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
```


13

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
```


14

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

15

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

16

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

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

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

19

Il pattern Prototype

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

20

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
```


21

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
```


22

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") resituisce 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

- point1 = Point(1, 2)
  - viene semplicemente invocato il costruttore della classe Point
- 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

eval

- https://docs.python.org/3/library/functions.html#eval
- Gli argomenti sono un stringa e due argomenti opzionali *globals* and l*ocals*.
- *globals* deve essere un dizionario mentre *locals* puo` essere di un qualsiasi tipo mapping
- La stringa passato come argomento viene valutata come un'espressione Python usando i dizionari *globals* and *locals* dictionaries come namespace globali e locali.
- Se in *globals* non è presente un valore per la chiave __builtins__, un riferimento al modulo built-in builtins è associato alla chiave __builtins__ prima di fare il parsing dell'espressione.
- Il valore di default di *locals* è il dizionario *globals.*
- Se entrambi i dizionari *globals* and *locals* sono omessi l'espressione è eseguita con i *globals* and *locals* nell'ambiente in cui eval() è invocata*.*

eval

```
>>> x = 4
>>> y = 3
# valuta l'espressione x + y
>>> s = eval('x + y')
>>> print('s: ', s)
s: 7
>>> # la prossima invocazione di eval usa il parametro globals al posto del namespace globale
>>> t = eval('x + y', {'x': 5, 'y': 8})
>>> print('t: ', t)
t: 13
>>> # la prossima invocazione di eval usa il valore globale di x uguale a 5 e quello locale di y uguale a 9
>>> r = eval('x + y', {'x': 5, 'y': 8}, {'y': 9, 'w': 2})
>>> print('r: ', r)
r: 14
>>> p = eval('print(x, y)')
4 3
>>> print('p: ', p)
p: None
```


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

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

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

29

Programmazione Avanzata

Design Pattern: Flyweight

30

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. Ad esempio, una lunga lista di stringhe molte delle quali sono duplicati potrebbe richiedere molto meno spazio se al posto delle stringhe venissero memorizzati i riferimenti ad esse.

```
red, green, blue = "red", "green", "blue"
x = (red, green, blue, red, green, blue, red, green)
y = ("red", "green", "blue", "red", "green", "blue", "red", "green")
```

Nel codice in alto, x immagazzina 3 stringhe usando 8 riferimenti mentre la tupla y immagazzina 8 stringhe usando 8 riferimenti dal momento che quello che abbiamo scritto corrisponde a _anonymous_item0 = "red", …, _anonymous_item7 = "green"; y = (_anonymous_item0, …_anonymous_item7).

31

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__

32

__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__* (e di *__weakref__)* 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
```


35

__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

36

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 a punti individuali.
- 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 è sempre possibile invertire queste priorità se è conveniente farlo.

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(*filename*, *flag='c'*, *protocol=None*, *writeback=False*) 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.

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 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.

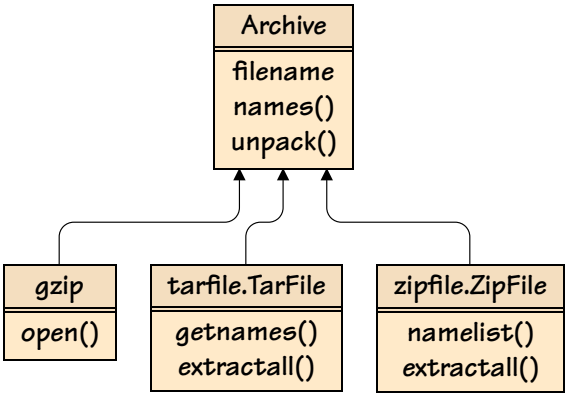
Programmazione Avanzata

Design Pattern: Facade

Il Design Pattern Facade

- Il design pattern Facade è un design pattern strutturale che fornisce un'interfaccia semplificata per un sistema costituito da interfacce o classi troppo complesse o troppo di basso livello.
- **Esempio:** La libreria standard di Python fornisce moduli per gestire file compressi gzip, tarballs e zip. Questi moduli hanno interfacce diverse.
- Immaginiamo di voler accedere ai nomi di un file di archivio ed estrarre i suoi file usando un'interfaccia semplice.
- **Soluzione:** Usiamo il design pattern Facade per fornire un'interfaccia semplice e uniforme che delega la maggior parte del vero lavoro alla libreria standard.

Il Design Pattern Facade: un esempio



47

Il Design Pattern Facade: un esempio

