Advanced

Simone Campagna

caratteristiche del linguaggio

- Tipizzazione forte
- Tipizzazione dinamica
- “Duck typing”

Attenzione: l'assegnamento non è un operatore come in C++!

```
>>> a = b
```

A sinistra c'è un nome simbolico, a destra un oggetto. Non c'è copia, solo binding!

tipizzazione forte/1

- Ciascun oggetto ha un tipo ben definito per tutta la sua esistenza; il tipo di un oggetto non può mai cambiare
- Le espressioni a cui l'oggetto può prendere parte dipendono dal tipo dell'oggetto.

tipizzazione forte/2

```
>>> 3 + 4
7
>>> "3" + "4"
'34'
>>> "3" + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> "3" * 4
'3333'
```

tipizzazione forte/3

- In linguaggi non tipizzati, espressioni come “3” + 4 possono avere senso; ad esempio, in Perl

```
print "3" + 4;
```

=> 7

- In altri linguaggi la stessa espressione potrebbe produrre la stringa “34”.
-

python vs perl

- Nel seguito del corso, userò sempre Perl come esempio di pessimo linguaggio di programmazione, al fine di mettere in luce gli aspetti positivi di Python...

tipizzazione forte/4

- La tipizzazione forte è un bene, evita che sia possibile compiere operazioni che non hanno senso!
- I linguaggi interpretati moderni sono tipizzati in maniera forte.
- Non ci sono particolari vantaggi in un linguaggio non tipizzato, solo molti svantaggi!

tipizzazione statica/1

- Alcuni linguaggi, come C, C++ o Fortran, adottano una tipizzazione statica.
- In questi linguaggi, il tipo è associato al nome simbolico
- Il tipo viene associato al nome simbolico all'atto della dichiarazione, e non può più cambiare nell'ambito dello *scope* di quel nome simbolico.
- In Fortran la dichiarazione può non essere obbligatoria, ma la tipizzazione è comunque statica (c'è un tipo di default).

tipizzazione statica/2

Un nome simbolico può essere associato a diversi oggetti nell'ambito del suo *scope*, purché questi oggetti abbiano tutti il tipo del nome simbolico!

```
INTEGER a, b

REAL c

a = 1

a = b

a = c !!! cast REAL -> INTEGER
```

tipizzazione dinamica/1

- Nella maggior parte dei linguaggi interpretati moderni, la tipizzazione è dinamica.
- Il tipo è associato all'oggetto, non al nome simbolico.
- Il nome simbolico perde gran parte della sua importanza: non deve essere dichiarato, e nell'arco della sua esistenza può puntare ad oggetti di tipo arbitrario!

tipizzazione dinamica/2

```
>>> a = 4
>>> print type(a)
<type 'int'>
>>> a = 4.5
>>> print type(a)
<type 'float'>
>>> a = "sono una stringa"
>>> print type(a)
<type 'str'>
>>> a = 1, 2
>>> print type(a)
<type 'tuple'>
>>>
```

type system/1

- Il type system stabilisce quali operazioni possono essere eseguite su un oggetto.
- Che cosa stabilisce se è possibile valutare l'espressione “x + y”?
- Che cosa stabilisce se è possibile eseguire x.some_method()?

type system/2

- Diversi linguaggi implementano diversi sistemi di tipizzazione:
- Structural typing (Ocaml)
- Nominal typing (C, C++, Fortran)
- “Duck” Typing (python, ruby)

structural typing

- Due oggetti si riferiscono allo stesso tipo se hanno la stessa struttura.
- Se un numero complesso è definito da una coppia di reali, e le coordinate di un punto sul piano sono definite da una coppia di reali, si tratta dello stesso tipo.

nominal typing

- Due oggetti si riferiscono allo stesso tipo se e solo se le loro dichiarazioni sono identiche, “nominano” lo stesso tipo.
- Il C (ed il C++) seguono il nominal typing, a parte per l'uso di typedef.
- Se “complex” e “point” hanno la stessa struttura, ma sono tipi dichiarati separatamente, non sono lo stesso tipo.

