Perché usare i metodi getter e setter?

Nel video precedente abbiamo potuto constatare che gli attributi di una classe Python sono pubblici; in Python il concetto di attributo privato non esiste; ci sono delle convenzioni generalmente accettate che hanno il seguente significato:

>>> Il singolo underscore usato come prefisso dell'attributo è una convenzione per indicare che l'attributo o metodo è per uso interno alla classe o alla sottoclasse (attributo protetto)

>>> Il prefisso doppio underscore invece significa che l'attributo è privato e Python lo tratta in maniera particolare perché applica il cosiddetto mangling (Il nome originale __attributo diventa _NomeClasse__attributo) e serve a nascondere gli attributi "privati" per evitare conflitti nelle sottoclassi

Queste convenzioni sono dei Warning: essi esortano caldamente l'utilizzatore della classe a non usare l'attributo perchè il progettista della classe vuol sentirsi libero di poterlo cambiare / eliminare in una futura versione.

La pubblicità di un attributo può essere un vantaggio in quanto ci permette di accedere al suo valore in maniera estremamente semplice ma a volte può essere necessario avere dei metodi che permettono di proteggere e o convalidare i valori memorizzati in tali attributi.

E' in questo contesto che vengono utilizzati i metodi getter e setter:
generalmente si utilizza un metodo setter quando oltre a memorizzare un mero dato nell'attributo di una istanza vogliamo che questo dato si comporti in un certo modo o rispetti determinate regole; cioè quando ad un dato voglio "attaccare" un comportamento (che si traduce in un metodo che esegue controlli ed elaborazioni varie)

Come si procede?

```
1 class Fornitore:
2     def __init__(self, c, n):
3         print('sono in __init_ di istanza fornitori')
4         self._codice=c
5         self._nome=n
```

argomento1

argomento2

Queste sono le variabili che conterranno i valori. Il prefisso _ ci mette in guardia e ci suggerisce di non usare questi nomi, perchè il progettista potrebbe cambiarli!

Un programmatore che volesse creare una istanza di questa classe dovrebbe scrivere la seguente linea di codice:

```
f1=Fornitore('001','alfa srl')
```

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
```

Questo risultato ci fa capire che l'esecuzione è passata per il metodo __init__ (linea 3 print...)

Ridefiniamo il metodo __str__ con cui mostriamo i dati memorizzati in una istanza:

```
def __str__(self):
    return f"\ncodice: {self._codice}\nnome: {self._nome}"
```

Eseguiamo nuovamente il codice per creare l'istanza f1 e faccio un print di f1, ottenendo:

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
>>> print(f1)

codice: 001
nome: alfa srl
```

Il programmatore che utilizza la classe Fornitore per accedere al codice e al nome potrebbe scrivere e ottenere:

```
>>> print(f1._codice, f1._nome)
001 alfa srl
```

Funziona? Sì ma avevamo detto che l'uso di _codice e _nome da parte del programmatore era fortemente sconsigliato!

Come fare?

Si usa una property che funge da getter

potrebbe essere un nome qualsiasi, ma ovviamente diamo un nome chemi faccia capire a cosa ci stiamo riferendo

Questo è il nome che il progettista della classe si impegna a mantenere per garantire la compatibilità nelle versioni future della classe; il programmatore che usa questo nome ha la garanzia che l'interfaccia della classe non cambierà.

```
10      @property
11      def codice(self):
12          print("passo dal getter del codice fattura")
13          return self._codice
```

Ora il programmatore può accedere al dato in due modi diversi:

```
>>> print(f1._codice)
001
>>> print(f1.codice)
passo dal getter del codice fattura
001
```

sconsigliato! (si ottiene il dato grezzo)

istruzione corretta per l'accesso

(il dato potrebbe essere soggetto ad elaborazione nel metodo; p.e. formattazione)

Proviamo ora a modificare il valore del codice:

```
>>> f1.codice='099'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    f1.codice='099'
AttributeError: property 'codice' of 'Fornitore' object has no setter
```

NON FUNZIONA!!!