```
class Archive:

    def __init__(self, filename):
        self._names = None
        self._unpack = None
        self._file = None
        self.filename = filename
```

- La variabile self._names è usata per contenere un callable che restituisce una lista dei nomi dell'archivio.
- La variabile the self._unpack è usata per mantere un callable che estrae tutti i flie dell'archivio nella directory corrente.
- La variabile self._file è usata per mantenere il file object che è stato aperto per accedere all'archivio.
- self.filename è una proprietà che mantiene il nome del file di archivio.

48

Il Design Pattern Facade: un esempio

```
@property
def filename(self):
    return self.__filename

@filename.setter
def filename(self, name):
    self.close()
    self.__filename = name
```

Se l'utente cambia il filename, ad esempio archive.filename = newname, allora il file d'archivio corrente, se aperto, viene chiuso e viene aggiornata la variabile __filename.
Non viene immediatamente aperto il nuovo archivio, in quanto la classe Archive apre l'archivio solo se necessario.

Il Design Pattern Facade: un esempio

```
def close(self):
    if self._file is not None:
        self._file.close()
    self._names = self._unpack = self._file = None
```

Gli utenti della classe Archive invocano close() quando hanno finito con un'istanza.
Il metodo chiude il file object , se c'è un file object aperto, e setta self._names, self._unpack, e self._file a None per invalidarli.

La classe Archive è un context manager e così in pratica gli utenti non hanno bisogno di chiamare close(), a patto che usino la classe in uno statement with, come nel codice qui in basso:

```
with Archive(zipFilename) as archive:
    print(archive.names())
    archive.unpack()
```

Il Design Pattern Facade: un esempio

```
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.close()
```

- Questi due metodi rendono un Archivio un context manager
- Il metodo __enter__() method restituisce self (un'istanza di Archive) che viene assegnata alla variabile dello statement with ...as
- Il metodo __exit__() chiude il file object dell'archivio se c'è ne uno aperto..

Il Design Pattern Facade: un esempio

```
    def names(self):
        if self._file is None:
            self._prepare()
        return self._names()
```

Questo metodo restituisce una lista dei nomi dei file dell'archivio aprendo l'archivio (se non è già aperto) e ponendo in self._names e in self._unpack i callable appropriati utilizzando il metodo self.prepare().

Il Design Pattern Facade: un esempio

```
    def unpack(self):
        if self._file is None:
            self._prepare()
        self._unpack()
```

Questo metodo spacchetta tutti i file di archivio ma solo se tutti i loro nomi sono "safe".

Il Design Pattern Facade: un esempio

```
    def _prepare(self):
        if self.filename.endswith((".tar.gz", ".tar.bz2", ".tar.xz",
                ".zip")):
            self._prepare_tarball_or_zip()
        elif self.filename.endswith(".gz"):
            self._prepare_gzip()
        else:
            raise ValueError("unreadable: {}".format(self.filename))
```

Questo metodo delega la preparazione ai metodi adatti a occuparsene,
Per i tarball e i file zip il codice necessario è molto simile e per questo essi vengono preparati dallo stesso metodo.
I file gzip richiedono una gestione diversa e per questo hanno un metodo a parte.
I metodi di preparazione devono assegnare dei callable alle variabili self._names e self._unpack in modo che queste possano essere chimate nei metodi names() e unpack().

Il Design Pattern Facade: un esempio

- Questo metodo comincia con il creare una funzione innestata safe_extractall() che controlla tutti i nomi dell'archivio e lancia ValueError se qualcuno di essi non è safe.
- Se tutti i nomi sono safe viene invocato o il metodo tarball.TarFile.extractall() oppure il metodo zipfile.ZipFile.extractall().

