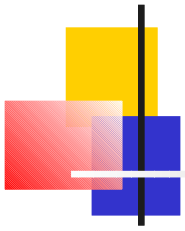
DIEE - Università degli Studi di Cagliari

Programmazione Orientata agli Oggetti e Scripting in Python

Paradigma ad Oggetti - 2

Alessandro Orro – DIEE Univ. di Cagliari



ereditarietà: metodi

- Il meccanismo di eredità dei metodi è simile a quello del C++ e del Java:
  - Tutti i metodi di una classe base sono ereditati dalla classe figlia.
  - I metodi statici e di classe si ereditano con le stesse regole dei metodi di istanza.
  - Se ridefinisco un metodo, la chiamata al relativo metodo della classe base non è automatica.
  - Il metodo di istanza `__init__` non fa eccezione
    - Si comporta diversamente dai costruttori di C++ e Java

ereditarietà: metodi

```
class B(object):
    def m(self): print 'B.m()'
    def sm(): print 'B.sm()'
    sm = staticmethod(sm)
    def cm(cls): print 'B.cm()'
    cm = classmethod(cm)
class D(B):
    pass
D().m();
D().cm(); D.cm()
```

ereditarietà: attributi

- Gli attributi sono aggiunti/cancellati in maniera dinamica per cui non si può parlare di vera e propria eredità degli attributi.
- Una classe può avere o meno gli attributi della classe base a seconda di come si comporta l'inizializzatore.
  - L'inizializzatore, come tutti i metodi di istanza, viene ereditato.

ereditarietà: attributi

- In questo caso **`B.__init__`** viene ereditato per cui in fase di costruzione ad ogni istanza di D verranno aggiunti gli stessi attributi di B.

```
class B(object):
  def __init__(self):
    self.x=10
class D(B):
  pass
D().x
```

ereditarietà: attributi

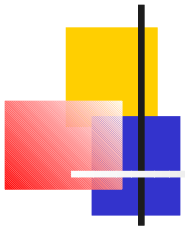
- Anche in questo caso gli attributi vengono "ereditati" in quanto `D.__init__` è ridefinito ma richiama `B.__init__`.

```
class B(object):
  def __init__(self):
    self.x=10
class D(B):
  def __init__(self):
    B.__init__(self)
D().x
```

ereditarietà: attributi

- In questo caso invece non vengono aggiunti attributi in quanto **`B.__init__`** non è richiamato automaticamente.

```
class B(object):
  def __init__(self):
    self.x=10
class D(B):
  def __init__(self):
    pass
D().x # errore
```



ereditarietà: property

- Le property vengono ereditate come se fossero metodi anziché attributi
  - richiamare una property è equivalente chiamare i metodi fget/fset/fdel
  - non dipendono in maniera diretta dal comportamento di **`__init__`**.

```
X = property(fget, fset, fdel)
```

ereditarietà: property

```
class Persona(object):
  def __init__(self,n):
    self.nome = n
  def getnome(self):
    return self.nome
  Nome = property(getnome) # read-only

p = Persona('Mario')
print p.Nome
```

(continua)

ereditarietà: property

```
class Medico(Persona):
  def __init__(self):
    pass

m = Medico()
print m.Nome # errore!!!
             # Nome esiste ma nome no!
```

ereditarietà: property

- Questo invece funziona. Perché ?

```
class Persona(object):
  def __init__(self,n):
    self.nome = n
  def getnome(self): return self.nome
  def setnome(self,n): self.nome = n
  Nome = property(getnome,setnome)
class Medico(Persona):
  def __init__(self): pass

m = Medico()
m.Nome = 'Mario'
print m.Nome # corretto
```

attributi e metodi speciali

- Tutti gli attributi/metodi che iniziano e finiscono con __ hanno un significato speciale in Python.
  - Alcuni gli abbiamo già visti in dettaglio:
    - `__new__`
    - `__init__`
    - `__mro__`
  - altri ...
    - `__class__`
    - `__name__`
    - `__dict__`
    - `__slots__`
    - `...`

attributi e metodi speciali

- **`__class__`**

E' un attributo che contiene un riferimento alla classe a cui appartiene l'oggetto.

