Pattern Recognition
and Applications Lab

La Programmazione ad Oggetti in Python

Docente: Ambra Demontis

Anno Accademico: 2020 - 2021

University of Cagliari,
Italy

Department of Electrical and
Electronic Engineering

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica

La Programmazione ad Oggetti in Python

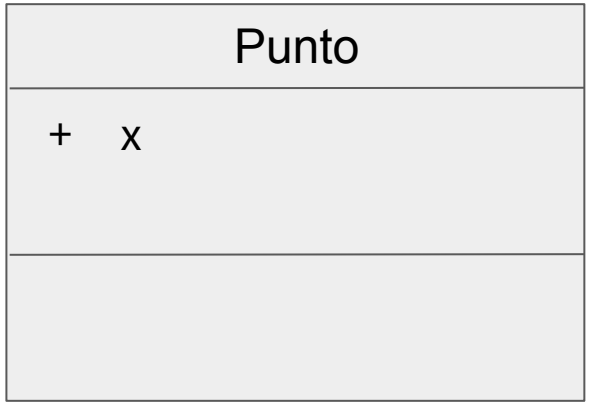
In queste slide vedremo

- I metodi Setter e Getter
- Attributi di sola lettura.
- Come utilizzare setter/getter per:
  - effettuare il controllo dell'input
  - calcolare il valore di un attributo

Getter e Setter

Supponete di voler creare una classe che serve a permettervi di memorizzare il valore della coordinata di un punto in uno spazio ad una sola dimensione:

la classe punto con un attributo pubblico x, che serve a memorizzare il valore della coordinata del punto.



Getter e Setter

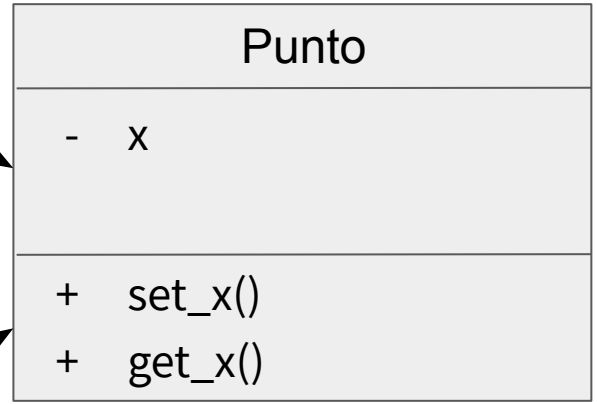
```
class CPunto():

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

oggetto_punto = CPunto(5)

# L’ attributo x è pubblico quindi l'utente può modificarlo direttamente
oggetto_punto.x = oggetto_punto.x + 1
```

Getter e Setter

Supponete ora di voler far si che l'utente non possa modificare (utilizzare) direttamente il valore dell'attributo ma che abbia a disposizione due metodi pubblici che gli permettono di modificare (utilizzare) il valore dell'attributo.

1 Definiamo l'**attributo come privato**.

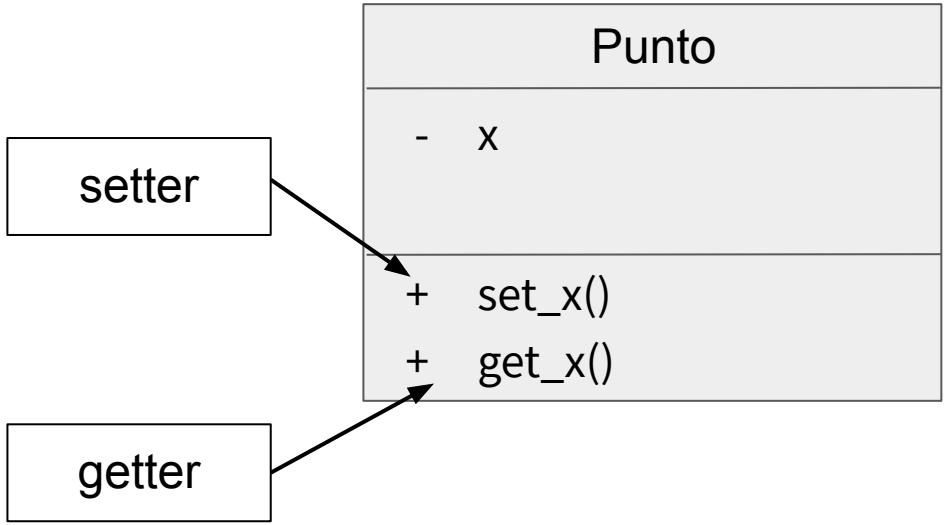
2 Definiamo un'**interfaccia pubblica** per l'attributo:
- un metodo per settarne il valore
- un metodo per utilizzarne il valore



