__text = text

    @property
    def text(self):
        return self.__text

    @text.setter
    def text(self, text):
        if self.text != text:
            self.__text = text
            self.on_change()
```


59

Il Design Pattern Mediator: un esempio

```
def main():
    form = Form()
    test_user_interaction_with(form)

def test_user_interaction_with(form):
    form.okButton.click()         # Ignored because it is disabled
    print(form.okButton.enabled)    # False
    form.nameText.text = "Fred"
    print(form.okButton.enabled)    # False
    form.emailText.text = "fred@bloggers.com"
    print(form.okButton.enabled)    # True
    form.okButton.click()          # OK
    form.emailText.text = ""
    print(form.okButton.enabled)    # False
    form.cancelButton.click()      # Cancel

if __name__ == "__main__":
    main()
```

Un programma che crea e usa una form

Il Design Pattern Mediator: un esempio basato su coroutine

- Un mediatore si presta ad un'implementazione mediante coroutine perche' puo` essere visto come una pipeline che riceve messaggi (derivanti da invocazioni di on_change()) e passa questi messaggi agli oggetti interessati
- In questo esempio viene implementato un mediator mediante coroutine per lo stesso problema considerato nell'esempio precedente.
- A differenza di quanto accadeva prima, in questa implementazione ogni widget e` associato ad un mediatore che e` una pipeline di coroutine (prima il mediatore era un oggetto associato all'intera form e tutti i widget della form erano associati insieme ai rispettivi callable al mediatore).
  - Ogni volta che un widget cambia stato (ad esempio, viene cliccato un bottone), esso invia se stesso alla pipeline.
  - Sono le componenti della pipeline a decidere se vogliono svolgere o meno azioni in risposta al cambio di stato del widget.
- Nell'approccio precedente il metodo on_change() del mediatore invoca i metodi associati al widget nel caso in cui il widget cambia stato
- Il codice non illustrato e` identico a quello visto nell'esempio precedente..

Il Design Pattern Mediator: un esempio basato su coroutine

- Non abbiamo bisogno di una classe Mediator in quanto il mediator e` di fatto una pipeline di coroutine
- Il metodo in basso crea una pipeline di coroutine di due componenti, self._update_ui_mediator() e self._clicked_mediator().
- Una volta creata la pipeline, l'attributo mediator della pipeline viene settato con questa pipeline.
- Alla fine, viene inviato None alla pipeline e siccome nessun widget e` None, nessuna azione specifica sara` intrapresa ad eccezione di azioni che interessano la form (come per esempio abilitare o disabilitare il bottone OK in _update_ui_mediator()).

```
def create_mediator(self):
    self.mediator = self._update_ui_mediator(self._clicked_mediator())
    for widget in (self.nameText, self.emailText, self.okButton,
            self.cancelButton):
        widget.mediator = self.mediator
    self.mediator.send(None)
```


Il Design Pattern Mediator: un esempio basato su coroutine

- Questa coroutine e` parte della pipeline.
- Ogni volta che un widget notifica un cambio di stato, il widget passato alla pipeline e ` restituito dall'espressione yield e salvato nella variabile widget.
- Quando occorre abilitare o disabilitare un bottone, questo viene fatto indipendentemente da quale widget abbia cambiato stato.
  - Potrebbe non essere cambiato lo stato di nessun widget, che il widget sia None, e quindi che la form sia stata semplicemente inizializzata.
  - Dopo aver settato il campo enabled del bottone, la coroutine passa il widget alla chain

```
@coroutine
def _update_ui_mediator(self, successor=None):
    while True:
        widget = (yield)
        self.okButton.enabled = (bool(self.nameText.text) and
                                 bool(self.emailText.text))
        if successor is not None:
            successor.send(widget)
```

Il Design Pattern Mediator: un esempio basato su coroutine

- Questa coroutine si occupa solo dei click dei bottoni Ok e Cancel
- Se uno di questi bottoni e` il widget che ha cambiato stato allora questa coroutine gestisce il cambio di stato altrimenti passa il widget alla prossima coroutine nella pipeline, se ve ne e` una.

```
@coroutine
def _clicked_mediator(self, successor=None):
    while True:
        widget = (yield)
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
        elif successor is not None:
            successor.send(widget)
```


64

Il Design Pattern Mediator: un esempio basato su coroutine

- Le classi Text e Button sono le stesse dell'implementazione basata sull'approccio convenzionale.
- La classe Mediated e` lievemente diversa in quanto il suo metodo on_change() invia il widget che ha cambiato stato alla pipeline.