- Tutti gli oggetti, comprese le classi, hanno l'attributo **`__class__`**

```
class P(object):
  pass
p = P()
p.__class__
```

attributi e metodi speciali

- **`__name__`**

E' un attributo che contiene il nome di una classe.

- Tutte le classi, comprese le built-in, hanno l'attributo **`__name__`**

```
class P(object):
  pass
P.__name__       # 'P'
int.__name__     # 'int'
P().__name__     # errore
```

attributi e metodi speciali

- **`__dict__`**

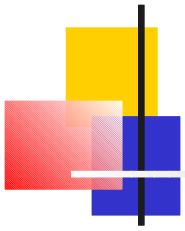
E' un attributo molto importante che contiene un dizionario che associa i nomi degli attributi di un oggetto al loro valore.

- Tutti gli oggetti, comprese le built-in, hanno l'attributo **`__dict__`**.
- aggiungere/rimuovere un attributo da un oggetto è equivalente ad aggiungere/rimuovere un elemento dal dizionario **`__dict__`**.
- **`__dict__`** non contiene le property!

attributi e metodi speciali

- Esempio

```
class P(object):
  def __init__(self):
      self.x=10
p = P()
p.__dict__    # {'x': 10}
p.__dict__['y'] = 20
p.y           # 20
del p.__dict__['y']
```



attributi e metodi speciali

- **`__slots__`**

E' possibile limitare il nome degli attributi a quelli presenti nell'attributo di classe **`__slots__`**.

- Migliora l'efficienza.
- Non è ereditabile

```
class P(object):
    __slots__ = 'a', 'b'

p = P()
p.x = 10   # errore
```

metodo per chiamata a funzione

- **`__call__(self,...)`**

E' un metodo di istanza che corrisponde all'operatore di chiamata a funzione **`()`**.

- Permette di creare *"oggetti funzione"* come in C++
- Tutte le funzioni hanno **`__call__`** (chiamata)
- Tutte le classi hanno **`__call__`** (costruttore)
- Gli altri oggetti possono avere **`__call__`** se lo definisco in maniera esplicita.

metodo per chiamata a funzione

- Gli oggetti funzione sono più flessibili delle funzioni normali perché possono avere uno stato interno.

```
class Retta(object):
  def __init__(self,P):
    self.P = P
  def __call__(self,x):
    return self.P * x
retta = Retta(10)   # oggetto funzione
print retta(1.2)
```

metodo per chiamata a funzione

- In alternativa posso aggiungere dinamicamente attributi ad una funzione normale in quanto è a tutti gli effetti un oggetto:

```
def retta(x):
  return retta.P * x
retta.P = 2

print retta(10) # 20
```

metodi per la rappresentazione

- **`__str__`**
  - Serve per costruire una rappresentazione “concisa” dell'oggetto
  - E' utilizzata quando l'oggetto è stampato con **`print`** o convertito in stringa con **`str`**.

- **`__repr__`**
  - Serve a costruire una rappresentazione “completa” dell'oggetto.
  - La stringa generata deve permettere di ricostruire completamente l'oggetto.

metodi per i contenitori

- Esistono alcuni metodi utili per definire il comportamento di classi “contenitore”
  - **`__len__`**: chiamata dalla funzione **`len`**.
  - **`__contains__`**: operatore **`in`**.
  - **`__iter__`**: comportamento come iteratore (**`for`**)
  - **`__getitem__`**: accesso in lettura ad un elemento
  - **`__setitem__`**: accesso in scrittura ad un elemento
  - **`__delitem__`**: cancellazione di un elemento
- Tutti contenitori built-in (**`str`**, **`list`**, **`dict`**) hanno questi metodi già definiti.

metodi per i contenitori

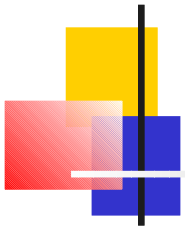
- **`__getitem__(self, key)`**
  `__setitem__(self, key, value)`
  `__delitem__(self, key)`