Abbiamo bisogno del metodo setter

Scriviamo il metodo setter

```
10      @property
11      def codice(self):
12          print("passo dal getter del codice fattura")
13          return self._codice
14
15      @codice.setter
16      def qwerty(self, valore):
17          print("passo dal setter del codice della fattura")
18          self._codice = valore
```

Questo decoratore sta comunicando all'interprete Python che il metodo che segue (qwerty) è un setter per la property codice

nome che il progettista garantisce di non modificare nelle versioni future per mantenere inalterata l'interfaccia della classe

Nome del metodo volutamente non significativo. Funziona

lo scenario è il seguente

per impostare un valore il programmatore scrive:

f1.querty='099'

come se qwerty fosse un attributo

in realtà qwerty(self, valore) è un metodo che può fare alcune elaborazioni prima di assegnare effettivamente il valore

099

_codice

in realtà codice(self) è un metodo che può fare alcune elaborazioni prima di restituire il risultato

per leggere il valore il programmatore scrive:

f1.codice

come se codice fosse un attributo

f1: istanza della classe Fornitore

Vi invito a scrivere nel terminale le istruzioni evidenziate per constatare che effettivamente funziona (dopo aver fatto f5 sul file che definisce la classe)

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
>>> f1
<__main__.Fornitore object at 0x7f1ce6986f90>
>>> print(f1)

codice: 001
nome: alfa srl
>>> f1.__dict__
{'_codice': '001', '_nome': 'alfa srl'}
>>> f1._codice
'001'
>>> f1.codice
passo dal getter del codice fattura
'001'
>>> f1.qwerty='099'
passo dal setter del codice della fattura
>>> f1.codice
passo dal getter del codice fattura
'099'
>>> f1.__dict__
{'_codice': '099', '_nome': 'alfa srl'}
>>> print(f1)

codice: 099
nome: alfa srl
```

Ritorniamo al nostro codice e cambiamo il nome al metodo setter per renderlo congruente con il contesto:

```
1  class Fornitore:
2      def __init__(self, c, n):
3          print('sono in __init__ di istanza fornitori')
4          self._codice=c
5          self._nome=n
6
7      def __str__(self):
8          return f"\ncodice: {self._codice}\nnome: {self._nome}"
9
10     @property
11     def codice(self):
12         print("passo dal getter del codice fattura")
13         return self._codice
14
15     @codice.setter
16     def codice(self, valore):
17         print("passo dal setter del codice della fattura")
18         self._codice = valore
19
```

quindi da ora in poi il programmatore scriverà:

per leggere `f1.codice`

per assegnare un valore `f1.codice='099'`

Bello e pulito. vero?

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
>>> print(f1)

codice: 001
nome: alfa srl
>>> print(f1.__dict__)
{'_codice': '001', '_nome': 'alfa srl'}
>>> f1.codice='099'        # accedo in scrittura
passo dal setter del codice della fattura
>>> print(f1.__dict__)
{'_codice': '099', '_nome': 'alfa srl'}
>>> print(f1.codice)       # accedo per la lettura
passo dal getter del codice fattura
099
```

Qualcuno potrebbe chiedermi:
ma perchè abbiamo praticamente raddoppiato le linee di codice da scrivere per operare poi allo stesso modo?

La differenza la fanno queste 2 linee

Con questa architettura noi costringiamo l'esecutore a passare attraverso i due metodi getter e setter all'interno dei quali possiamo effettuare elaborazioni in maniera centralizzata

Osservazione

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
>>> print(f1)

codice: 001
nome: alfa srl
>>> print(f1.__dict__)
{'_codice': '001', '_nome': 'alfa srl'}
>>> f1.codice='099'
passo dal setter del codice della fattura
>>> print(f1.__dict__)
{'_codice': '099', '_nome': 'alfa srl'}
>>> print(f1.codice)
passo dal getter del codice fattura
099
```

Questo messaggio non appare in fase di __init__

Perchè in fase di __init__ non appare la scritta di passaggio dal setter?

La risposta è semplice: perchè non passa dal setter!!!

```
1 class Fornitore:
2     def __init__(self, c, n):
3         print('sono in __init__ di istanza fornitori')
4         self.codice=c
5         self.nome=n
```

Se guardiamo bene il codice, nella __init__ si sta assegnando il valore direttamente all'attributo _codice, non alla property codice;

ne consegue che il meccanismo che abbiamo architettato non scatta. Esso scatta solo se assegnamo un valore alla property.

Quindi dobbiamo apportare una piccola modifica alle linee 4 e 5 del codice: togliere l'underscore

Versione (quasi) definitiva della definizione della classe Fornitore

