

business

collaboration

people

# Il futuro di Python

Alex Martelli



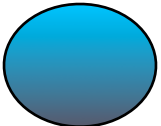
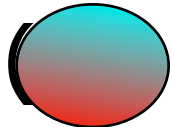
# Questo talk e il suo pubblico

---

- conoscete (almeno un poco) Python (magari non l'ultimissima versione...)
- siete programmatori abbastanza esperti (magari in altri linguaggi)
- volete qualche indicazione sul presente e il futuro di Python

# Il progresso di Python

Ver. data la + recente per (mesi):

1.5.2	1999/04	17	stabile
2.0	2000/09	07	abb. stabile
	2001/04	08	abb. stabile
2.2	2001/12	19	stabile
2.3	2003/07	(15?)	<b>molto</b> stabile
	2004/10?)	??	...?...



# Cinque anni di novità

---

- Unicode e codec
- list comprehension
- iteratori e generatori
- "nuove classi", metaclassi, descrittori
- ..."e chi + ne ha + ne metta"...
- ...per non dire delle **librerie**...!
  - standard ("batterie comprese")
  - terze parti

# La stabilità di Python

- rivoluzioni 1.5.2→2.0→2.1→2.2
- stabilità 2.2→2.3(→2.4 ?)
  - minuscoli (o piccoli) cambiamenti al linguaggio
  - in tutto, ~3 anni (o +) di esperienza stabile
  - implementazione via via migliore (velocità, solidità)
  - la libreria continua a crescere e migliorare
- ...risultato netto...:

adattissimo all'uso in produzione!

# List comprehension

```
L = [ x*x for x in LL if x>0 ]
```

come un insieme  $\{x*x \mid x \in L, x > 0\}$  -- è come...:

```
L = [ ]
```

```
for x in LL:
```

```
    if x>0:
```

```
        L.append(x*x)
```

vantaggi: 1 pensiero → 1 idioma compatto  
è un'*espressione*...

# Filosofia x le list comprehension

- sto pensando ad un ciclo (magari di azioni, effetti collaterali, ...)?
- o sto pensando a un insieme di elementi (magari in un certo specifico ordine)?
- se é un ciclo, scrivo un ciclo
  - normalmente 'for' -- vedi oltre
- se é un insieme → list comprehension
  - o, in 2.4, un'iterator comprehension

# Gli iteratori

Una classe è *iterabile* se ha un metodo `__iter__` che torna un oggetto **iteratore**:

```
class iterabile:  
    def __iter__(self):  
        return mio_iter(self)
```

Ogni istanza di iteratore tiene traccia dello stato di una iterazione, torna `self` da `__iter__`, ha un metodo `next`

# Struttura tipica di iteratore

```
class myiter:
    def __init__(self, ...):...
    def __iter__(self):return self
    def next(self):
        [ [ ...avanza un passo... ] ]
        if [ [ é finita ] ]:
            raise StopIteration
        return [ [ il prossimo valore ] ]
```

# Iteratori e istruzione for

```
for x in itrbl: corpo
```

é per definizione identico a:

```
_tmp = iter(itrbl)
```

```
while True:
```

```
    try: x = _tmp.next()
```

```
    except StopIteration: break
```

```
corpo
```



# Filosofia x iteratori

---

- il ciclo migliore di solito é un `for`
- anche entro una `comprehension`
- o implicito (ad es., `built-in sum`)
- lineare, mono-livello, "displayed"
- grazie agli iteratori, scrivo piú `for`

# Generatori

- funzione contenente la keyword `yield`
- chiamata, torna un nuovo iteratore `x`
- a ogni chiamata `x.next()`, riprende l'esecuzione della funzione sino al prossimo `yield` o `return`
- se `yield`, il risultato di `x.next()` è l'argomento della `yield`
- se `return`, solleva `StopIteration`



# Filosofia x generatori

---

- posso riassumere quasi qualsiasi struttura di controllo in un `for`
- anche recursione e cicli "complicati"
- "generatori" → "riuso di strutture logiche di controllo del codice"

## 2.4: iterator comprehension

- come list comprehension, ma con `()` e non `[]`
- risparmiano memoria, generando 1 solo elemento alla volta
- adatte per iterazioni e accumulatori come `sum` (e il nuovo modulo `stats`)
- spingono sempre + avanti l'idea....:
- "tutto é un `for`" !



# Modulo itertools

---

- mattoni ad alto livello per iteratori
- ispirati da Haskell / SML
- rapidi, usano poca memoria
- trasformatori di iterabili:
  - chain, cycle, dropwhile, ifilter, imap, islice, izip, starmap, takewhile
- costruiscono nuovi iteratori:
  - count, repeat

# novità di itertools in 2.4

---

- groupby (simile al GROUP BY di SQL)
- tee ("fotografa"/replica un iteratore)
- ottimizzazioni importanti
- ...spingiamo sempre + avanti l'idea...:
- `for über alles!-)`

# Il nuovo object-model

- problemi del classico object model...:
  - 4 separati "casi" di oggetti
    - tipi, classi, istanze di classi, istanze di tipi
    - non semplice mischiarli/scambiarli
  - trasformazione funzione → metodo "magica"
  - metaclassi: terribilmente complesse
  - risoluzione nomi in MI troppo semplicistica
- caratteristiche che devono restare, x compatibilitá -- *classi classiche*
- in parallelo, emerge un nuovo OM

# L'istruzione class

```
class X [basi] : [corpo]
```

- esegue il corpo e fa il dict d,
- trova la metaclassa M e la chiama:

```
X = M('X', basi, d)
```

- connette il risultato al nome  
normalmente: `type(X) is M`  
→ le classi sono istanze di metaclassi

# Come si trova la metaclassa?

- `__metaclass__` nel corpo della classe
- ereditata dalla base + a sinistra
- `__metaclass__` nei globali
- ultima sponda: `types.ClassType`
  - NB: l'ultima sponda dá classi *classiche*
- i tipi built-in hanno metaclassa `type`
- tipo built-in `object`: null'altro...!

