OOP - Object Oriented Programming

- Python linguaggio **versatile**, supporta, sia programmazione procedurale/funzionale sia, programmazione a oggetti (in maniera non esclusiva)
- *Programmazione orientata agli oggetti*:
  - Paradigma di programmazione basato sul concetto di ***oggetti***, che sono strutture dati che contengono dati, chiamati ***attributi***, e funzioni, chiamate ***metodi***.
  - Vantaggi:
    - supporto **naturale alla modellazione software degli oggetti del mondo reale** o del modello astratto da riprodurre
    - facile gestione e manutenzione di **progetti di grandi dimensioni**
    - l'organizzazione del codice sotto forma di classi favorisce la **modularità e il riuso** di codice

Classi e oggetti

- La *classe* e' la **rappresentazione astratta** di un oggetto
  - Ogni classe e' composta da attributi, caratteristiche e proprieta' degli oggetti, e metodi, procedure che operano sugli attributi
- Un *oggetto* è una **istanza di una classe**

- Python segue il modello **object-factory**: ogni classe e' usata per *creare nuovi oggetti, sue istanze*

- Ereditarieta': è possibile creare nuove classi a partire da quelle esistenti, estendendole con caratteristiche aggiuntive

Definizione di una classe (old-style)

```
# Definisci una nuova classe MyClass
class MyClass:
      pass

class MyClass():
      pass

# Genera una nuova istanza di MyClass
myObjectInstance = MyClass()
```

- Tale metodo di dichiarazione di classe e' detto **old-style**!
- In python 3.0 tale metodo di dichiarazione di classe non sara' piu' ammesso

Definizione di una classe (new-style)

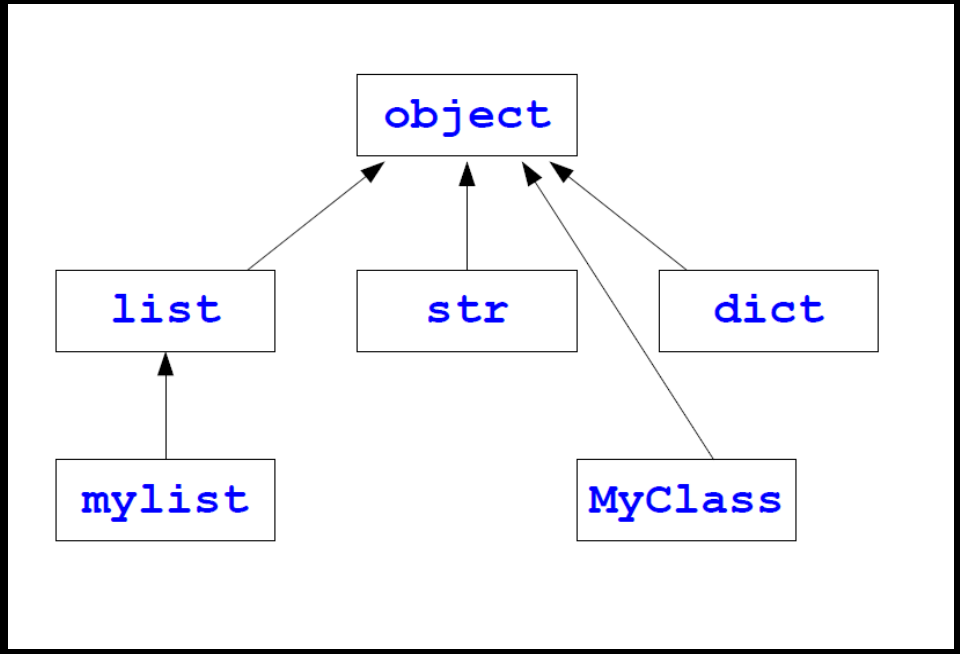
```
# Definisci una nuova classe MyClass
class MyClass(object):
     pass

# Genera una nuova istanza di MyClass
myObjectInstance = MyClass()
```

- In tale metodo detto **new-style**, tutte le classi ereditano dalla classe object!
- In python 2.7 entrambi new-style e old-style sono ammessi

Classe object

- ***object*** e' la classe base da cui ereditano tutti I tipi built-in
- Una classe eredita direttamente o indirettamente da **object**

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



Commenti

- La dichiarazione di una classe puo’ essere seguita da una stringa opzionale che documenta la classe stessa.
- Questa verra’ visualizzata dalla funzione *help*

```
class MyClass(object):
      “””
            Decrizione della classe
      “””

help(MyClass)
```