```
    def _prepare_tarball_or_zip(self):
        def safe_extractall():
            unsafe = []
            for name in self.names():
                if not self.is_safe(name):
                    unsafe.append(name)
            if unsafe:
                raise ValueError("unsafe to unpack: {}".format(unsafe))
            self._file.extractall()
        if self.filename.endswith(".zip"):
            self._file = zipfile.ZipFile(self.filename)
            self._names = self._file.namelist
            self._unpack = safe_extractall
        else: # Ends with .tar.gz, .tar.bz2, or .tar.xz
            suffix = os.path.splitext(self.filename)[1]
            self._file = tarfile.open(self.filename, "r:" + suffix[1:])
            self._names = self._file.getnames
            self._unpack = safe_extractall
```


Il Design Pattern Facade: un esempio

- A seconda dell'estensione del nome dell'archivio, viene aperto un tarball.TarFile o uno zipfile.ZipFile e assegnato a self._file.
- self._names viene settata al metodo bound corrispondente (namelist() o getnames())
- self._unpack viene settata alla funzione safe_extractall() appena creata. Questa funzione è una chiusura che ha catturato self e quindi può accedere a self._file e chiamare il metodo appropriato extractall()

```
    def _prepare_tarball_or_zip(self):
        def safe_extractall():
            unsafe = []
            for name in self.names():
                if not self.is_safe(name):
                    unsafe.append(name)
            if unsafe:
                raise ValueError("unsafe to unpack: {}".format(unsafe))
            self._file.extractall()
        if self.filename.endswith(".zip"):
            self._file = zipfile.ZipFile(self.filename)
            self._names = self._file.namelist
            self._unpack = safe_extractall
        else: # Ends with .tar.gz, .tar.bz2, or .tar.xz
            suffix = os.path.splitext(self.filename)[1]
            self._file = tarfile.open(self.filename, "r:" + suffix[1:])
            self._names = self._file.getnames
            self._unpack = safe_extractall
```

Il Design Pattern Facade: un esempio

```
    def is_safe(self, filename):
        return not (filename.startswith(("/", "\\")) or
            (len(filename) > 1 and filename[1] == ":" and
             filename[0] in string.ascii_letter) or
            re.search(r"[.][.][/\\]", filename))
```

- Un file di archivio creato in modo malizioso potrebbe, una volta spacchettato, sovrascrivere importanti file di sistema rimpiazzandoli con file non funzionanti o pericolosi.
- In considerazione di ciò, non dovrebbero mai essere aperti archivi contenenti file con path assoluti o che includono path relative ed evitare di aprire gli archivi con i privilegi di un utente come root o Administrator.
- is_safe() restituisce False se il nome del file comincia con un forward slash o con un backslash (cioè un path assoluto) o contiene ../ o ..\ (cioè un path relativo che potrebbe condurre ovunque), oppure comincia con D: dove D indica un'unità disco di Windows.

57

Il Design Pattern Facade: un esempio

```
    def _prepare_gzip(self):
        self._file = gzip.open(self.filename)
        filename = self.filename[:-3]
        self._names = lambda: [filename]
        def extractall():
            with open(filename, "wb") as file:
                file.write(self._file.read())
        self._unpack = extractall
```

Questo metodo fornisce un object file aperto per self._file e assegna callable adatti a self._names e self._unpack.
La funzione extractall(), legge e scrive dati.
Il pattern Facade permette di creare interfacce semplici e comode che ci permettono di ignorare i dettagli di basso livello. Uno svantaggio di questo design pattern potrebbe essere quello di non consentire un controllo più fine.
Tutttavia, un facade non nasconte o elimina le funzionalità del sistema sottostante e così è possibile usare un facade passando però a classi di più basso livello se abbiamo bisogno di un maggiore controllo.

58

I context manager

I context manager consentono di allocare e rilasciare risorse quando vogliamo
L'esempio più usato di context manager è lo statement with.