Getter e Setter

I metodi che servono a:

- settare (modificare) il valore di un attributo vengono chiamati **setter**
- modificare (utilizzare) il valore di un attributo vengono chiamati **getter**



Getter e Setter

```
class CPunto():

  def __init__(self, x):
     self._x = x

  def get_x(self):
     return self._x

  def set_x(self, value):
     self._x = value
```

Getter e Setter

```
oggetto_punto = CPunto(5)

# L'utente non direttamente utilizzare/modificare l'attributo in quanto è privato...
# Deve utilizzare i metodi che definiscono la sua interfaccia pubblica.
oggetto_punto.set_x( oggetto_punto.get_x() + 1 )
```

Getter e Setter

Problemi:

1) Il codice diventa meno leggibile
2) Andrebbero modificate tutte le righe di codice nel quale veniva utilizzato l'attributo pubblico…

Soluzione:

Modificare i metodi setter e getter creati in modo che sfruttino la sintassi ad-hoc messa a disposizione da python.

Getter

In Python, la sintassi ad-hoc per definire un metodo getter è la seguente:

```
@property
def <nome_attributo>(self):
  ..
  return self._<nome_attributo>
```

Questo metodo **viene richiamato automaticamente quando un attributo viene utilizzato.**

Setter

In Python, la sintassi ad-hoc per definire un metodo setter è la seguente:

```
@<nome_attributo>.setter
def <nome_attributo>(self, valore):
  ..
  self._<nome_attributo> = valore
```

Questo metodo **viene richiamato automaticamente quando viene assegnato un valore ad un attributo.**

Setter

In Python **il metodo setter può essere creato solo se per lo stesso attributo è stato definito anche il getter** e la dichiarazione del codice del metodo setter va scritta dopo quella del metodo getter.

(Nel caso contrario verrà evidenziato un errore di sintassi e non sarà possibile eseguire il programma).

Getter e Setter

```
class CPunto():

  def __init__(self, x):
    self._x = x

  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, value):
    self._x = value
```

Getter e Setter

```
oggetto_punto = CPunto(5)

# Possiamo utilizzare/modificare l'attributo utilizzando lo stesso codice che avremmo
# utilizzato se l'attributo fosse stato pubblico!
oggetto_punto.x = oggetto_punto.x + 1
```

Getter e Setter

Quando può servire far si che l'utente non possa modificare/utilizzare direttamente il valore dell'attributo ma che abbia a disposizione dei metodi (pubblici) che gli permettono di modificare/utilizzare il valore dell'attributo?

Spesso, quando si realizzano programmi complessi è necessario:

-assicurarsi che un attributo assuma i valori che il programmatore si aspetta.

-calcolare il valore di un attributo.

Quando sopra si può fare creando dei **metodi per utilizzare (settare) il valore di un attributo chiamati getter (setter)**.

Attributi di Sola Lettura

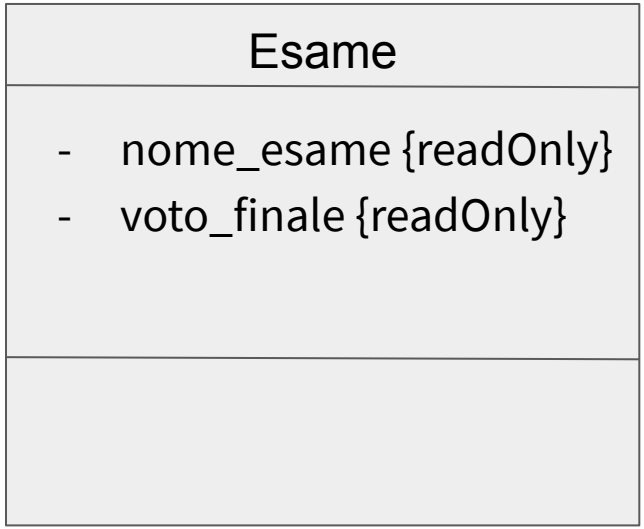
In alcuni casi vogliamo far si che un attributo possa essere utilizzato ma non modificato dall'utente.