Metodi di istanza

- Funzioni definiti dentro il corpo di una classe invocabili solo sulle istanza della stessa
- Il primo argomento, solitamente per convenzione ***self***, e' il riferimento all'istanza su cui il metodo e' invocato

```
class MyClass(object):
        def method(self):
                print 'metodo di istanza'

ist = MyClass()

ist.method()
```

Metodi di istanza

- Dato che la definizione stessa di una classe in python e' un oggetto e come tale puo' essere riferito metodi di istanza possono essere invocati cosi:

```
class MyClass(object):
        def method(self):
                print 'metodo di istanza'

ist = MyClass()

ist.method()

#Il primo argomento deve essere un'istanza di tipo MyClass
MyClass.method(ist)
```

Metodi di classe

- Funzioni definiti dentro il corpo di una classe applicabili alla classe stessa
- Il primo argomento, solitamente per convenzione ***cls***, e' il riferimento alla classe stessa

```
class MyClass(object):
        def cmethod(cls):
                print 'metodo di classe'
        cmethod = classmethod(cmethod)

MyClass.cmethod()
```

- Un metodo di classe si genera a partire da un metodo di istanza tramite ***classmethod*** che sovrascrive il riferimento ad un metodo di istanza con un riferimento ad un metodo di classe con lo stesso nome

Metodi di classe

- Il metodo di classe puo' essere creato con una sintassi alternative tramite i ***decorators***

```
class MyClass(object):
      @classmethod
      def cmethod(cls):
            print 'metodo di classe'

MyClass.cmethod()
```

Metodi di classe

- I metodi di classe possono essere chiamati sia dalla classe che da una sua istanza
- Il riferimento alla classe e' passato implicitamente

```
ist = MyClass()

MyClass.cmethod()

ist.cmethod()
```

Metodi statici

- Un metodo static si comporta come una funzione globale dentro il namespace della classe

```
class MyClass(object):
        def smethod():
                print 'metodo statico'
        smethod = staticmethod(smethod)

MyClass.smethod()
```

- Un metodo di classe si genera a partire da un metodo di istanza tramite ***staticmethod***

Metodi statici

- Il metodo statico dichiarato usando i ***decorators***

```
class MyClass(object):
      @statocmethod
      def smethod(cls):
            print 'metodo statico'

MyClass.smethod()
```

- I metodi statici non possono essere chiamati su un'istanza di classe

```
MyClass.smethod() # OK

MyClass().smethod() # sbagliato
```

Creazione/inizializzazione

- Oggetti python sono *creati* e *inizializzati* in due passi successivi separate, *new* e *init*.
- Per ridefinire le operazione da svolgere in queste due fasi si possono ridefinre le seguenti due funzioni:
  - Creazione, funzione membro statica ***__new__***
  - Inizializzazione, funzione membro di istanza ***__init__***

```
ist = myClass(…)

# Equivale

ist = MyClass.__new__(…)
ist.__init__(…)
```

__new__

- La funzione __new__ ***crea e restituisce una nuova istanza di classe*** non inizializzata
- E' richiamata automaticamente in fase di creazione
- Se non e' ridefinita, viene chiamata quella classe ***object*** (classi old-style non hanno questo metodo)

```
class MyClass(object):
      def __new__(cls):
            print 'new called'
            return object.__new__(cls)
```

__init__

- La funzione __init__ ***inizializza*** l'istanza appena creata
- Viene chiamata immediatamente dopo __*new*__
- Nel caso piu semplice e' usata per aggiungere attributi all'istanza di classe che viene inizializzata

```
class MyClass(object):
      def __init__(self):
            print 'init called'
            self.x = 10

ist = MyClass()
print ist.x # stampa 10
```

__init__

- __init__ come le altre funzioni puo' avere un numero arbitrario di parametric usati per l'inizializzazione

```
class MyClass(object):
      def __init__(self, x, y):
            print 'init called'
            self.x = x
            self.y = y
```

- La funzione deve restituire *None* altrimenti viene lanciata un eccezione

```
class MyClass(object):
      def __init__(self, x, y):
            print 'init called'
            return 1
TypeError: __init__() should return None
```

__del__

- Il metodo **__del__** viene chiamato quando una data istanza di un oggetto viene distrutta
- Il metodo e' l'equivalente di un distruttore nei linguaggi C++
- Per come il linguaggio python gestisce il garbage collector, questo metodo viene eseguito solo dopo che tutte le referenze ad un oggetto sono state rimosse

```
class MyClass(object):
      def __init__(self):
            print 'init called'
      def __del__(self):
            print 'del called'

ist = MyClass()
del ist
```