```
with open('some_file', 'w') as opened_file:
        opened_file.write('Hola!')
```

Questo codice apre il file, scrive alcuni dati in esso e lo chiude. Se si verifica un errore mentre si scrivono i dati, esso cerca di chiuderlo.
Il codice in alto è equivalente a

```
file = open('some_file', 'w')
try:
        file.write('Hola!')
finally:
        file.close()
```


I context manager

- è possibile implementare un context manager con una classe.

```
class File:
        def __init__(self, file_name, method):
                self.file_obj = open(file_name, method)
        def __enter__(self):
                return self.file_obj
        def __exit__(self, type, value, traceback):
                self.file_obj.close()
```

I context manager

- è sufficiente definire __enter__() ed __exit__() per poter usare la classe File in uno statement with.

```
with File('demo.txt', 'w') as opened_file:
        opened_file.write('Hola!')
```


61

I context manager

- Come funziona lo statement with:
  - Immagazzina il metodo __exit__() della classe File
  - Invoca il metodo __enter__() della classe File
    - Il metodo __enter__ restituisce il file object per il file aperto.
  - L'object file è passato a opened_file.
  - Dopo che è stato eseguito il blocco al suo interno, lo statement with invoca il metodo __exit__()
    - Il metodo __exit__() chiude il file

```
with File('demo.txt', 'w') as opened_file:
        opened_file.write('Hola!')
```


62

I context manager

- Se tra il momento in cui viene passato l'object file a opened_file e il momento in cui viene invocata __exit__, si verifica un'eccezione allora Python passa type, value e traceback dell'eccezione come argomenti a __exit__() per decidere come chiudere il file e se eseguire altri passi. In questo esempio gli argomenti di exit non influiscono sul suo comportamento.

```
with File('demo.txt', 'w') as opened_file:
        opened_file.write('Hola!')
```


63

I context manager

- Se il file object lanciasse un'eccezione, come nel caso in cui provassimo ad accedere ad un metodo non supportato dal file object:

```
with File('demo.txt', 'w') as opened_file:
        opened_file.undefined_function('Hola!')
```

- with eseguirebbe i seguenti passi:

1. passerebbe type, value e traceback a __exit__()
2. permetterebbe a __exit__() di gestire l'eccezione
3. Se __exit__() restituisse True allora l'eccezione non verrebbe rilanciata dallo statement with.
4. Se __exit__() restituisse un valore diverso da True allora l'eccezione verrebbe lanciata dallo statement with

64

I context manager

```
with File('demo.txt', 'w') as opened_file:
        opened_file.undefined_function('Hola!')
```

Nel nostro esempio, __exit__() restituisce (implicitamente) None per cui with lancerebbe l'eccezione:

```
Traceback (most recent call last):
      File "<stdin>", line 2, in <module>
AttributeError: 'file' object has no attribute 'undefined_function'
```


65

I context manager

Il metodo __exit__() in basso invece gestisce l'eccezione:

```
class File(object):
        def __init__(self, file_name, method):
                self.file_obj = open(file_name, method)
        def __enter__(self):
                return self.file_obj
        def __exit__(self, type, value, traceback):
                print("Exception has been handled")
                self.file_obj.close()
                return True

with File('demo.txt', 'w') as opened_file:
        opened_file.undefined_function()
```


66

I context manager

- è possibile implementare un context manager con un generatore utilizzando il modulo contextlib. Il decoratore Python contextmanager trasforma il generatore open_file in un oggetto GeneratorContextManager

```
from contextlib import contextmanager
@contextmanager
def open_file(name):
        f = open(name, 'w')
        yield f
        f.close()
```

- Si usa in questo modo

```
with open_file('some_file') as f:
        f.write('hola!')
```


67

I context manager

- Nel punto in cui c'è yield il blocco nello statement with viene eseguito.
- Il generatore riprende all'uscita del blocco.
- Se nel blocco si verifica un'eccezione non gestita, essa viene rilanciata nel generatore nel punto dove si trova yield.
- è possibile usare uno statement try...except...finally per catturare l'errore.
- Se un'eccezione è catturata solo al fine di registrarla o per svolgere qualche azione (piuttosto che per sopprimerla), il generatore deve rilanciare l'eccezione. Altrimenti il generatore context manager indicherà allo statement with che l'eccezione è stata gestita e l'esecuzione riprenderà dallo statement che segue lo statement with.

68

I context manager

- Lo statement try...finally garantisce che il file venga chiuso anche nel caso si verifichi un' eccezione nel blocco del with

