Programmazione Orientata agli Oggetti e Scripting in Python

Paradigma ad Oggetti - 1

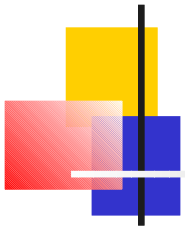
Alessandro Orro – DIEE Univ. di Cagliari

Introduzione

- Il Python è un linguaggio che supporta diversi paradigmi/stili di programmazione:
  - procedurale
  - funzionale
  - a oggetti
  - meta-programmazione
  - scripting

Introduzione

- In questa lezione e nella prossima vedremo:
  - le caratteristiche fondamentali del Python in una prospettiva Object Oriented
  - le differenze con altri due linguaggi di programmazione a oggetti: C++ e Java
  - alcuni esempi pratici



Caratteristiche fondamentali

- in breve ...
  - *ereditarietà*: multipla
  - *dispatching*: singolo (un oggetto è proprietario del metodo, eventualmente tramite la classe)
  - *binding*: dinamico (e il controllo di esistenza di un metodo/attributo è fatto a run-time)
  - *information hiding*: privatezza "debole" e property
  - *polimorfismo*: per inclusione (overriding e riuso) e coercion (esplicita e implicita)

Caratteristiche fondamentali

- prima di andare avanti, vediamo la differenza fra oggetti e riferimenti

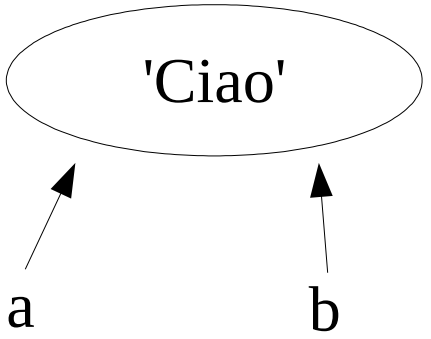
  - gli *oggetti* hanno un tipo che non può mai cambiare.
    Da questo punto di vista il Python è fortemente tipizzato

  - i *riferimenti* (variabili/attributi) sono delle etichette che possono essere "legate" a oggetti di qualsiasi tipo.
    I riferimenti non hanno tipo

Caratteristiche fondamentali

- Esempio

```
a = 10
a = 'Ciao'
b = a
```

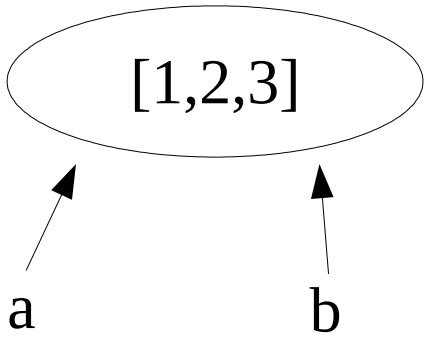


- Il riferimento **a** è legato prima ad un intero e poi a una stringa
- **b** e **a** sono riferimenti allo stesso oggetto in memoria che è di tipo stringa.

Caratteristiche fondamentali

- Se l'oggetto [1,2] viene modificato tramite **a** la modifica viene vista anche da **b**

```
a = [1,2]
b = a
a += [3]
```

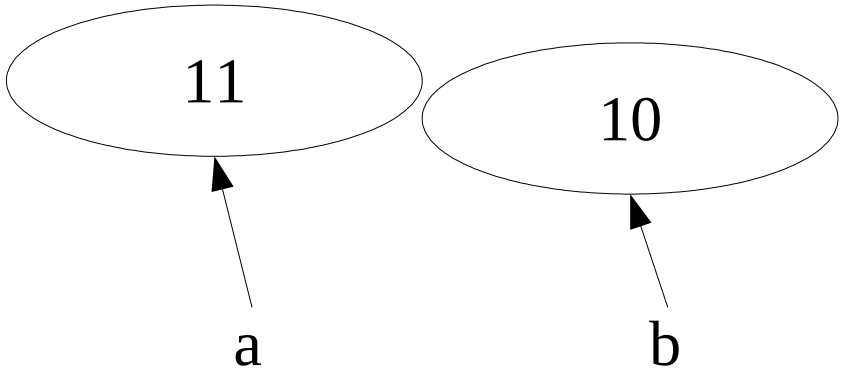


  - L'operatore += aggiunge un elemento alla lista senza creare un nuovo oggetto

Caratteristiche fondamentali

- Se si prova a modificare 10 tramite **a** la modifica viene non vista da **b**

```
a = 10
b = a
a += 1
```



  - L'operatore += crea una nuova copia dell'oggetto e lo assegna ad **a**

Caratteristiche fondamentali

- Tutti i tipi elementari (int,float,str) creano copie di oggetti quando si prova a modificarli. Si tratta di oggetti *immutabili*. Altri tipi built-in (list,dict) sono invece *mutabili*.

```
a = 'Ciao'
a += ' a tutti' # crea una nuova striga
```

- Per tutti gli altri oggetti definiti dall'utente si può scegliere il comportamento scrivendo gli opportuni operatori.