Attributi

- Elementi che appartengono ad un dato oggetto, accessibility tramite la classica notazione punto
- Gli attribute sono *elementi dinamici*, la **creazione/rimozione/modifica** degli attributi puo' avvenire in qualunque momento della vita dell'oggetto (classe o istanza)
- In python istanze della stessa classe possono avere **attributi differenti**
- Possono essere:
  - Di istanza
  - Di classe

Attributi di istanza

- Attributi di istanza sono attribute associati con una istanza specifica di un oggetto
- Questi possono essere creati al momento dell'inizializzazione nella __init__ o dinamicamete

```
class MyClass(object):
      def __init__(self):
            self.x = 10 # Aggiungi x come attributo

ist = MyClass()
ist.y = 20 # Aggiungi y
del ist.x # Rimuovi x
ist.y = 300 # Modifica y
```

Attributi di classe

- Anche le classi sono oggetti su cui e' possibile aggiungere attributi
- In questo caso gli attribute sono detti attribute di classe
- Il metodo piu' semplice e' quello di dichiarare all'interno della classe un attributo e assegnargli un valore

```
class MyClass(object):
      a = 20
print MyClass.a # stampa 20
MyClass.a = 30
print MyClass.a # stampa 30
```

Attributi di classe

- Le varie istanze della classe contengono anche gli attribute di classe
- Questi vengono automaticamente aggiunti come attributi di istanza e prendono come valore il valore che l'attributo di classe ha in quell momento

```
class MyClass(object):
       a = 0

print MyClass.a # stampa 0
ist = MyClass()
print ist.a # stampa 0
MyClass.a = 10
print MyClass.a # stampa 10
print ist.x # stampa 0
ist2 = MyClass()
print ist2.x # stampa 10
```

Attributi di classe

- Gli attribute di classe possono essere aggiunti e rimossi dinamicamente

```
class MyClass(object):
      a = 0

print MyClass.a # stampa 0
ist = MyClass()
MyClass.z = 10 # attributo z aggiunto
ist2 = MyClass()
ist2.z = 20
print MyClass.z # stampa 10
print ist2.z # stampa 20
print ist.z # AttributeError: 'MyClass' object has no attribute 'z'
del ist2.z
del MyClass.z
```

Attributi di classe

- Gli attribute di classe possono essere recuperati tramite:
  - La funzione built-in ***dir()***
  - Tramite l'attributo di classe ***__dict___***

```
class MyClass(object):
      a = 20

print MyClass.__dict__

print dir(MyClass)
```

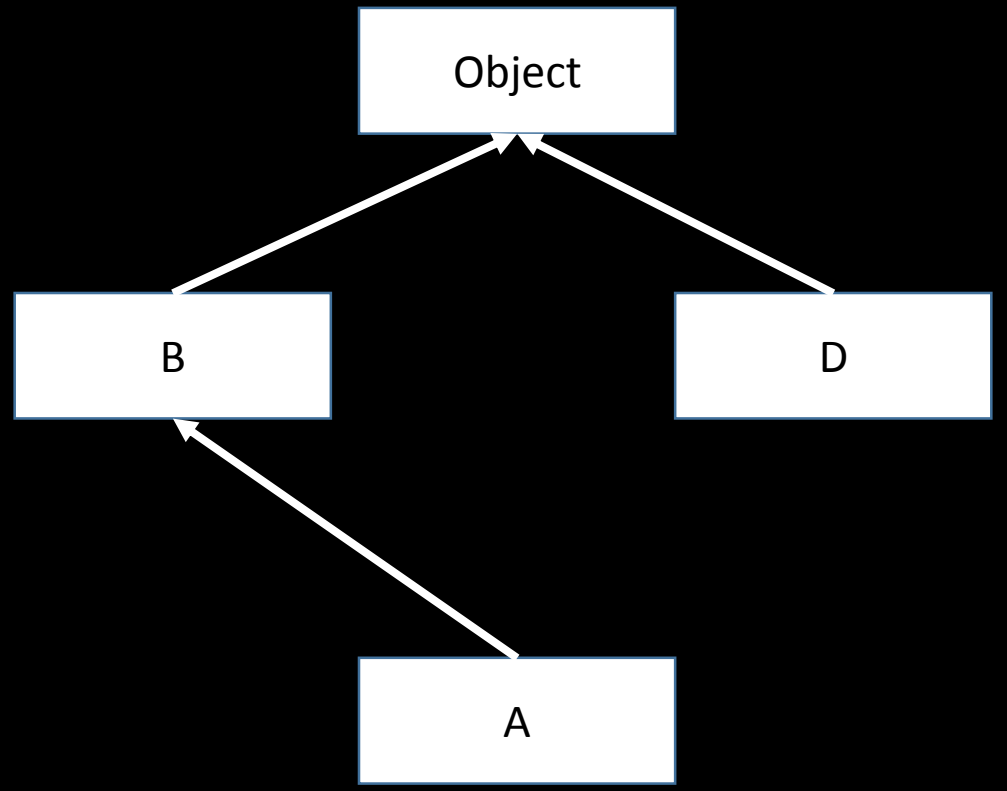
- __dict__ puo' essere usato anche per recuperare gli attribute di istanza