```
from contextlib import contextmanager
@contextmanager
 def open_file(name):
        f = open(name, 'w')
        try:
                yield f
        finally:
                f.close()
```


Programmazione Avanzata

Design Pattern: Template Method

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 può 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 può 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 è in grado di contare le parole del file. il metodo restituisce implicitamente None per indicare che non è 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 è un sottoinsieme di UTF-8).

re.compile(r"\w+")
restituisce un oggetto espressione regolare per fare il match di parole

regex.finditer(*line*) scandice line da sinistra a destra e restituisce i match (nel nostro caso le parole) 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
```

re.compile

- re.**compile**(*pattern*)
- compila un pattern di un'espessione regolare in un oggetto espressione regolare che puo` essere usato per fare il match usando metodi quali match() e search().
- restituisce uun oggetto di tipo re.Pattern

- esempio:

```
>>> pattern = re.compile("d")
>>> pattern.search("dog") # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1) # No match; search doesn't include the "d"
```


75

Programmazione Avanzata

Design Pattern: Factory Method

76

Factory Method Pattern

- È un design pattern creazionale.
- Si usa quando vogliamo definire un'interfaccia o una classe astratta per creare degli oggetti e delegare le sue sottoclassi a decidere quale classe istanziare quando viene richiesto un oggetto.
  - Particolarmente utile quando una classe non può conoscere in anticipo la classe degli oggetti che deve creare.

Factory Method Pattern: un'applicazione

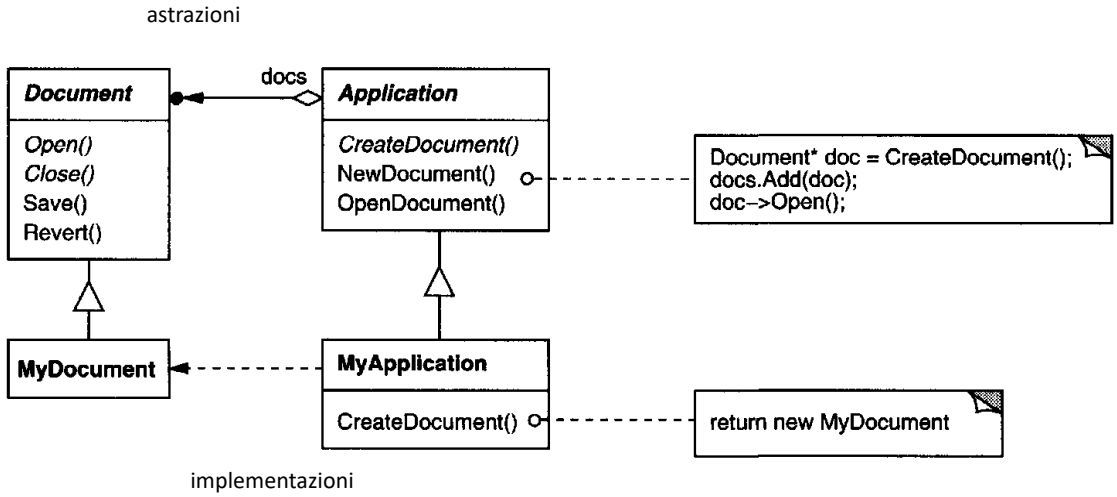
- **Esempio:** Consideriamo un framework per delle applicazioni ciascuna delle quali elabora documenti di diverso tipo.
  - Abbiamo bisogno di due astrazioni: la classe Application e la classe Document
    - La classe Application gestisce i documenti e li crea su richiesta dell'utente, ad esempio, quando l'utente seleziona Open o New dal menu.
  - Entrambe le classi sono astratte e occorre definire delle loro sottoclassi per poter realizzare le implementazioni relative a ciascuna applicazione
    - Ad esempio, per creare un'applicazione per disegnare, definiamo le classi DrawingApplication e DrawingDocument.
  - Definiamo un'interfaccia per creare un oggetto ma lasciamo alle sottoclassi decidere quali classi istanziare.

