

Classi in Python

Qualche dettaglio in più

Nelle slide precedenti...

Abbiamo visto qualche accenno di base alle classi in python.

E' arrivato il momento di fare un confronto diretto con Java per ciò che concerne i diversi approcci filosofici "OOP" tra questi due linguaggi.

Per ora, abbiamo solo capito che Python integra ereditarietà multipla, cosa che Java **non** può fare.

Altre caratteristiche chiave

- Incapsulamento degli attributi
- Classi\Interfacce astratte
- Metodi\Attributi statici
- Custom comparators

Incapsulamento

In Java, si tende a rendere private\protette tutti le variabili di classe.

Questo perché i setter ed i getter ci permettono di aggiungere regole di accesso alle variabili personalizzate. E questa è una necessità che si presenta spesso.

Esempio:

Si implementi una classe Cerchio. La classe Cerchio ha due costruttori, uno di default (che inizializza il raggio, suo unico attributo ad 1) ed uno che accetta un raggio di dimensione fissata dal programmatore. Il valore di raggio dev'essere strettamente maggiore di 0.

Proteggere gli attributi

“There’s no privacy in Python...” R. Hettinger, Python Guru.

```
class Cerchio:
    def __init__(self, raggio):
        self.__raggio = raggio #una delle tre possibilità...
        self._raggio = raggio
        self.raggio = raggio
```

self.raggio	Attributo pubblico. Fanne quello che vuoi
self._raggio	Per favore, non accedere direttamente a questo attributo
self.__raggio	Se provi ad accedere a questo attributo io non ti parlo più

Sperimentiamo

```
class Classe:
    def __init__(self,a,b,c):
        self.a = a
        self._b = b
        self.__c = c

ogg = Classe(1,2,3)

print(ogg.a)
print(ogg._b)

try:
    print(ogg.__c)
except:
    print("TI HO DETTO DI NON ACCEDERVI!")
finally:
    print(ogg._Classe__c)
```

1
2
TI HO DETTO DI NON ACCEDERVI!
3

Conclusione

In python non esiste privacy degli attributi.

getter e *setter* sono considerati brutti, secondo “import this”

La privacy viene sostituita da “suggerimenti” riguardo l’accesso delle variabili

Questo fa storcere il naso a chi viene da altri linguaggi, perché la protezione delle variabili di classe non può essere risolta con suggerimenti o velate minacce.

Python risolve comunque questo problema con un meccanismo molto elegante che “unisce” le due filosofie: **controllo degli accessi delle variabili senza dover imbruttire il codice con getter e setter.**

Proprietà in python

L'annotazione **@property** crea un alias di "___raggio" con "raggio", ma qualsiasi scrittura\lettura di "raggio" è subordinato al codice definito tra le altre annotazioni

@nomeattributo.[getter|setter|deleter]

Chi usa questa classe non dovrà mai chiamare direttamente questi metodi, in quanto verranno eseguiti ogni qualvolta si accede "normalmente" all'attributo raggio di Cerchio.

Si veda la slide successiva.

```
class Cerchio:

    def __init__(self,raggio = 1):
        self.raggio = raggio

    def __str__(self):
        return "Questo cerchio ha raggio " +
str(self.raggio)

    @property
    def raggio(self):
        return self.__raggio

    @raggio.setter
    def raggio(self,value):
        if value <= 0:
            self.__raggio = 1
            print("Errore!")
        else: self.__raggio = value

    @raggio.getter
    def raggio(self):
        return self.__raggio
```


Output

```
c = Cerchio(-3)
print(c)
c.raggio = -2
print(c)
c = Cerchio(3)
print(c)
```

Errore!

Questo cerchio ha raggio 1

Errore!

Questo cerchio ha raggio 1

Questo cerchio ha raggio 3

Classi\Metodi astratti ed interfacce

In Java un'interfaccia è una collezione di metodi astratti, pertanto non implementati. In realtà a partire da Java 8, le interfacce Java possono mostrare un'implementazione di default nei loro metodi.