```
1  class Fornitore:
2      def __init__(self, c, n):
3          print('sono in __init__ di istanza fornitori')
4          self.codice=c
5          self.nome=n
6
7      def __str__(self):
8          return f"\ncodice: {self._codice}\nnome: {self._nome}"
9
10
11     @property
12     def codice(self):
13         print("passo dal getter del codice fattura")
14         return self._codice
15
16
17     # consente di assegnare un valore all'attributo codice
18     @codice.setter
19     def codice(self, valore):
20         print("passo dal setter del codice della fattura")
21         self._codice = valore
22
23     #f1=Fornitore('001','alfa srl')
24
```

Se mandiamo in esecuzione otteniamo

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
passo dal setter del codice della fattura
>>> print(f1)

codice: 001
nome: alfa srl
>>> f1.__dict__
{'_codice': '001', '_nome': 'alfa srl'}
>>> f1.codice
passo dal getter del codice fattura
'001'
>>> f1.codice='099'
passo dal setter del codice della fattura
>>> print(f1)

codice: 099
nome: alfa srl
>>> f1.codice
passo dal getter del codice fattura
'099'
>>> f1.__dict__
{'_codice': '099', '_nome': 'alfa srl'}
```

Ora passa dal setter anche in fase di __init__

Vedremo adesso come è possibile sfruttare il metodo setter per fare alcune validazioni del dato che il programmatore vorrebbe memorizzare.
Come al solito partiamo con un esempio.

Supponiamo che per iL codice del Fornitore debbano valere le seguenti regole:

- Il codice deve essere di tipo stringa
- Il codice deve essere lungo esattamente 3 caratteri

Il metodo setter diventa:

```
17      # consente di assegnare un valore alla property codice
18      def codice(self, valore):
19          print("passo dal setter del codice della fattura")
20          if not isinstance(valore, str):
21              raise TypeError("il codice deve essere di tipo stringa")
22          if len(valore) !=3:
23              raise ValueError("lunghezza del codice non valida")
24
25          self._codice = valore
```

Se il valore che si vuole impostare non è una istanza della classe str, solleva una errore di tipo

Se la stringa che si vuole impostare non è esattamente lunga 3 caratteri, solleva un errore di valore

Solo se si superano tutti i controlli, l'istanza viene creata o il dato viene modificato

Facciamo un test generale di funzionamento

```
>>> f1=Fornitore('001','alfa srl')
sono in __init__ di istanza fornitori
passo dal setter del codice della fattura
>>> print(f1)
passo dal getter del codice fattura
codice: 001
nome: alfa srl
>>> f1.__dict__
{'_codice': '001', '_nome': 'alfa srl'}
>>> f1.codice
passo dal getter del codice fattura
'001'
```

non è una stringa

```
>>> f2=Fornitore(1,'alfa srl')
sono in __init__ di istanza fornitori
passo dal setter del codice della fattura
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    f2=Fornitore(1,'alfa srl')
  File "/media/sdh2/LezioniMultimediali/OOP/40-property-getter-e-
    self.codice=c
  File "/media/sdh2/LezioniMultimediali/OOP/40-property-getter-e-
    raise TypeError("il codice deve essere di tipo stringa")
TypeError: il codice deve essere di tipo stringa
>>>
```

deve essere lunga 3 caratteri

```
>>> f2=Fornitore('01','alfa srl')
sono in __init__ di istanza fornitori
passo dal setter del codice della fattura
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    f2=Fornitore('01','alfa srl')
  File "/media/sdh2/LezioniMultimediali/OOP/40-property-getter-e-
    self.codice=c
  File "/media/sdh2/LezioniMultimediali/OOP/40-property-getter-e-
    raise ValueError("lunghezza del codice non valida")
ValueError: lunghezza del codice non valida
>>>
```

Teoricamente, dovrei scrivere un metodo getter e setter per ciascun attributo della classe, ma si intuisce che può diventare un lavoro lungo e tedioso come si può constatare guardando le linee di codice che ho abbozzato e che potete trovare sul mio sito.

Svantaggi di questo metodo

>>> ripetizione di codice: c'è una logica ridondante per accedere e validare ciascun attributo; alla lunga si vengono a creare tanti metodi getter e setter quanti sono gli attributi e questo vuol dire avere una quantità di linee di codice da mantenere che può diventare sproporzionato

Per evitare di scrivere centinaia di righe di codice, cercare di seguire le seguenti linee guida:

✖ Non usare getter e setter se non è necessario (cioè se il dato non deve essere validato)

✔ Scrivi i metodi getter e setter, ma se ti accorgi che esistono metodi setter che fanno più o meno le stesse cose passa senza indugio ai descrittori, dei quali ci occuperemo nei prossimi video