Factory Method Pattern: un'applicazione

- Poiché la particolare sottoclasse di Document da istanziare dipende dalla particolare applicazione, la classe Application non può fare previsioni riguardo alla sottoclasse di Document da istanziare
- La classe Application sa solo quando deve essere creato un nuovo documento ma non ne conosce il tipo.
- **Problema:** devono essere istanziate delle classi ma si conoscono solo delle classi astratte che non possono essere istanziate
- Il Factory method pattern risolve questo problema incapsulando l'informazione riguardo alla sottoclasse di Document da creare e sposta questa informazione all'esterno del framework.

79

Factory Method Pattern: un'applicazione



80

Factory Method Pattern: un'applicazione

- Le sottoclassi di Application ridefiniscono il metodo astratto CreateDocument per restituire la sottoclasse appropriata di Document
- Una volta istanziata, la sottoclasse di Application può creare istanze di Document per specifiche applicazioni senza dover conoscere le sottoclassi delle istanze create (CreateDocument)
- CreateDocument è detto factory method perché è responsabile della creazione degli oggetti

Factory Method Pattern: un esempio

Voglio creare una scacchiera per la dama ed una per gli scacchi

```
def main():
    checkers = CheckersBoard()
    print(checkers)

    chess = ChessBoard()
    print(chess)
```

Factory Method Pattern: un esempio

- la scacchiera è una lista di liste (righe) di stringhe di un singolo carattere
- __init__ Inizializza la scacchiera con tutte le posizioni vuote e poi invoca populate_board per inserire i pezzi del gioco
- populate_board è astratto

- La funzione console() restituisce una stringa che rappresenta il pezzo ricevuto in input sul colore di sfondo passato come secondo argomento.

```
BLACK, WHITE = ("BLACK", "WHITE")

class AbstractBoard:

    def __init__(self, rows, columns):
        self.board = [[None for _ in range(columns)] for _ in range(rows)]
        self.populate_board()

    def populate_board(self):
        raise NotImplementedError()

    def __str__(self):
        squares = []
        for y, row in enumerate(self.board):
            for x, piece in enumerate(row):
                square = console(piece, BLACK if (y + x) % 2 else WHITE)
                squares.append(square)
            squares.append("\n")
        return "".join(squares)
```

83

Factory Method Pattern: un esempio

- La classe per creare scacchiere per il gioco della dama

```
class CheckersBoard(AbstractBoard):

    def __init__(self):
        super().__init__(10, 10)

    def populate_board(self):
        for x in range(0, 9, 2):
            for row in range(4):
                column = x + ((row + 1) % 2)
                self.board[row][column] = BlackDraught()
                self.board[row + 6][column] = WhiteDraught()
```


84

Factory Method Pattern: un esempio

- La classe per scacchiere per il gioco degli scacchi

```
class ChessBoard(AbstractBoard):

    def __init__(self):
        super().__init__(8, 8)

    def populate_board(self):
        self.board[0][0] = BlackChessRook()
        self.board[0][1] = BlackChessKnight()
        ...
        self.board[7][7] = WhiteChessRook()
        for column in range(8):
            self.board[1][column] = BlackChessPawn()
            self.board[6][column] = WhiteChessPawn()
```

I metodi populate_board() di CheckersBoard e ChessBoard non sono dei factory method dal momento che le classi usate per creare i pezzi sono fissate nel codice.