duck typing/1

“if it walks like a duck, and quacks like a duck, then it is a duck”

- In pratica, nel nominal typing e nello structural typing il controllo se “x” può comparire in una espressione avviene interamente in compilazione
- Nel caso del duck typing, il controllo è fatto in esecuzione: se l'espressione è possibile, allora viene eseguita.

duck typing/2

- Ad esempio:

```
>>> l = [1, 2, 3]
>>> d = { 0: 0, 'a': 2, 'b': 4 }
>>> i = 4
>>> print l[0] + 3
4
>>> print d[0] + 3
3
>>> print i[0] + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is unsubscriptable
```

duck typing/3

- I tre oggetti *l*, *d*, ed *i* hanno tipi diversi; *l* e *d* possono prendere il posto di *"**x**"* nell'espressione ***x[0] + 3*** in quanto i loro tipi implementano entrambi l'indicizzazione attraverso parentesi quadre; *i* invece no.

type system/3

- Il nominal typing consente di rilevare più errori in compilazione, e questo è un vantaggio. Consente anche maggiore velocità di esecuzione. Ma è scomodo e rallenta lo sviluppo. Inoltre rallenta notevolmente la fase di compilazione
- Il duck typing è molto più comodo e pratico, e consente uno sviluppo notevolmente più rapido, soprattutto è molto naturale implementare il polimorfismo.

la tipizzazione in python

- Strong
- Dynamic
- “Duck”
- Questo è quanto di meglio si possa desiderare in un linguaggio interpretato moderno!

“=”

- ATTENZIONE! L'apparentemente inoffensivo '=' assume un nuovo ruolo!

```
>>> a = [1, 2, 'c']
>>> b = a
>>> b.append(10)
>>> print b
[1, 2, 'c', 10]
>>> print a
[1, 2, 'c', 10] ### Argh! “a” è cambiato!!!
>>>
```

object reference/1

- Il fatto è che *'a=b'* non esegue una copia da *b* ad *a*, come in C, C++ o Fortran; il significato è **completamente** diverso:
- *a=b* significa che il nome simbolico *a* d'ora in avanti sarà utilizzato per accedere all'oggetto *b*.
- Quindi a e b sono ora nomi simbolici alternativi per lo stesso oggetto fisico, una lista. Sono *object reference.*

object reference/2

```
>>> b = 4
>>> a = [1, 2, 3]
>>> b = a # ora b si riferisce alla lista [1, 2, 3]
>>> print b
[1, 2, 3]
>>> b = 4 # ora b si riferisce all'intero 4
>>> print b
4
>>> a = “abc” # ora a si riferisce alla stringa “abc”
```

object reference/3

- Cosa ne è della lista *[1, 2, 3]* al termine dell'esempio precedente?
- Viene “creata” alla riga 2 (`a = [1, 2, 3]`), ed è riferita da *“a”*
- Alla riga 3 (`b = a`) è riferita anche da *“b”*
- Dopo la riga 5 (`b = 4`) è nuovamente riferita solo da *“a”*
- Dopo la riga 7 (`a = “abc”`) non è più riferita da nessuno!

object reference/4

- Ogni oggetto python contiene un attributo che rappresenta il numero di reference a quell'oggetto.
- Quando creiamo un nuovo reference, questo attributo viene incrementato di 1; quando un reference sparisce, viene decrementato di 1.
- Quando il numero di reference per un oggetto diventa 0, l'oggetto viene automaticamente distrutto.

garbage collection/1

- Dunque, alla riga 7 la lista *[1, 2, 3]* viene distrutta, perché ormai inutilizzabile
- Questo processo si chiama *garbage collection*.