```
class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.send(self)
```


65

Il Design Pattern Template Method

- Il pattern Template Method e un design pattern comportamentale che permette di definire i passi di un algoritmo lasciando alle sottoclassi il compito di definire alcuni di questi passi.
- Vediamo un esempio in cui viene creata una classe astratta AbstractWordCounter class che fornisce due metodi.
  - il primo metodo, can_count(*filename*), restituisce un valore Booleano che indica se la classe puo` contare le parole del file dato in base all'estensione del file.
  - Il secondo metodo, count(*filename*), restituisce un conteggio di parole.
- Il codice comprende anche due sottoclassi, una che conta le parole in file di testo e l'altro per contare le parole in file HTML.

Il Design Pattern Template Method

- Tutti i metodi sono statici per cui non si ha mai a che fare con istanze della classe
- Il metodo count_words (esterna rispetto alla classi) itera su due oggetti classe (sottoclassi della classe atratta)
- Se una delle due classi puo` contare le parole nel file passato a count_words allora viene effettuato il conteggio e questo viene restituito dalla funzione.
- Se nessuna delle due classi e` in grado di contare le parole del file. il metodo restituisce implicitamente None per indicare che non e` stato in grado di effettuare il conteggio.

```
def count_words(filename):
    for wordCounter in (PlainTextWordCounter, HtmlWordCounter):
        if wordCounter.can_count(filename):
            return wordCounter.count(filename)
```

Il Design Pattern Template Method

- Di seguito sono mostrati due diversi codici per la classe astratta AbstractWordCounter.
- Questa classe fornisce i metodi che devono essere implementati nelle eventuali sottoclassi.

```
class AbstractWordCounter:

    @staticmethod
    def can_count(filename):
        raise NotImplementedError()

    @staticmethod
    def count(filename):
        raise NotImplementedError()
```

```
class AbstractWordCounter(
        metaclass=abc.ABCMeta):

    @staticmethod
    @abc.abstractmethod
    def can_count(filename):
        pass

    @staticmethod
    @abc.abstractmethod
    def count(filename):
        pass
```


Il Design Pattern Template Method

- Questa sottoclasse implementa il contatore per i file testuali e assume che i file con estensione .txt siamo codificati con UTF-8 (o 7-bit ASCII, che e` un sottoinsieme di UTF-8).

re.finditer(*pattern*, *string*, *flags=0*) scandice string da sinistra a destra e restituisce i match (rispetto all'espressione regolare pattern nell'ordine in cui li trova.

```
class PlainTextWordCounter(AbstractWordCounter):

    @staticmethod
    def can_count(filename):
        return filename.lower().endswith(".txt")

    @staticmethod
    def count(filename):
        if not PlainTextWordCounter.can_count(filename):
            return 0
        regex = re.compile(r"\w+")
        total = 0
        with open(filename, encoding="utf-8") as file:
            for line in file:
                for _ in regex.finditer(line):
                    total += 1
        return total
```

IL Design Pattern Observer

- Il pattern Observer e` un design pattern comportamentale che supporta relazioni di dipendenza many-to-many tra oggetti in modo tale che quando un oggetto cambia stato tutti gli oggetti collegati sono informati del cambio.

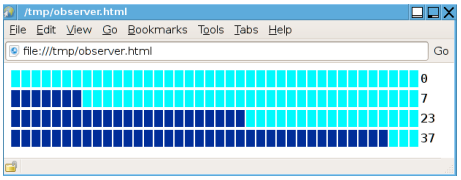
IL Design Pattern Observer:un esempio

- Uno degli esempi di questo pattern e delle sue varianti e` il paradigma model/view/controller (MVC) che consiste nel separare un'applicazione in tre componenti logiche: modello, view e controller.
  - Il modello gestisce i dati e la logica dell'applicazione indipendentemente dall'interfaccia utente, una o piu` view visualizzano i dati in una o piu` forme comprensibili per l'utente ed uno o piu` controller mediano tra input e modello, cioe` converte l'input in comandi per il modello o le view. Ogni cambio nel modello si riflette automaticamente nelle view associate.
- Una popolare semplificazione dell'approccio MVC consiste nell'usare un paradigma model/view dove le view si occupano sia di visualizzare i dati sia di mediare tra input e modello.
  - In termini di Observer Pattern cio` significa che le view sono osservatori del modello e il modello e` l'oggetto dell'osservazione.

IL Design Pattern Observer: un esempio

- Consideriamo un modello che rappresenta un valore con un minimo e un massimo, come ad esempio una scrollbar o un controllo della temperatura.
- Vengono creati due osservatori (view) separati per il modello: uno per dare in output il valore del modello ogni volta che esso cambia sotto forma di una barra di progressione in formato HTML, l'altro per mantenere la storia dei cambiamenti (valori e timestamp).