Factory Method Pattern: un esempio

- La classe base per i pezzi
- Si è scelto di creare una classe che discende da str invece che usare direttamente str per poter facilmente testare se un oggetto z è un pezzo del gioco con isinstance(z,Piece)
- ponendo __slots__=() ci assicuriamo che gli oggetti di tipo Piece non abbiano variabili di istanza

```
class Piece(str):
    __slots__ = ()
```

Factory Method Pattern: un esempio

- La classe pedina nera e la classe re bianco
- le classi per gli altri pezzi sono create in modo analogo
  - Ognuna di queste classi è una sottoclasse immutabile di Piece che è sottoclasse di str
  - Inizializzata con la stringa di un unico carattere (il carattere Unicode che rappresenta il pezzo)

```
class BlackDraught(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```


87

Factory Method Pattern: un esempio

- Notiamo che qui la stringa che indica il pezzo è assegnata da __new__
- Il metodo __new__ non prende argomenti in quanto la stringa che rappresenta il pezzo è codificato all'interno del metodo.
  - TypeError: __new__() takes 1 positional argument but 2 were given
- Per i tipi che estendono tipi immutable, come str, l'inizializzazione è fatta da __new__.
  - https://docs.python.org/3/reference/datamodel.html : __new__() is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation.

```
class BlackDraught(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```


88

Factory Method Pattern: un esempio

- Questa nuova versione del metodo CheckersBoard.populate_board() è un **factory method** in quanto dipende dalla factory function create_piece()
- Nella versione precedente il tipo di pezzo era indicato nel codice
- La funzione create_piece() restituisce un oggetto del tipo appropriato (ad esempio, BlackDraught o WhiteDraught) in base ai suoi argomenti.
- Il metodo ChessBoard.populate_board() viene anch'esso modificato in modo da usare la stessa funzione create_piece() invocata qui.

```
def populate_board(self):
    for x in range(0, 9, 2):
        for y in range(4):
            column = x + ((y + 1) % 2)
            for row, color in ((y, "black"), (y + 6, "white")):
                self.board[row][column] = create_piece("draught",
                        color)
```

89

Factory Method Pattern: un esempio

- La funzione factory create_piece usa la funzione built-in eval() per creare istanze della classe
- Ad esempio se gli argomenti sono "knight" and "black", la stringa valutata sarà "BlackChessKnight()".
- In generale è meglio non usare eval per eseguire il codice rappresentato da un'espressione perché è potenzialmente rischioso dal momento che permette di eseguire il codice rappresentato da una qualsiasi espressione

```
def create_piece(kind, color):
    if kind == "draught":
        return eval("{}{}()".format(color.title(), kind.title()))
    return eval("{}Chess{}()".format(color.title(), kind.title()))
```


90

Factory Method Pattern: un esempio

- Questa versione di CheckersBoard.populate_board() differisce da quella precedente in quanto il pezzo e il colore sono specificati da costanti e usa un nuovo factory create_piece().

```
DRAUGHT, PAWN, ROOK, KNIGHT, BISHOP, KING, QUEEN = ("DRAUGHT", "PAWN",
       "ROOK", "KNIGHT", "BISHOP", "KING", "QUEEN")

class CheckersBoard(AbstractBoard):
    ...

    def populate_board(self):
        for x in range(0, 9, 2):
            for y in range(4):
                column = x + ((y + 1) % 2)
                for row, color in ((y, BLACK), (y + 6, WHITE)):
                    self.board[row][column] = self.create_piece(DRAUGHT,
                            color)
```


91

Factory Method Pattern: un esempio

- Questa versione di create_piece() è un metodo di AbstractBoard che viene ereditato da CheckersBoard e ChessBoard.
- Prende in input due costanti kind e color e cerca nel dizionario __classForPiece la classe associata alla chiave (kind,color)
- La classe cosi` individuata viene quindi utilizzata per instanziare il pezzo desiderato.

```
class AbstractBoard:

    __classForPiece = {(DRAUGHT, BLACK): BlackDraught,
            (PAWN, BLACK): BlackChessPawn,
            ...
            (QUEEN, WHITE): WhiteChessQueen}
    ...
    def create_piece(self, kind, color):
        return AbstractBoard.__classForPiece[kind, color]()
```