In Java inoltre, una classe si dice astratta se possiede almeno un metodo astratto. Istanziare un oggetto di una classe astratta è un errore.

In **python**, si taglia la testa al toro. Tradizionalmente **non esisteva il concetto di interfaccia, così come non esisteva il concetto di classe o metodo astratto**. Esistevano solo metodi\funzioni “non implementate”. La presenza di metodi non implementati **non impedisce all'utente di istanziare comunque** un oggetto.

Esempio

```
class ClasseAstratta:

    def __init__(self,a):
        self.a = a

    def metodo_astratto(self):
        raise NotImplementedError("Non sono implementato!")
        #pass ##fallimento "silenzioso"

ogg = ClasseAstratta(3)
print(ogg.a)
ogg.metodo_astratto() #questa istruzione fallisce
```

Python più moderno

Esiste “abc” un modulo python che permette la creazione di “vere” classi astratte.

<https://docs.python.org/3/library/abc.html>

Usare “abc” è una soluzione migliore rispetto alle istruzioni “pass\NotImplemented”, ma decisamente più complessa da usare.

Il metodo visto nella slide precedente permette la creazione di un oggetto, ma il programma andrà in crash solo quando un metodo non implementato viene chiamato.

“abc” invece causa un crash se si cerca di implementare le classi annotate come “abstract”, ergo si dice “fallisce prima\più velocemente”

Attributi statici

In generale, se non riguardano “self”, allora sono statici.

class Classe:

```
    attributo_statico = 10
```

```
    def __init__(self,a)
```

```
        self.attributo_dinamico = a
```

```
o = Classe(3)
```

```
print(o.attributo_dinamico)
```

```
print(Classe.attributo_statico)
```

Metodi statici

I metodi statici in Python si dividono in due categorie:

- **Metodi statici puri**
 - Questi somigliano molto ai metodi statici di altri linguaggi, come Java e C++
 - Usano il decoratore\annotazione `@staticmethod`
- **Metodi statici di classe**
 - Si usano per avere un modo “pulito” per creare factory methods
 - Usano il decoratore\annotazione `@classmethod`

```
from datetime import date

class Persona:
    def __init__(self, nome, eta):
        self.nome = nome
        self.eta = eta
        print(f"La persona inserita è adulta?
              {Persona.maggiorenne(eta)}")

    # un class method che crea una persona
    # a partire dal suo anno di nascita
    @classmethod
    def dalla_nascita(cls, name, year):
        return cls(name, date.today().year - year)

    # Un metodo statico che ci dice se
    # una generica persona è maggiorenne
    @staticmethod
    def maggiorenne(eta):
        return eta > 18

person1 = Persona('Nome', 21)
person2 = Persona.dalla_nascita('Nome', 2010)
```

```
from datetime import date
```

```
class Persona:
```

```
    def __init__(self, nome, eta):
```

```
        self.nome = nome
```

```
        self.eta = eta
```

```
        print(f"La persona inserita è adulta?  
              {Persona.maggiorenne(eta)}")
```

```
# un class method che crea una persona
```

```
# a partire dal suo anno di nascita
```

```
@classmethod
```

```
def dalla_nascita(cls, name, year):
```

```
    return cls(name, date.today().year - year)
```

```
# Un metodo statico che ci dice se
```

```
# una generica persona è maggiorenne
```

```
@staticmethod
```

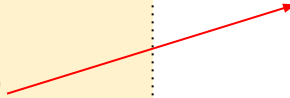
```
def maggiorenne(eta):
```

```
    return eta > 18
```

```
person1 = Persona('Nome', 21)
```

```
person2 = Persona.dalla_nascita('Nome', 2010)
```

Sono obbligato a mettere
Persona.maggiorenne...?