  - Queste funzioni servono per leggere, scrivere e cancellare un elemento del contenitore.
  - L'elemento viene identificato tramite una chiave **key**.
  - La chiave può essere per esempio un indice (liste), un valore (dizionari), un oggetto slice

metodi per i contenitori

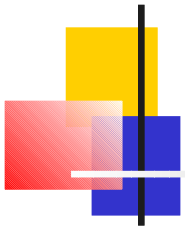
```
L = [1,2,3]
L.__getitem__(1) # L[1]
L.__setitem__(0,'a') # L[0]='a'
L.__getitem__(slice(1,2)) # L[1:2]

D = {'a': 0, 'b': 2}
D.__delitem__('a') # del D['a']
```



metodi per i contenitori

- slice
  - E' una classe built-in si usa per rappresentare in maniera compatta delle sequenze di indici.
  - **`slice(start,end,step)`** rappresenta tutti gli indici che partono da **`start`** e terminano in **`end-1`** con passo **`step`**.
  - Le classi **`str`** e **`list`** hanno già implementato il meccanismo di slicing.
  - Quando viene creato un nuovo contenitore posso ridefinire il comportamento dello slicing.



metodi per i contenitori

- Vediamo alcuni esempi:
  - Lista circolare
  - Set
  - Contatore di caratteri

metodi per l'accesso agli attributi

- Nella maggior parte dei casi l'accesso agli attributi di un'istanza può essere gestito con le property.
- Il Python presenta un ulteriore meccanismo molto più flessibile per l'accesso agli attributi che consiste nel definire alcuni metodi speciali:
  - **`__getattr__`**
  - **`__setattr__`**
  - **`__delattr__`**

metodi per l'accesso agli attributi

- **`__getattr__(self,name)`**
  - Viene chiamato quando si accede all'attributo di nome **`name`** e questo non viene trovato in **`self`**.
- **`__setattr__(self,name,value)`**
  - Viene chiamato quando si cerca di settare l'attributo di nome **`name`** con un valore **`value`**.
- **`__delattr__(self,name)`**
  - Viene chiamato quando si cerca di eliminare l'attributo di nome **`name`**.

metodi per l'accesso agli attributi

```
p.x         chiama p.__getattr__('x')

p.x = 100   chiama p.__setattr__('x',100)

del p.x     chiama p.__delattr__('x')
```

metodi per l'accesso agli attributi

- Quando si ridefiniscono questi metori bisogna sempre fare riferimento all'attributo **`__dict__`** in modo da evitare chiamate ricorsive.

- Esempi:
  - Struttura con campi fissi
  - Attributo non sensibile alle maiuscole

metodi per la comparazione

- Ci sono diversi metodi che implementano gli operatori matematici di comparazione
  - `__eq__` `==`
  - `__ne__` `!=`
  - `__ge__` `>=`
  - `__gt__` `>`
  - `__le__` `<=`
  - `__lt__` `<`
  - `__cmp__` `comparazione per differenza`

metodi per la comparazione

- **`__eq__`** e **`__ne__`**
  - Ogni oggetto ha un id univoco durante l'esecuzione del programma (solitamente l'indirizzo di memoria)
  - Normalmente gli operatori == e != controllano se due variabili sono uguali/differennti per id cioè se si riferiscono o meno allo stesso oggetto
  - Se riscrivo questi metodi posso effettuare un controllo sul valore anziché sull'id

metodi per la comparazione

```
class P(object):
  pass
class Persona(object):
  def __init__(self,n):
    self.nome=n
  def __eq__(self,other):
    return self.nome==other.nome

  p1,p2 = P(),P()
  print p1==p2 # diversi
  p1,p2=Persona('Ale'),Persona('Ale')
  print p1==p2 # uguali
```

metodi per i tipi numerici

- Ci sono infine diversi metodi che implementano gli operatori matematici più comuni
  - `__add__` `+`
  - `__sub__` `-`
  - `__mul__` `*`
  - `__div__` `/`
  - `__pow__` `**`
  - `...`

metodi per i tipi numerici

- Operatori “aumentati”
  - `__iadd__` `+=`
  - `__isub__` `-=`
  - `__imul__` `*=`
  - `__idiv__` `/=`
  - `__ipow__` `**=`
  - `...`

metodi per i tipi numerici

- In tutto sono circa 50:
  - Operatori booleani
  - Operatori di shift
  - Operatori di conversione
  - ...
  - *(Vedere il manuale di riferimento)*