Classi

- In Python ci sono vari modi di vedere le classi, uno dei più comuni è l'*object-factory*: oggetti che permettono di costruire altri oggetti, ma non riferimenti.

- Per creare un riferimento ad un oggetto basta semplicemente assegnarlo

```
a = list([1,2,3])
```

Classi

- La classe più semplice: nessun metodo e nessun attributo

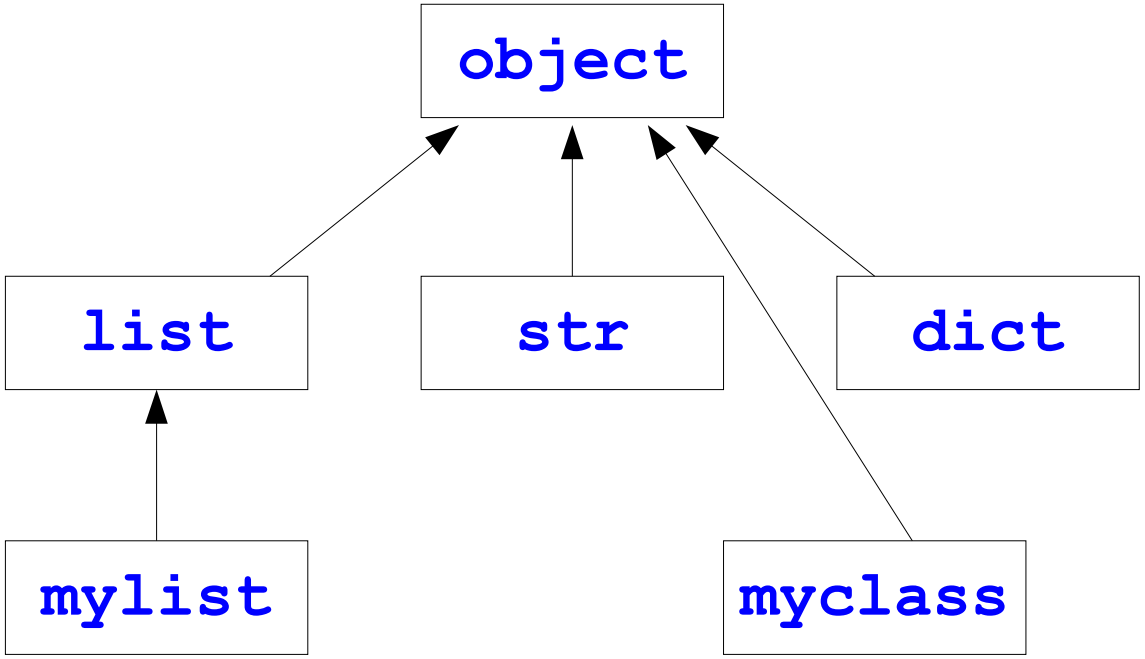
```
class MyClass(object):
  def __init__(self):
    pass

myobj = MyClass() # costruttore
```

Classe object

- **object** è la classe di base da cui ereditano tutti i tipi built-in (list,dict,...).
- tutte le classi devono ereditare da **object**, anche se indirettamente

```
class MyClass(object):
   ...
class MyList(list):
   ...
```

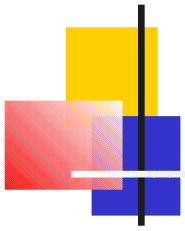
Classe object



Classe object

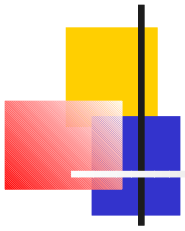
- al momento (ver 2.3.3) il Python si trova in uno stato di transizione:
  - classi *Old-Style*: non ereditano da object
  - classi *New-Style*: ereditano esplicitamente da object

