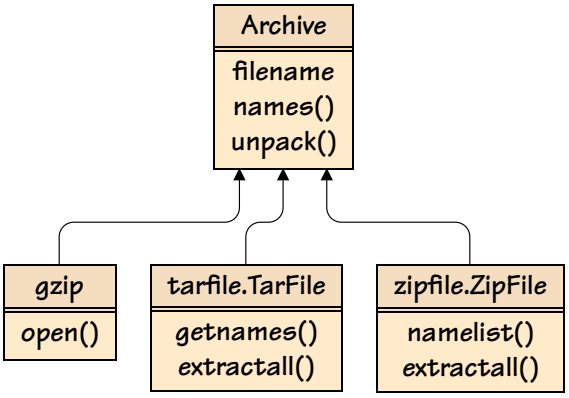
Programmazione Avanzata

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!')
```


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

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: Factory Method

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.

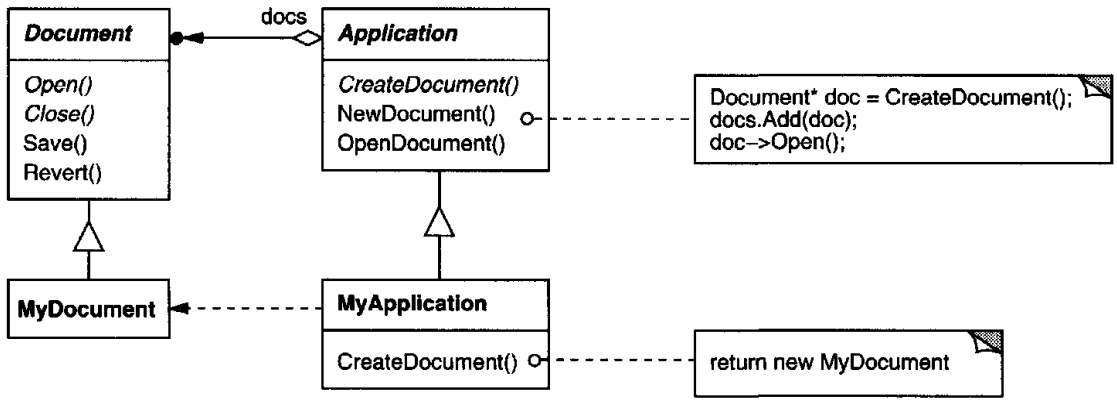
Factory Method Pattern: un'applicazione

- **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.

Factory Method Pattern: un'applicazione



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 semplice esempio

```
class Pizza():
        def __init__(self):
            self._price = None

        def get_price(self):
            return self._price
```

Factory Method Pattern: un semplice esempio

```
class HamAndMushroomPizza(Pizza):
        def __init__(self):
            self._price = 8.5

class DeluxePizza(Pizza):
        def __init__(self):
            self._price = 10.5

class HawaiianPizza(Pizza):
        def __init__(self):
            self._price = 11.5
```


Factory Method Pattern: un semplice esempio

- PizzaFactory fornisce il metodo createPizza che è statico per cui può essere invocato quando non è stata ancora creata una pizza

```
class PizzaFactory:
    @staticmethod
    def create_pizza(pizza_type):
        if pizza_type == 'HamMushroom':
            return HamAndMushroomPizza()
        elif pizza_type == 'Deluxé:
            return DeluxePizza()
        elif pizza_type == 'Hawaiian':
            return HawaiianPizza()
```

Factory Method Pattern: un semplice esempio

```
if __name__ == '__main__':
    for pizza_type in ('HamMushroom', 'Deluxé, 'Hawaiian'):
       print('Price of {0} is {1}'.format(pizza_type,
       PizzaFactory.create_pizza(pizza_type).get_price())
```

- il tipo di pizza avrebbe potuto essere fornito dall'utente
- il tipo di pizza indicato dall'utente potrebbe essere stato inserito successivamente nel menu e la classe concreta corrispondente creata successivamente al main
- occorre modificare solo la factory

Factory Method Pattern: un semplice esempio

- Che cosa accade se vogliamo creare diversi tipi di negozi ciascuno dei quali vende pizze nello stile di una certà città
- Creiamo una classe astratta PizzaStore al cui interno c'è il metodo astratto **create_pizza**
- Dalla classe PizzaStore deriviamo NYPizzaStore, ChicagoPizzaStore e così via. Queste sottoclassi sovrascriveranno il metodo astratto. La decisione sul tipo di pizza da creare è presa dal metodo create_pizza della specifica sottoclasse.
  - analogamente a quanto accadeva nel framework per la gestione dei documenti
- PizzaStore avrà anche un metodo orderPizza() che invoca createPizza ma non ha idea su quale pizza verrà creata fino a che non verra` creata una classe concreta di PizzaStore

Factory Method Pattern: un semplice esempio

```
from abc import ABC, abstractmethod

class Pizza(ABC):

	@abstractmethod
	def prepare(self):
		pass

	def bake(self):
		print("baking pizza for 12min in 400 degrees..")

	def cut(self):
		print("cutting pizza in pieces")

	def box(self):
		print("putting pizza in box")
```


81

Factory Method Pattern: un semplice esempio

```
class NYStyleCheesePizza(Pizza):
	def prepare(self):
		print("preparing a New York style cheese pizza..")

class ChicagoStyleCheesePizza(Pizza):
	def prepare(self):
		print("preparing a Chicago style cheese pizza..")

class NYStyleGreekPizza(Pizza):
	def prepare(self):
		print("preparing a New York style greek pizza..")

class ChicagoStyleGreekPizza(Pizza):
	def prepare(self):
		print("preparing a Chicago style greek pizza..")
```


82

Factory Method Pattern: un semplice esempio

```
class PizzaStore(ABC):

      @abstractmethod
      def _createPizza(self, pizzaType: str) -> Pizza:
            pass

      def orderPizza(self, pizzaType):


            pizza = self._createPizza(pizzaType)

            pizza.prepare()
            pizza.bake()
            pizza.cut()
            pizza.box()
```


83

Factory Method Pattern: un semplice esempio

```
class NYPizzaStore(PizzaStore):

      def _createPizza(self, pizzaType: str) -> Pizza:

            pizza = None

            if pizzaType == 'Greek':
                  pizza = NYStyleGreekPizza()
            elif pizzaType == 'Cheese':
                  pizza = NYStyleCheesePizza()
            else:
                  print("No matching pizza found in the NY pizza store...")
            return pizza
```


84

Factory Method Pattern: un semplice esempio

```
class ChicagoPizzaStore(PizzaStore):

      def _createPizza(self, pizzaType: str) -> Pizza:

            pizza = None

            if pizzaType == 'Greek':
                  pizza = ChicagoStyleGreekPizza()
            elif pizzaType == 'Cheese':
                  pizza = ChicagoStyleCheesePizza()
            else:
                  print("No matching pizza found in the Chicago pizza store...")
            return pizza
```


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)
```

87

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()
```


88

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()
```


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}")
```


91

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}")
```


92

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)
```

93

Factory Method Pattern: un esempio

- Questa funzione factory 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()))
```


94