Ereditarietà Multipla

Quando abbiamo trattato le classi in Python abbiamo detto che questo linguaggio supporta l'ereditarietà multipla. Ovvero una classe può estendere più classi padre.

Con tutti i problemi che ne conseguono. 

V'invito a leggere le coding style guide di Google riguardo C++ e Multiple-Inheritance

<https://google.github.io/styleguide/cppguide.html#Inheritance>

Esempio

Ho una classe A

Ho una classe B

Ho una terza classe C che eredita sia da A che da B.

C non ha un costruttore. Chiamerà quello di default.... di A, di B o di entrambi?

in C, chiamare `super()` su un metodo che effetto ha?

Facciamo qualche esperimento

```
class A:
```

```
    def __init__(self,a):
```

```
        print("A inizializzato")
```

```
        self.a = a
```

```
    def metodo(self):
```

```
        print(f"Sono A e Stampo {self.a}")
```

```
    def metodo_specifico_a(self):
```

```
        print("Sono un metodo specifico di  
A")
```

```
class B:
```

```
    def __init__(self,b):
```

```
        print("B inizializzato")
```

```
        self.b = b
```

```
    def metodo(self):
```

```
        print(f"Sono B e Stampo {self.b}")
```

```
class ClasseBA(B,A):
```

```
    def metodo(self):
```

```
        super().metodo()
```

```
class ClasseAB(A,B):
```

```
    def metodo(self):
```

```
        super().metodo()
```

```
oggetto = ClasseBA(3,4) #Errore!
```

```
oggetto1 = ClasseBA(3) #inizializza solo B
```

```
oggetto1.metodo() #Andrà a chiamare metodo di B
```

```
oggetto2 = ClasseAB(4) #inizializza solo A
```

```
oggetto2.metodo() #andrà a chiamare metodo di A.
```

```
print(f"Posso comunque raggiungere {oggetto1.a}?")  
#no...
```

```
oggetto1.metodo_specifico_a() #posso farlo? Si.  
Perchè non usa attributi non inizializzati
```

Spiegazione

L'ordine dei due “padri” è fondamentale per cercare di prevedere il comportamento della classe derivata!

Questo in python è noto come **MRO** (Method Resolution Order)

Non è semplice e si presta a svariate ambiguità.

Sebbene questo esempio sia triviale, nel caso in cui la “profondità” della catena ereditaria crescesse fare previsioni risulterebbe molto difficile.

Morale della favola: ecco perchè non tutti sono entusiasti di usare l'ereditarietà multipla. Se proprio va usata, si devono fare sforzi aggiuntivi per evitare queste situazioni ambigue.

Rimuovendo le ambiguità

```
class ClasseABMigliore(A,B):
```

```
    def __init__(self, a=0, b=0):
```

```
        A.__init__(self,a)
```

```
        B.__init__(self,b)
```

```
    def metodo(self):
```

```
        A.metodo(self)
```

```
        B.metodo(self)
```

```
    def metodo_specifico(self):
```

```
        print(f"Metodo specifico della classe {type(self)} : ", end="")
```

```
        super() .metodo_specifico_a()
```

Output

```
oggetto3 = ClasseABMigliore(3,4)
```

```
oggetto3.metodo()
```

```
oggetto3.metodo_specifico()
```

```
oggetto3.metodo_specifico_a()
```

A inizializzato

B inizializzato

Sono A e Stampo 3

Sono B e Stampo 4

Metodo specifico della classe <class '__main__.ClasseABMigliore'> : Sono un metodo specifico di A

Sono un metodo specifico di A

Ma quindi... perché ne abbiamo parlato?

L'ereditarietà multipla viene **spesso** utilizzata in django.

In particolare, esistono classi Views e Classi chiamate “Access Mixins” che come suggerisce il termine “access” si accertano di aggiungere regole di accesso per le richieste gestite dalla View in questione:

```
class MiaView(GroupRequiredMixin, CreateView): ....
```