- senza entrare nei dettagli:
  - le new-style hanno delle caratteristiche aggiuntive importanti
  - in futuro ci saranno soltanto le new-style
  - per evitare confusione faremo riferimento solo alle classi new-style



creazione/inizializzazione

- A differenza di altri linguaggi (C++/Java), gli oggetti sono costruiti in due passi successivi:
  - *creazione*
    funzione membro statica **__new__**
  - *inizializzazione*
    funzione membro di istanza **__init__**



__new__

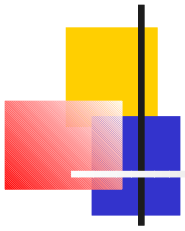
- La funzione **__new__** crea e restituisce una nuova istanza della classe non ancora inizializzata.
- Viene sempre richiamata automaticamente in fase di creazione di un'istanza di classe. Se non viene trovata si risale la gerarchia fino a **object**.

```
myobj = MyClass()
```

__new__

- Sovrascrivendo la funzione **__new__** si possono ottenere dei comportamenti molto particolari. Per esempio:
  - **__new__** può restituire un oggetto già creato.
  - **__new__** può costruire un oggetto di una classe diversa da quella attuale

- Solo in rari casi si ha effettivamente l'esigenza di riscrivere **__new__**.
  - Useremo praticamente sempre la **__new__** fornita dalla classe **object**.



__init__

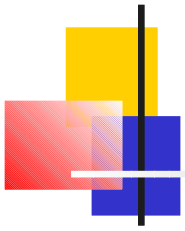
- La funzione **__init__** inizializza l'istanza (oggetto) appena creata da **__new__**.
- Viene chiamata immediatamente dopo **__new__**.
- Nel caso più semplice, l'inizializzazione aggiunge alcuni attributi all'istanza stessa.

```
class MyClass(object):
  def __init__(self):
    self.x=10
myobj = MyClass()
print myobj.x # stampa 10
```

__init__

- Ovviamente **__init__**, come tutte le funzioni, può avere un numero arbitrario di parametri in ingresso che possono essere usati per l'inizializzazione.

```
class MyClass(object):
  def __init__(self,x,y):
    self.x=x
    self.y=y
```



attributi di istanza

- l'aggiunta/rimozione di attributi può avvenire in qualunque momento durante la vita dell'oggetto

```
class MyClass(object):
  def __init__(self):
    self.x=10   # aggiunta di x
myobj = MyClass()
myobj.y=20      # aggiunta di y
del myobj.x     # rimozione di x
```

attributi di istanza

- *C++ e Java*
  gli oggetti hanno un numero e un tipo di attributi predeterminati dalla classe.

- *Python*
  oggetti della stessa classe possono avere attributi differenti
  __init__ serve solo per inizializzare ma non vincola il numero di attributi di un oggetto.

attributi di istanza

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

o1,o2 = MyClass(), MyClass()
o2.y=20; del o2.x
print o1.__class__, o2.__class__

<class '__main__.MyClass'>
<class '__main__.MyClass'>
```

attributi di classe

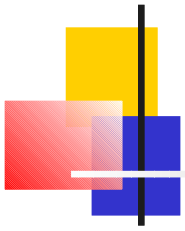
- In Python le classi sono a loro volta oggetti e come tali posso aggiungere ad esse degli attributi
- Dal punto di vista della programmazione a oggetti si parla di *attributi di classe*

```
class P(object):
   a = 20
P.b = 10
print P.a, P.b # a,b sono equivalenti
```

attributi di classe

- Agli attributi di classe si può accedere anche tramite le singole istanze

```
class P(object):
  a=[1,2,3]
P.a += [4];    print P.a
P().a += [5]; print P.a
```



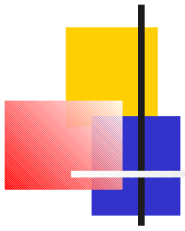
metodi di istanza

- I metodi di istanza sono degli attributi di istanza "legati" a delle funzioni.
- Il modo più semplice di definirli è quello di inserire una funzione dentro il corpo della classe che ha come primo argomento **self**

```
class P(object):
   ...
   def method(self):
     print 'metodo di P'
```

metodi di istanza

- il primo argomento di ogni metodo di istanza è un riferimento all'istanza stessa
  - deve essere sempre indicato nel metodo
  - per convenzione viene chiamato **self** ma è solo una convenzione (posso dare un altro nome).
  - è l'analogo del **this** in C++/Java
  - il suo utilizzo è obbligatorio se si vuole far riferimento ad attributi/metodi dell'istanza dentro un il codice di un metodo



metodi di istanza

- Un metodo di istanza è un attributo dell'istanza che incapsula un attributo di classe

```
class C(object):
   ...
   def m(self): pass
x=C()
```

  - **x.m()** equivale **C.m(x)**
  - attenzione: **C.m(10)** è sbagliato

metodi di classe

- Un metodo di classe è un metodo che si applica alla classe stessa.
- Il parametro **cls** viene passato implicitamente e rappresenta la classe.