Questo tipo di attributi viene chiamato attributi **di sola lettura**.

In quel caso basta definire l'attributo come **privato** e **creare il getter ma non il setter.**

Attributi di Sola Lettura

Ad esempio, supponiamo di avere una classe Esame con due attributi nome_esame e voto_finale che vogliamo vengano settati dall'utente quando l'oggetto viene creato ma poi non debbano poter essere modificati.



Attributi di Sola Lettura

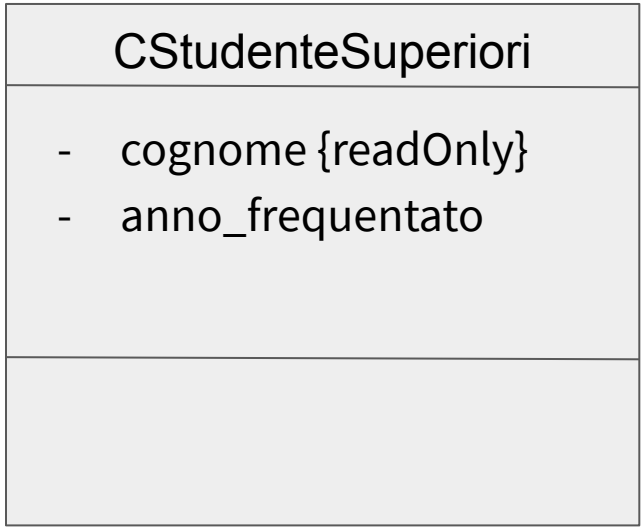
```
class CEsame:
  def __init__(self, nome, voto):
    self._nome = nome
    self._voto = voto

  @property
  def nome(self):
    return self._nome

  @property
  def voto(self):
    return self._voto
```

```
oggetto_esame = CEsame('LPO', 28)
print("nome esame ", oggetto_esame.nome)
print("voto esame ", oggetto_esame.voto)
```

Esercizio Setter e Getter

Creare una classe per memorizzare i dati di uno studente delle superiori.

Questa classe deve avere un attributo di sola lettura chiamato *cognome* e un attributo chiamato *anno_frequentato* che deve essere privato ma modificabile dall'utente.



Esercizio Setter e Getter

```
class CStudente:

  def __init__(self, cognome, anno_frequentato):

    self._cognome = cognome
    self.anno_frequentato = anno_frequentato

  @property
  def cognome(self):
    return self._cognome
```

Esercizio Setter e Getter

```
    @property
    def anno_frequentato(self):
      return self._anno_frequentato

    @anno_frequentato.setter
    def anno_frequentato(self, anno_frequentato):
      self._anno_frequentato = anno_frequentato

oggetto_studente = CStudente("Bianchi", 4)
oggetto_studente.anno_frequentato = 5
```

Setter e Getter per Effettuare il Controllo dell'Input

Consideriamo l'implementazione della classe CLibro sottostante:

```
class CLibro:
  def __init__(self, titolo, prezzo):
    print("inizializza i valori degli attributi del libro")
    self.titolo = titolo
    self.prezzo = prezzo
```

Setter e Getter per Effettuare il Controllo dell'Input

Il valore dell'attributo prezzo viene inizializzato al momento della creazione dell'oggetto e può potenzialmente essere cambiato successivamente.

```
libro_LPO = CLibro("Python 3: OOP", 55)
libro_LPO.prezzo = input("inserisci il nuovo prezzo del libro")
print(libro_LPO.prezzo)
```

Stamperà:

inizializza i valori degli attributi del libro

prezzo libro: 58

Setter e Getter per Effettuare il Controllo dell'Input

Supponiamo di voler far si che ogni volta il valore dell'attributo *prezzo* viene aggiornato, venga controllato se il valore è positivo e altrimenti tale valore venga chiesto input all'utente.

Aggiungiamo un metodo setter e un metodo getter per l'attributo prezzo.