```
$ ./observer.py > /tmp/observer.html
  0 2013-04-09 14:12:01.043437
  7 2013-04-09 14:12:01.043527
 23 2013-04-09 14:12:01.043587
 37 2013-04-09 14:12:01.043647
```



72

IL Design Pattern Observer: un esempio

- La classe Observed e` estesa dai modelli o da ogni altra classe che supporta l'osservazione.
- La classe Observed mantiene un insieme di oggetti osservatori.
  - Ogni volta che viene aggiunto un oggetto osservatore all'oggetto osservato, il metodo update() dell'osservatore e` invocato per inizializzare l'osservatore con lo stato attuale del modello.
  - Se in seguito il modello cambia stato, esso invoca il metodo ereditato observers_notify() in modo tale che il metodo update() di ogni osservatore possa essere invocato per assicurare che ogni osservatore (view) rappresenti il nuovo stato del modello.

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)

    def observer_discard(self, observer):
        self.__observers.discard(observer)

    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)
```

73

IL Design Pattern Observer: un esempio

- Il metodo observers_add()
  - accetta uno o piu` osservatori da aggiungere. Per questo motivo oltre a *observer c'e` il parametro observer che assicura che il numero di osservatori passati in input al metodo non sia zero.
  - usa nel for il metodo intertools.chain(*iterables) che crea un iteratore che restituituisce gli elementi dall'oggetto iterabile specificato come primo argomento e quando non ci sono piu` elementi da restituire in questa lista, passa alla prossima collezione iterabile e cosi` via fino a che non vengono restituiti gli elementi di tutte le collezioni iterabili in iterables. Il for avrebbe potuto usare la concatenazione di tuple in questo modo "for observer in (observer,) + observers:"

```
class Observed:

    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)

    def observer_discard(self, observer):
        self.__observers.discard(observer)

    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)
```

74

IL Design Pattern Observer: un esempio

- La classe SliderModel eredita dalla classe Observed un insieme privato di osservatori che inizialmente e` vuoto e i metodi observers_add(), observer_discard() e observers_notify()
- Quando lo stato del modello cambia, per esempio quando il suo valore cambia, esso deve invocare il metodo observers_notify() in modo che ciascun osservatore possa rispondere di conseguenza.
- SliderModel ha anche le proprieta` minimum e maximum i cui setter, come quello di value, invocano il metodo observers_notify().

```
class SliderModel(Observed):

    def __init__(self, minimum, value, maximum):
        super().__init__()
        # These must exist before using their property setters
        self.__minimum = self.__value = self.__maximum = None
        self.minimum = minimum
        self.value = value
        self.maximum = maximum

    @property
    def value(self):
        return self.__value

    @value.setter
    def value(self, value):
        if self.__value != value:
            self.__value = value
            self.observers_notify()
    ...
```


75

IL Design Pattern Observer: un esempio

- HistoryView e` un osservatore del modello e per questo fornisce un metodo update() che accetta il modello osservato come suo unico argomento (oltre self).
- Ogniqualvolta il metodo update() e` invocato, esso aggiunge una tupla (value, timestamp) alla sua self.data list, mantenendo in questo modo la storia di tutti i cambiamenti applicati al modello.

```
class HistoryView:

    def __init__(self):
        self.data = []

    def update(self, model):
        self.data.append((model.value, time.time()))
```


IL Design Pattern Observer: un esempio

- Questa e` un'altra view per osservare il modello. L'attributo length rappresenta il numero di celle usate per rappresentare il valore del modello in una riga della tabella HTML.
- Il metodo update viene invocato quando il modello e` osservato per la prima volta e quando viene successivamente aggiornato
  - Il metodo stampa una tabella HTML di una riga con un numero self.length di celle per rappresentare il modello. Le celle sono di colore ciano se sono vuote e blu scuro altrimenti.

```
class LiveView:

    def __init__(self, length=40):
        self.length = length
```

```
    def update(self, model):
        tippingPoint = round(model.value * self.length /
                (model.maximum - model.minimum))
        td = '<td style="background-color: {}"> </td>'
        html = ['<table style="font-family: monospace" border="0"><tr>']
        html.extend(td.format("darkblue") * tippingPoint)
        html.extend(td.format("cyan") * (self.length - tippingPoint))
        html.append("<td>{}</td></tr></table>".format(model.value))
        print("".join(html))
```

IL Design Pattern Observer: un esempio

- Il main() comincia con il creare due view, HistoryView e LiveView.
- Poi crea un modello con minimo 0, valore attuale 0 e massimo 40 e rende le due view osservatori del modello
- Non appena viene aggiunta LiveView come observer del modello, essa produce il suo primo output e non appena viene aggiutno HistoryView, esso registra il suo primo valore e il suo primo timestamp.
- Poi viene aggiornato il valore del modello tre volte e ad ogni aggiornamento LiveView restituisce una nuova tabella di una riga e HistoryView registra il valore e il timestamp .

```
def main():
    historyView = HistoryView()
    liveView = LiveView()
    model = SliderModel(0, 0, 40) # minimum, value, maximum
    model.observers_add(historyView, liveView)  # liveView produces output
    for value in (7, 23, 37):
        model.value = value                     # liveView produces output
    for value, timestamp in historyView.data:
        print("{:3} {}".format(value, datetime.datetime.fromtimestamp(
                timestamp)), file=sys.stderr)
```