```
class P(object):
   ...
   def cmethod(cls):
     print cls.__name__, 'ha chiamato
cmethod'
   cmethod = classmethod(cmethod)
P.cmethod()
```

metodi di classe

- I metodi di classe possono essere chiamati sia dalla classe che da un'istanza della classe.
- In ogni caso si applicano alla classe: viene passato come parametro implicito la classe
- non esiste un equivalente in C++/Java

```
p=P()
P.cmethod()    # si applica alla classe P
p.cmethod()    # si applica alla classe P
```

metodi di classe

- Un metodo di classe ovviamente può modificare la classe stessa.

```
class P(object):
   ...
   def cmethod(cls):
     print cls.__name__, 'ha chiamato
cmethod'
     cls.testo='Ciao'
   cmethod=classmethod(cmethod)
```

metodi statici

- Un metodo statico si comporta come una funzione globale dentro il namespace della classe.
- E' l'analogo dei metodi static del C++/Java

```
class P(object):
  ...
  def smethod():
    print ''
  smethod=staticmethod(smethod)
P.smethod()
```

metodi statici

- I metodi statici possono essere chiamati solo dalla classe e non dall'istanza
- Non si riferiscono alla classe.
- Non viene passato nessun parametro implicito al metodo!

```
P.smethod()
P().smethod() # sbagliato
```

ereditarietà

- Il Python supporta il meccanismo di ereditarietà multipla

```
class C(B):
   ...
class C(B1,B2,...):
   ...
```

- la classe a livello più alto è sempre **object**

ereditarietà

- *lista di precedenza*
  Quando viene chiamato un metodo per un oggetto, questo viene cercato in maniera ordinata in una lista di classi
- La lista di precedenza è univocamente determinata dalla gerarchia delle classi stesse.
- questa lista (tupla) è memorizzata nell'attributo di classe read-only **__mro__**

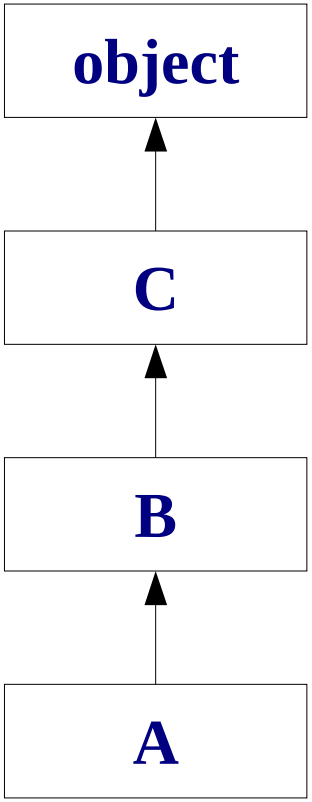
ereditarietà

- Il caso di eredità singola è quello più semplice

```
class C(object): pass
class B(C): pass
class A(B): pass
print [cls.__name__ for cls in  A.__mro__]