garbage collection/2

```
>>> class My(object):
...   def __del__(self):
...     print "DEL CALLED!"
...
>>> m = My()
>>> del m
DEL CALLED!
>>> m = My()
>>>
>>> n=m
>>> del m # l'oggetto non può essere cancellato ora!
>>> n=10
DEL CALLED!
>>>
```

object identifier

- Ad ogni oggetto python inoltre è automaticamente attribuito un identificatore unico; la funzione *id* consente di conoscerlo:

```
>>> a = 4
>>> print id(a)
33534544
>>> b = a
>>> print id(b)
33534544
>>> a = [1, 2, 3]
>>> print id(a)
36130544
>>> b = a
>>> print id(b)
36130544
>>>
```

mutable/immutable objects/1

Alcuni tipi (int, float, long, str, ...) sono *immutable*: non ci sono metodi per modificarne le istanze.

```
>>> a = "ciao"
>>> id(a)
139857688217520
>>> a += " mondo"
>>> id(a)
139857688221600
>>>
```

mutable/immutable objects/2

Altri tipi (list, dict, set, ...) sono *mutable*, cioè ci sono metodi per cambiare lo stato delle singole istanze.

```
>>> l = [1, 2, 3]
>>> id(l)
139857688122000
>>> l += [4, 5]
>>> l
[1, 2, 3, 4, 5]
>>> id(l)
139857688122000
>>> l[3] = 10
>>> l
[1, 2, 3, 10, 5]
```

mutable/immutable objects/3

- In pratica, se un oggetto è *immutable*, l'oggetto cambia solo apparentemente: nella realtà viene creato un nuovo oggetto, ed il reference viene fatto puntare al nuovo oggetto:

```
>>> a = 4
>>> print id(a)
33534544
>>> a += 1 # è come a = a + 4; il precedente oggetto int “4”
           # è stato distrutto, ed un nuovo int “5” creato
>>> print id(a)
33534448
>>>
```

mutable/immutable objects/4

- Quando un oggetto mutable viene modificato, cambia il contenuto, lo stato dell'oggetto, ma non il suo identificatore:

```
# modifica della stessa
# istanza
>>> l = [0, 3, 4]
>>> print id(l)
36144696
>>> l += ['a', 'b']
>>> print id(l)
36144696
```

```
# creazione di una nuova
# istanza
>>> l = [0, 3, 4]
>>> print id(l)
140561359168792
>>> l = l + ['a', 'b']
>>> print id(l)
140561358639688
```

pitfall: mutable default arguments/1

Attenzione: il binding dei valori di default agli argomenti della funzione viene fatto una volta sola, al momento della creazione della funzione!

pitfall: mutable default arguments/2

```
>>> def add_to_list(element, initial_list=[]):
...     initial_list.append(element)
...     return initial_list
...
>>> add_to_list(10,[1,2,3])
[1,2,3,10]
>>> add_to_list(1)
[1]
>>> add_to_list(90)
[1, 90]
>>> add_to_list(“alfa”)
[1, 90, 'alfa']
```

closure/1

Il termine *closure* assume spesso significati diversi.

Tecnicamente una *closure* è una funzione con variabili libere il cui binding è conosciuto in anticipo. Tutte le funzioni python sono chiuse in questo senso!

Per closure in python si intende invece una funzione che può accedere ad un ambiente non più attivo.

closure/2

```
>>> functions = []
>>> for i in range(3):
...     functions.append(lambda x: x+i)
...
>>> for function in functions:
...     print function(10)
...
12
12
12
```

closure/3

```
>>> functions = []
>>> def make_fun(i):
...     return lambda x: x+i
>>> for i in range(3):
...     functions.append(make_fun(i))
...
>>> for function in functions:
...     print function(10)
...
10
11
12
```

decorators

Un decorator è una speciale funzione che “trasforma” generiche funzioni in altre funzioni, aggiungendo del codice all'inizio e/o alla fine.