Setter e Getter per Effettuare il Controllo dell'Input

```
@property
def prezzo(self):
    print("usa il prezzo del libro")
    return self._prezzo

@prezzo.setter
def prezzo(self, value):
    print("setta il prezzo del libro")
    while value < 0:
        value = input("Inserisci il prezzo del libro. Il prezzo deve essere maggiore di zero.")
    self._prezzo = value
```

Codice completo nello script:
OOP_in_python_esempio_8_setter_e_getter_1.py

Setter e Getter per Effettuare il Controllo dell'Input

```
libro_LPO = CLibro("Python 3: OOP", 55)
libro_LPO.prezzo = 58
print("prezzo libro:", libro_LPO.prezzo)
```

Ora stamperà:

inizializza i valori degli attributi del libro

setta il prezzo del libro

setta il prezzo del libro

usa il prezzo del libro

prezzo libro: 58

Esercizio sul Controllo dell'Input utilizzando Setter e Getter

Considerando la classe CEsame implementata come segue:

```
class CEsame:
  def __init__(self, nome_esame, voto_esame):
    self.nome_esame = nome_esame
    self.voto_esame = voto_esame
```

Aggiungere un metodo setter che controlla che il voto dell'esame sia compreso tra 0 e 30 e nel caso in cui non lo sia stampi a schermo un messaggio di errore e chieda all'utente di inserire un voto compreso tra 0 e 30.

Esercizio sul Controllo dell'Input utilizzando Setter e Getter

Dovremo creare un metodo setter per l'attributo *voto_esame*:

```
@voto_esame.setter
def voto_esame(self, voto):

  while ( voto < 0 ) or ( voto > 30 ):
      voto = int(input("Il voto deve essere compreso tra 0 e 30, "
                       "reinserisci il voto "))

  self._voto_esame = voto
```

Esercizio sul Controllo dell'Input utilizzando Setter e Getter

Dovremo creare un metodo getter per l'attributo *voto_esame*:

```
@property
def voto_esame(self):
  return self._voto_esame
```

Esercizio sul Controllo dell'Input utilizzando Setter e Getter

Mentre il metodo inizializzatore rimarrà invariato:

```
def __init__(self, nome_esame, voto_esame):

    self.nome_esame = nome_esame
    self.voto_esame = voto_esame
```

Quando viene eseguita l'ultima istruzione di questo metodo viene richiamato il metodo setter.

Esercizio sul Controllo dell'Input utilizzando Setter e Getter

Proviamo ad utilizzare la classe creata:

```
esame_lpo = CEsame("LPO", 50)
print(esame_lpo.voto_esame)
```

Stamperà:

Il voto deve essere compreso tra 0 e 30, reinserisci il voto

28 (supponendo l'utente abbia inserito il voto 28)

Setter e Getter per Calcolare il Valore di un Attributo

Consideriamo la seguente classe:

```
class CEsame:
  def __init__(self, nome_esame, voto_primo_parziale, voto_secondo_parziale):
    self.nome_esame = nome_esame
    self.voto_primo_parziale = voto_primo_parziale
    self.voto_secondo_parziale = voto_secondo_parziale

  def calcola_voto_finale(self):
    self._voto_finale = (self.voto_primo_parziale + self.voto_secondo_parziale) / 2
    return voto_finale
```

Il voto finale viene calcolato e non memorizzato (va ricalcolato ogni volta che serve.)

Setter e Getter per Calcolare il Valore di un Attributo

Potremmo considerare il voto_finale come un attributo che necessita dei calcoli. Potremmo calcolarlo nel metodo __init__?

```
class CEsame:

  def __init__(self, nome_esame, voto_primo_parziale, voto_secondo_parziale):
     self.nome_esame = nome_esame
     self.voto_primo_parziale = voto_primo_parziale
     self.voto_secondo_parziale = voto_secondo_parziale
     self.voto_finale = (self.voto_primo_parziale +
                         self.voto_secondo_parziale)/ 2
```

Setter e Getter per Calcolare il Valore di un Attributo

Problema: cosa succederebbe se uno dei voti venisse modificato?

```
esame_analisi = CEsame('LPO', 28, 29)
print(esame_analisi.voto_finale)
esame_analisi.voto_secondo_parziale = 30
print(esame_analisi.voto_finale)
```

Stamperebbe:

28.5

28.5

Setter e Getter per Calcolare il Valore di un Attributo

Problema: cosa succederebbe se uno dei voti venisse modificato?

```
esame_analisi = CEsame('LPO', 28, 29)
print(esame_analisi.voto_finale)
esame_analisi.voto_secondo_parziale = 30
print(esame_analisi.voto_finale)
```

Stamperebbe:

28.5

28.5

Il voto finale non verrebbe aggiornato e non sarebbe coerente, quindi non è una soluzione possibile!