['A', 'B', 'C', 'object']
```

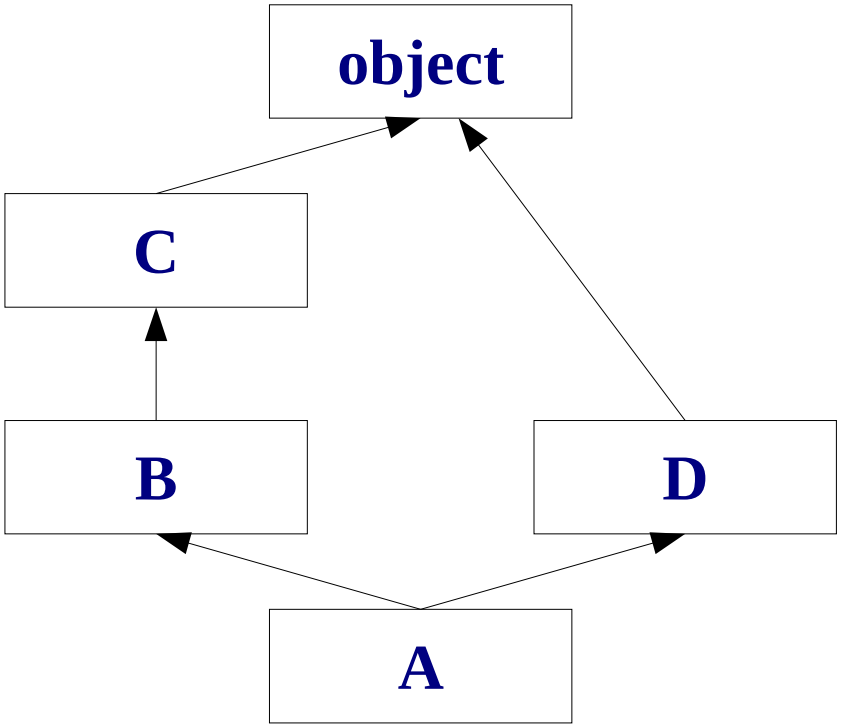
ereditarietà

- eredità semplice



ereditarietà

- Nel caso di eredità multipla la lista di precedenza viene valutata procedendo tipicamente left-to-right e depth-first.



ereditarietà

- L'eredità multipla può portare a delle gerarchie molto complesse in cui non è semplice determinare la lista di precedenza nella chiamata dei metodi
- vedremo in seguito come si gestiscono questi casi in Python

ereditarietà

- I meccanismi di *overriding* del Python sono simili a quelli di C++/Java
- ogni metodo riscritto in una classe va a nascondere quello delle classi base
- Il modo più semplice di richiamare un metodo di una classe base è

```
def method(self,attr):
    Base.method(self,attr)
```

ereditarietà

- Spesso un metodo di una sottoclasse deve delegare una parte del lavoro all'analogo metodo superclasse.
- Un caso tipico è quello dell'**__init__**. Infatti non viene richiamato automaticamente quello della classe base ma deve essere fatto esplicitamente

```
class MyList(list):
  def __init__(self,L):
    list.__init__(self,L)
```

ereditarietà

- Questa soluzione può portare problemi nel caso di eredità multipla.
- E' possibile che alcuni metodi della classe base vengano chiamati più di una volta.
  - spesso non è la cosa voluta

- soluzione: funzione built-in **super**.
- Vedremo in seguito questi aspetti

information hiding

- In Python esiste un concetto di privatezza molto diverso rispetto a quello di altri linguaggi.
  - in C++/Java: public/private/protected

- In Python gli attributi sono normalmente pubblici.
- Gli attributi che iniziano con __ sono però trattati in maniera speciale dall'interprete.

information hiding

- gli *attributi di sistema* iniziano e finiscono con __ e servono per rappresentare funzioni e operatori particolari (**__new__**, **__init__**, **__add__**, **__call__**)
- gli *attributi privati* iniziano con __. L'interprete rinomina questi attributi usando il nome della classe di appartenenza.

```
__x  ->  _ClassName__x
```

- Si possono creare attributi "veramente" privati con tecniche avanzate (vedremo in seguito).

information hiding

```
class MyClass(object):
  def __init__(self):
    self.__x=10   # membro privato
  def getX(self):
    return self.__x
  def setX(self,value):
    self.__x=value
o = MyClass()
print o.getX()
print o._MyClass__x
```

information hiding

- La tecnica delle *property* è una delle più potenti in Python per realizzare l'information hiding

- Sintassi

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

information hiding

```
class MyClass(object):
  def __init__(self):
    self.__x=10
  def getX(self):
    return self.__x
  def setX(self,value):
    self.__x=value
  x=property(fget=getX, fset=setX)

o = MyClass(object)
o.x = 10  # richiama setX
print o.x # richiama getX
```

information hiding

- Nell'esempio precedente non ho specificato fdel per cui l'attributo x sarà non cancellabile
- In maniera analoga posso definire attributi non modificabili (read-only) e non accessibili.

```
class MyClass(object):
  def __init__(self):
    self.__x=10 # non modificabile
  def getX(self):
    return self.__x
  x=property(fget=getX)
o = MyClass(object)
o.x = 10  # errore
```