Ad esempio, la temporizzazione può essere fatta con un decorator.

decorator example

```
>>> def traced(f):
...     def traced_f(*la,**kw):
...       print “BEGIN %s” % f.__name__
...       r = f(*la,**kw)
...       print “END %s” % f.__name__
...       return r
...     return traced_f
...
>>> s = traced(math.sqrt)
>>> @traced
... def fun(a, b, alfa=1000):
...   print a+b+alfa
...
>>> fun(1,2)
BEGIN fun
1003
END fun
```

pickle

Spesso capita di voler salvare interi oggetti (persistenza). Il modulo pickle fa questo.

```
>>> myobj = SomeClass(...)
>>> import pickle
>>> f = open(“myobj.dump”, ”wb”)
>>> pickle.dump(myobj,f)
...
>>> f = open(“myobj.dump”, “rb”)
>>> myobj = pickle.load(f)
```

classi

Due tipi di classi:

- Old-style:

```
class MyClass:
    ...
```

- New-style:

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

metodi/1

```
>>> class MyClass(object):
...     def set(self, a, b):
...         self.s = a+b
...     def get(self):
...         return self.s
...
>>> m = MyClass()
>>> m.set(10,12)
>>> print m.get()
22
```

metodi/2

Tutti i metodi hanno come primo attributo l'istanza della classe su cui si applicano:

```
>>> m = MyClass()
>>> m.set(10,12)
```

equivale a:

```
>>> MyClass.set(m,10,12)
```

C'è comunque una differenza fra le funzioni m.set e MyClass.set!

```
>>> m.set
<bound method MyClass.set of <__main__.MyClass object
at 0x7fee012667d0>>
>>> MyClass.set
<unbound method MyClass.set>
```

bound/unbound methods/1

I metodi non legati ad una istanza sono “unbound methods”: il primo argomento è libero e può essere associato ad un qualsiasi oggetto di quella classe.

Quando un metodo viene associato ad una istanza, esso diventa un “bound method”: una nuova funzione ottenuta dall'”unbound method” fissando il primo parametro; qualcosa del genere

```
m.set

lambda i,j : MyClass.set(m,i,j)
```

bound/unbound methods/2

Dunque, mentre MyClass.set è una funzione con tre argomenti liberi, m.set ha solo due argomenti liberi, in quanto il primo è fissato (m).

oggetti/1

Qualsiasi oggetto (istanza di una classe) è dotato di vari attributi; fra questi:

- **`__class__`**

  la classe di cui l'oggetto è una istanza

- **`__dict__`**

  il dizionario dei membri dell'istanza

Poiché anche le stesse classi sono istanze di altre classi, anche le classi hanno questi attributi!

oggetti/2

```
>>> print m.__class__
<class '__main__.MyClass'>
>>> print MyClass.__class__
<type 'type'>
>>> print m.__dict__
{}
>>> m.set(100,20)
>>> print m.__dict__
{'s': 120}
>>> print MyClass.__dict__
{'__module__': '__main__', 'set': <function set at 0x7fbdfc669488>,
'get': <function get at 0x7fbdfc669500>, '__dict__': <attribute
'__dict__' of 'MyClass' objects>, '__weakref__': <attribute
'__weakref__' of 'MyClass' objects>, '__doc__':None}
>>>
>>> dir(m)
```

come avviene la ricerca degli attributi

- Quando si accede ad un attributo su un oggetto (x.attr), per prima cosa si cerca di vedere se l'attributo è in x.__dict__
- Se non c'è, si cerca l'attributo è nel dizionario della classe: x.__class__.__dict__
- In realtà per le classi la questione è più complessa, perché l'ereditarietà impone l'eventuale ricerca dell'attributo nelle superclassi secondo l'MRO (Method Resolution Order)

membri di classe/1

Anche le classi possono avere attributi o metodi (sono simili agli attributi static in C++).

```
>>> class MyClass(object):
...     ALFA = 10
...     BETA = “ciao”
```

In questo caso ALFA e BETA sono attributi della classe, non delle istanze.

membri di classe/2

```
>>> class MyClass(object):
...     ALFA = 10
...     BETA = “ciao”
...
>>> m = MyClass()
>>> print m.ALFA
10
```

membri di classe/3

Attenzione, però!

```
>>> m.ALFA = 33
```

In questo caso non ho modificato l'attributo di classe, ma ho aggiunto un parametro ALFA all'istanza m!