92

Factory Method Pattern: un esempio

- L'ultima versione di create_piece comunque fa uso di informazioni riguardanti le sottoclassi di Piece che si trovano all'interno della classe AbstractBoard e in particolare nel dizionario __classForPiece di quella classe
- La versione di create_piece riportata in basso non usa il dizionario AbstractBoard. __classForPiece ma ricerca direttamente la sottoclasse da utilizzare nel dizionario restituito da global()

```
def create_piece(kind, color):
    color = "White" if color == WHITE else "Black"
    name = {DRAUGHT: "Draught", PAWN: "ChessPawn", ROOK: "ChessRook",
            KNIGHT: "ChessKnight", BISHOP: "ChessBishop",
            KING: "ChessKing", QUEEN: "ChessQueen"}[kind]
    return globals()[color + name]()
```


Factory Method Pattern: un esempio

- Un modo di rendere piu` dinamiche le implementazioni fino ad ora viste è di aggiungere le sottoclassi dinamicamente:
  - Invece di codificare tutte le sottoclassi di Piece una ad una (staticamente), possiamo crearle dinamicamente con il seguente frammento di codice:

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```

Factory Method Pattern: un esempio

- Questo frammento di codice scandisce tutti i numeri esadecimali associati alle pedine e agli scacchi e per ciascuno di essi
  - Salva in **char** la stringa che rappresenta il carattere Unicode ad esso associato.
  - Salva in **name** la stringa assegnata a **char** dopo aver trasformato in maiuscole le iniziali di tutte le parole al suo interno ed eliminato gli spazi
    - Ad esempio per code= 0x2654, setta char='♔' e name='WhiteChessKing'
  - Se name finisce con "sMan" (cioe` se è un pezzo della dama) cancella il suffisso 'sMan' da name
  - Invoca make_new_method(char) per creare una nuova funzione che viene memorizzata in **new**

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```

la descrizione del corpo del for continua nella slide successiva

95

Factory Method Pattern: un esempio

- Memorizza in Class una nuova classe. Questa nuova classe è ottenuta invocando la funzione built-in **type** con i seguenti argomenti:
  - name : nome della classe
  - (Piece,): tupla delle classi base della classe
  - dizionario **dict**(__slots__=(), __new__=new)) : dizionario degli attributi della classe
    - in questo modo le istanze della classe non avranno __dict__ e sarrano create usando il metodo new
- aggiunge la classe Class al modulo corrente con setattr (l'attributo corrispondente avra` lo stesso nome della classe (name))

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```


96

Factory Method Pattern: un esempio

```
def make_new_method(char): # Needed to create a fresh method each time
    def new(Class): # Can't use super() or super(Piece, Class)
        return Piece.__new__(Class, char)
    return new
```


97

Factory Method Pattern: un esempio

class **type**(*name*, *bases*, *dict*, ***kwds*)

- Cosa dice la documentazione:
- type invocato con tre argomenti restituisce un nuovo oggetto tipo (una nuova classe). Questa è essenzialmente una forma dinamica dello statement class.
- La stringa *name* string è il nome della classe e diventa il valore dell'attributo __name__
- La tupla *bases* contiene le classi base e diventa l'attributo __bases__; se è vuota viene aggiunta object come classe base
- Il dizionario *dict* dictionary contiene le definizioni degli attributi e dei metodi della classe.
- Questi due frammenti di codice creano la stessa classe:
  - **class X**:
    a = 1
  - X = type('X', (), dict(a=1))

98

Factory Method Pattern: un esempio

```
>>> for x in range(0x2654, 0x2660):
...     chr(x)
...
'♔'
'♕'
'♖'
'♗'
'♘'
'♙'
'♚'
'♛'
'♜'
'♝'
'♞'
'♟'
```

```
>>> for x in range(0x2654, 0x2660):
...     unicodedata.name(chr(x))
...
'WHITE CHESS KING'
'WHITE CHESS QUEEN'
'WHITE CHESS ROOK'
'WHITE CHESS BISHOP'
'WHITE CHESS KNIGHT'
'WHITE CHESS PAWN'
'BLACK CHESS KING'
'BLACK CHESS QUEEN'
'BLACK CHESS ROOK'
'BLACK CHESS BISHOP'
'BLACK CHESS KNIGHT'
'BLACK CHESS PAWN'
```