Ereditarieta’

- Python supporta il meccanismo dell’ereditarieta’ tra classi
- **Ereditarieta’ è la capacità di definire una nuova classe come versione modificata di una classe già esistente**
- Il vantaggio principale dell'ereditarietà è che si possono aggiungere nuovi metodi ad una classe senza dover modificare la definizione originale
- E’ chiamata "ereditarietà" perché la nuova classe "eredita" tutti i ***metodi*** della classe originale e gli ***attributi di classe***
- Estendendo questa metafora la classe originale è spesso definita "genitore" e la classe derivata "figlia" o "sottoclasse"
- Questa puo’ essere:
  - ***Singola***, la classe deriva da una sola classe
  - ***Multipla***, la classe deriva da una e piu’ classi

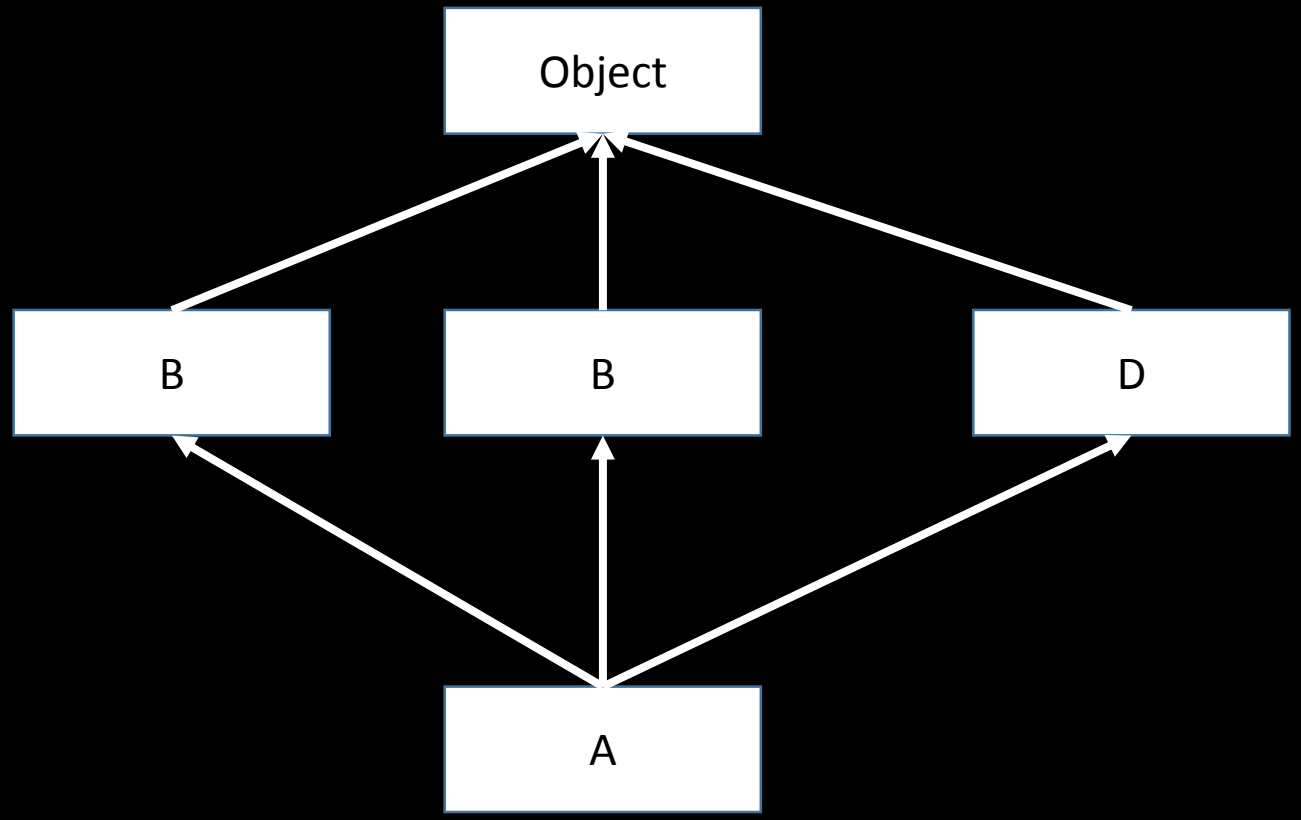
Ereditarieta’ singola

```
class C(object): pass
class B(C): pass
class D(C): pass
class A(B): pass
```