Per cambiare l'attributo di classe ovviamente si fa così:

```
>>> m.__class__.ALFA = 3
```

o analogamente

```
>>> MyClass.ALFA = 3
```

membri di istanza e di classe

I membri di istanza si creano e gestiscono attraverso l'argomento “self” di un qualsiasi metodo.

Sia le classi che le istanze sono “aperte”, si possono inserire attributi o metodi in qualsiasi momento:

```
>>> m.xx = 10

>>> MyClass.GAMMA = 101
```

membri di istanza/1

Diversamente da quanto accade in C++, quindi, non è affatto detto che tutte le istanze di una stessa classe abbiano esattamente gli stessi attributi!

membri di istanza/2

```
>>> class MyClass(object):
...     pass
...
>>> m = MyClass()
>>> n = MyClass()
>>> print m.__dict__, n.__dict__
{} {}
>>> m.x = 0
>>> print m.__dict__, n.__dict__
{'x': 0} {}
```

costruttore

Il costruttore della classe è il metodo __init__:

```
>>> class ModInt(object):
...     def __init__(self,mod,val=0):
...         self.mod = mod
...         self.val = val
...
```

Esiste un solo costruttore!

distruttore

Il distruttore non è così importante come in C++, anche a causa dell'efficiente garbage collector. Non lo si definisce quasi mai...

```
>>> class ModInt(object):
... ....
...    def __del__(self):
...     print “banzai!”
...
```

altri metodi speciali

```
>>> class ModInt(object):
... ....
...   def __repr__(self):
...     return “ModInt(%s,%s)” % (self.mod, self.val)
...
...   def __str__(self):
...     return “%s mod %s” % (self.val, self.mod)
...
>>> m = ModInt(10,2)
>>> print m
2 mod 10
```

special method names

- `__new__, __init__, __del__`
- `__repr__, __str__`
- `__le__, __lt__, __ge__, __gt__, __ne__, __eq__`
- `__cmp__, __rcmp__`
- `__hash__`
- `__nonzero__`
- `__unicode__`
- `__call__`

special methods for attribute access

- __getattr__, __setattr__, __delattr__
- __getattribute__
- __slots__
-

special methods for class creation

- __metaclass__

special methods for containers

- __len__
- __getitem__, __setitem__, __delitem__
- __iter__
- __reversed__
- __contains__
- __getslice__, __setslice__, __delslice__

descriptors

- __get__, __set__, __del__

mathematical operators

- `__add__, __sub__, __mul__, __div__, __truediv__, __floordiv__, __mod__, __divmod__, __pow__, __lshift__, __rshift__, __and__, __xor__, __or__`
- `__iadd__, __isub__, ...`
- `__radd__, __rsub__, ...`
- `__neg__, __pos__, __abs__, __invert__`
- `__complex__, __float__, __int__, __long__`
- `__oct__, __hex__`
- `__index__`
- `__coerce__`

descriptors

I metodi __get__, __set__ e __del__ consentono di definire modalità di accesso.

Se una classe *owner* possiede nel proprio __dict__ un *descriptor*, ovvero una istanza di una classe che definisce almeno uno fra __get__, __set__ e __del__, allora la modalità di accesso a quell'attributo di owner avviene attraverso le funzioni del descriptor.

properties/1

In termini più semplici, si realizza una property, vale a dire un attributo con *binding behavior*: un attributo di cui possiamo stabilire le regole di accesso (get, set e del).

