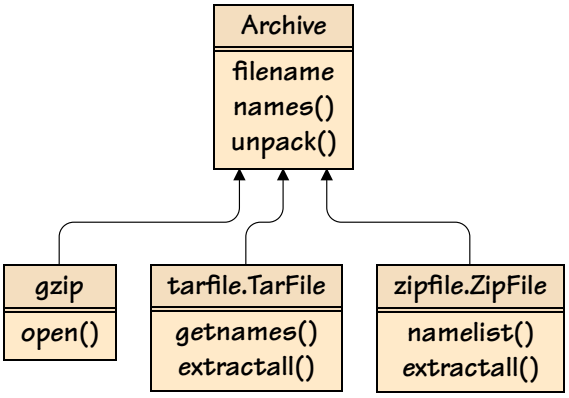
Programmazione Avanzata

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 di archive.
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 **tarfile.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 nasconde 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

- 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()
```


68