Ereditarieta' multipla

```
class C(object): pass
class B(object): pass
class D(object): pass
class A(B,C,D): pass
```



Ereditarieta’ multipla

- L’ereditarieta’ multipla puo’ portare a delle gerarchie complesse in cui non e’ semplice determinare la precedenza nella chiamata dei metodi
- I casi di ambiguita’ vengono gestiti seguendo delle regole complesse
- La lista delle classi, e la loro priorita’, e’ univocamente determinata dalla gerarchia contenuta nell’attributo read-only ***__mro__***

Ereditarieta': metodi

- I metodi della classe base sono ereditati dalla classe derivate
- Se nella derivate si ridefinisce un metodo (***override***), la chiamata alla classe base non e' automatica ma deve essere fatta in modo esplicito
- ***Lo stesso per il metodo __init__***

```
class A(object):
      def m(self): print 'A.m()'

      @staticmethod
      def sm(): print 'A.sm()'

      @classmethod
      def cm(cls): print 'A.cm()'
```

```
class B(A):
      pass

ist = B()

ist.m()
ist.cm()
ist.sm()
```

Ereditarieta': super

- Questa soluzione puo' generare problemi in caso di eredirarieta' multpla
- Alcuni metodi potrebbero essere chiamati piu' di una volta (grafi di ereditarieta' a diamante)
- ***Per evitare tale problema si usa la funzione super(aclass, obj) che ritorna un riferimento speciale al padre di tipo aclass dell'istanza obj che evita chiamate multiple allo stesso metodo***

```
class A(object):
      def __init__(self):
            print 'init A'
class B(A):
      def __init__(self):
            super(A,self).__init__()
class C(A):
      def __init__(self):
            super(A,self).__init__()
```

```
class D(B,C):
      def __init__(self):
            super(B, self).__init__()
            super(C, self).__init__()

ist = D()
```

Ereditarieta': attributi

- Attributi di classe sono ereditati
- Gli altri essendo aggiunti e cancellati in maniera dinamica non sempre sono ereditati
- Una classe puo' avere o meno gli attribute di istanza della classe base a seconda di come si comporta nell'inizializzazione
- Se gli attributi di istanza sono tutti aggiunti nella funzione __init__ e questa e' richiamata come nell'esempio precedente dale classi figlie, allora gli attribute di istanza sono aggiunti tutti di conseguenza

Ereditarieta': attributi

- In questo caso D ereditera' tutti gli attributi di istanza x, y e z

```
class A(object):
      def __init__(self):
            self.x = 10
            print 'init A'
class B(A):
      def __init__(self):
            super(A,self).__init__()
            self.y = 20
class C(B):
      def __init__(self):
            super(B,self).__init__()
            self.z = 30
```

```
class D(C):
      def __init__(self):
            super(C, self).__init__()

ist = D()
```

Ereditarieta': attributi

- In questo altro caso D ereditera' solo gli attributi di istanza y e z

```
class A(object):
      def __init__(self):
            self.x = 10
            print 'init A'
class B(A):
      def __init__(self):
            self.y = 20
class C(B):
      def __init__(self):
            super(B,self).__init__()
            self.z = 30
```

```
class D(C):
      def __init__(self):
            super(C, self).__init__()

ist = D()
```

Attributi: visibilita’

- Normalmente tutti gli attribute sono considerati *pubblici*
- Esiste la possibilita di rendere alcuni attribute *privati*
- Gli attributi che iniziano con doppio underscore __ sono trattate come fossero private dall’interprete
- Gli attributi speciali, iniziano e finiscono con doppio __
- Questi rappresentano funzioni e operatori particolari

Attributi speciali

- ***__class__*** contiene un riferimento alla classe a cui appartiene l'oggetto
- ***__name__*** contiene il nome della classe