Setter e Getter per Calcolare il Valore di un Attributo

Supponiamo che l’utente debba poter modificare il voto del primo o del secondo parziale, ad esempio perché gli studenti hanno la possibilità di effettuare una seconda volta il primo o il secondo parziale a loro scelta.

Setter e Getter per Calcolare il Valore di un Attributo

Si può utilizzare il seguente trucco:

-si creano dei setter e getter per gli attributi dai quali il voto_finale dipende (*voto_primo_parziale*, *voto_secondo_parziale*)

-si fa in modo che il loro setter ricalcoli il valore dell'attributo *voto_finale*

-si definisce *voto_finale* come attributo di sola lettura (si definisce il suo getter ma non il setter)

Setter e Getter per Calcolare il Valore di un Attributo

```
esame_analisi = CEsame('LPO', 28, 29)
print(esame_analisi.voto_finale)

esame_analisi.voto_secondo_parziale = 30

print(esame_analisi.voto_finale)
```

Stamperà:

28.5

29.0

Il voto finale è stato correttamente aggiornato!

Getter per Evitare la Modifica del Valore di un Attributo

```
class CEsame:

    def __init__(self, nome_esame, voto_primo_parziale, voto_secondo_parziale):
        self.nome_esame = nome_esame
        self._voto_primo_parziale = voto_primo_parziale
        self._voto_secondo_parziale = voto_secondo_parziale
        self._voto_finale = self.calcola_voto_finale()

    def _calcola_voto_finale(self):
        voto_finale = (self.voto_primo_parziale + self.voto_secondo_parziale) / 2
        return voto_finale
```

Getter per Evitare la Modifica del Valore di un Attributo

```
@property
def voto_primo_parziale(self):
  return self._voto_primo_parziale

@voto_primo_parziale.setter
def voto_primo_parziale(self, value):
  self._voto_primo_parziale = value
  self._voto_finale = self._calcola_voto_finale()
```

Getter per Evitare la Modifica del Valore di un Attributo

```
@property
def voto_secondo_parziale(self):
  return self._voto_secondo_parziale

@voto_secondo_parziale.setter
def voto_secondo_parziale(self, value):
  self._voto_secondo_parziale = value
  self._voto_finale = self._calcola_voto_finale()
```

Getter per Evitare la Modifica del Valore di un Attributo

```
@property
def voto_finale(self):
  return self._voto_finale
```

Setter e Getter per Calcolare il Valore di un Attributo

Questa tecnica si utilizza spesso in applicazioni che richiedono calcoli computazionalmente onerosi. Es, in applicazioni legate al Machine Learning.

Gli algoritmi di Machine Learning imparano a svolgere compiti da degli esempi (un insieme di dati) e per imparare a svolgerli correttamente hanno bisogno di tanti dati.

Ogni volta che dobbiamo fare un'operazione che coinvolge questi dati dobbiamo farla n volte (dove n è il numero di dati), con n molto grande.

Esercizio Setter e Getter (classe Stringa)

Supponete, in un progetto che state realizzando, di avere spesso la necessità di calcolare la lunghezza di una stringa. Un collega vi chiede quindi di creare la Classe CStringa con un attributo *stringa* e l'attributo di sola lettura *lunghezza_stringa*.

L'attributo *stringa* deve poter essere modificato e in quel caso l'attributo *lunghezza_stringa* deve venire ricalcolato in modo che sia coerente con la nuova stringa.

Esercizio Setter e Getter (classe Stringa)

Per calcolare la lunghezza di una stringa potete utilizzare le funzione *len*.

Esempio:

```
nome_e_cognome = "Anna Bianchi"
print(len(nome_e_cognome))
```

Esercizio Setter e Getter (classe Stringa)

```
class CStringa:
  def __init__(self, stringa):
    self._lunghezza_stringa = None
    self.stringa = stringa

  @property
  def stringa(self):
    return self._stringa

  @stringa.setter
  def stringa(self, value):
    self._stringa = value
    self._lunghezza_stringa = len(value)

  @property
  def lunghezza_stringa(self):
    return self._lunghezza_stringa
```

Esercizio Setter e Getter (classe Stringa)

```
oggetto_stringa = CStringa("LPO")
print(oggetto_stringa.stringa, oggetto_stringa.lunghezza_stringa)

oggetto_stringa.stringa = "LPO con Python"
print(oggetto_stringa.stringa, oggetto_stringa.lunghezza_stringa)
```

Stamperà:

LPO 3

LPO con Python 14