La funzione property crea un descriptor con metodi __get__, __set__ e __del__ associati a funzioni passate come argomento.

properties/2

```
>>> class ModInt(object):
...   def __init__(self, mod, val=0):
...     self.__mod, self.__val = mod,val
...
...   def __get_val(self):
...     return self.__val
...
...   def __set_val(self, val):
...     self.__val = val % self.__mod
...
...   val = property(__get_val,__set_val)
...
>>> m = ModInt(10,2)
>>> m.val = 99
```

metodi di istanza, statici, di classe

Abbiamo visto i metodi di istanza. Ci sono anche i metodi statici, che non hanno l'attributo *self*, e quelli di classe, che sono associati alla classe e non all'istanza.

metodi statici e di classe/1

```
>>> class My(object):
...   def fs():
...     print “fs”
...   fs = staticmethod(fs)
...
...   def fc(cls):
...     print “fc”,cls
...   fc = classmethod(fc)
...
>>> m = My()
>>> m.fs()
fs
>>> m.fc()
fc <class '__main__.My'>
```

decoratori per metodi

I decoratori *staticmethod* e *classmethod* servono per definire metodi statici e di classe.

Il decoratore *property* serve per definire una property caratterizzata dalla sola funzione get.

metodi statici e di classe/2

```
>>> class My(object):
...   @staticmethod
...   def fs():
...     print “fs”
...
...   @classmethod
...   def fc(clss):
...     print “fc”,clss.__name__
...
>>> m = My()
>>> m.fs()
fs
>>> m.fc()
fc <class '__main__.My'>
```

variabili private

Python ha un debole strumento per la restrizione dell'accesso a membri di classi. Se nella classe ALFA è definito un identificatore __xyz, il nome dell'identificatore è rimpiazzato con _ALFA_xyz. Comunque la variabile, con un nome diverso, resta accessibile:

```
>>> class ALFA(object):
...   def __init__(self):
...     self.__xyz = 10
...
>>> a.__xyz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'ALFA' object has no attribute '__xyz'
>>> a._ALFA__xyz
10
>>>
```

metaclass/1

```
>>> class Readonly(type):
...   def __new__(clss, classname, bases, dict):
...     def make_getmethod(attr_name):
...       def getmethod(self):
...         return self.__readonly__[attr_name]
...       return getmethod
...     for attr_name, attr_default in dict.get('__readonly__',
{}).iteritems():
...       dict[attr_name] = property(make_getmethod(attr_name))
...     return type.__new__(clss, classname, bases, dict)
...
>>> class My(object):
...   __metaclass__ = Readonly
...   __readonly__ = { 'alfa': 1, 'beta': 100 }
...   pass
...
```

metaclass/2

```
>>> m = My()
>>> m.x=20
>>> print m.x, m.alfa, m.beta
20, 1, 100
>>> m.alfa=33
Traceback (most recent call last):
  File "<stdin>", ...
    m.alfa=100
AttributeError: can't set attribute
```

inheritance

L'ereditarietà si effettua inserendo i nomi delle classi di base fra parentesi tonde di seguito al nome della classe che si sta definendo:

```
>>> class Car(Vehicle):
>>>   pass
>>> class D(B1, B2, B3):
>>>   pass
```

polimorfismo

Il polimorfismo è il default: tutti i metodi sono automaticamente virtuali. Anche le classi di base sono virtuali.

È una conseguenza naturale del duck typing, ed a rigore non richiede neppure l'ereditarietà!

super/1

Per le classi new-style (che ereditano da object) è possibile usare la funzione super(...) che fornisce un proxy object in grado di chiamare il metodo sulle classi di base:

```
>>> class D(B0, B1):
...     def __init__(self):
...         super(D,self).__init__()
...
```

super/2

L'effetto di super è di richamare il metodo in tutte le classi di base.

ATTENZIONE: ci sono dei vincoli:

- usare solo classi new style
- usare super sistematicamente (deve essere usato nelle classi di base!)
- i metodi devono avere la stessa interfaccia

gerarchie complesse

In python è molto semplice realizzare gerarchie di classi anche di notevole complessità.

Ma ricordate:

Simple is better than complex.

Complex is better than complicated.