# Rendere le classi nuovo-stile

- il modo piú tipico:

```
class X(object): ...
```

  - (o anche: `class X(list), &c)`
- parecchie nuove caratteristiche facoltative rispetto alle classi classiche
- *una* cosa richiede attenzione...:
- la ricerca implicita di metodi speciali é sulla *classe*, **non** sulla *istanza*

# I descrittori

- una classe o tipo contiene **descrittori**
- un descrittore ha `__get__` (se anche `__set__` é detto *data descriptor*)
  - "data descriptor" → prioritá sul dict d'istanza
- `x.y` → `type(x).y.__get__(x)`
- `x.y=z` → `type(x).y.__set__(x,z)`
- puó avere `__delete__` & `__doc__`

# Le propriété

```
class rect(object):
    def __init__(self,x,y):
        self.x=x; self.y=y
    def getArea(self):
        return self.x*self.y
    def setAtea(self, area):
        self.y = float(area)/self.x
    area=property(getArea,setArea)
```

# perché le proprietà **contano**

- senza proprietà potrei scrivere molti accessori `getThis`, `setThat`...
  - "hai visto mai" che un qualche accesso ad attributo dovesse richiedere esecuzione di codice in futuro...
- il 90+% degli accessori sono ripetitivi
  - il codice ripetitivo é una **pessima** cosa
- colle proprietà, usa sempre il semplice e naturale `x.this`, `x.that=23`
  - puoi cambiare attributo → proprietà se mai occorre

# Le funzioni sono descrittori

```
>>> def f(x, y): return x+y
>>> plus23 = f.__get__(23)
>>> print plus23(100)
123
>>>
```

- la trasformazione funzione → metodo non usa piú "magia", ma segue dalle regole generali

# staticmethod, classmethod

```
class nu(object):
    def f(): return 'hey'
    f = staticmethod(f)
    def g(cls): return 'ho%s' % cls
    g = classmethod(g)
class sb(nu): pass
print nu.f(), nu.g(), nu().f()
print sb.f(), sb.g(), sb().g()
```

# Esempio di classmethod

```
class dict:
    def _fks(cls, seq, val=None):
        x = cls()
        for k in seq: x[k]=val
        return x
    fromkeys = classmethod(_fks)
```

- é parte del builtin dict nella 2.3
- ctor alternativo → tipico classmethod

## 2.4: sintassi a modificatori

- `def f(cls) [classmethod]: corpo`

- `é come...:`

```
def f(cls): corpo
```

```
f = classmethod(f)
```

- può usare 1 o + modificatori
- valida sia x funzioni sia x metodi
- filosofia: la sintassi conta -- semplice é meglio di complicato

# Metodo `__new__`

- `type.__call__(cls, *a, **k)` usa una semplice DP "template method":

```
nu = cls.__new__(cls, *a, **k)
if isinstance(nu, cls):
    cls.__init__(nu, *a, **k)
return nu
```

- facilita caching, singleton, ...

# Risoluzione nomi: classica

```
class sic:
    def f(): return 'sic.f'
    def g(): return 'sic.g'
class d1(sic):
    def f(): return 'd1.f'
class d2(sic):
    def g(): return 'd2.g'
class leaf(d1, d2): pass
```

# Risoluzione nomi: moderna

```
class nu(object):
    def f(): return 'nu.f'
    def g(): return 'nu.g'
class d1(nu):
    def f(): return 'd1.f'
class d2(nu):
    def g(): return 'd2.g'
class leaf(d1, d2): pass
```

# Metaclassi custom

- Necessità rara, ma...:

```
class mymeta(type):  
    def __new__(c,n,b,d):  
        d.update(blah_blah)  
        return type.__new__(c,n,b,d)  
class funky: __metaclass__=mymeta
```

- eredita `type`, override `__new__` → le metaclassi custom sono spesso così

# ...chi + ne ha + ne metta...

- GC migliorata, weakref
- `import/as`, 'hook' x `import`, `zipimport`
- `%r`, `zip`, `sum`, `int/long w/base`, `bool`
- le funzioni hanno attributi
- dict: `setdefault`, `pop`, `**k`, iterazione
- slice arricchita, `list.index start/stop`
- stringhe potenziate: `in`, `strip`
- file potenziati: `'U'`, iterazione



# Gli insiemi

---

- in 2.3: modulo sets
- in 2.4: built-in set e frozenset
- é un salto importante:
  - si affiancano appieno a list e dict
  - espressione diretta dell'idea di progetto
  - meno cicli, e cicli piú semplici

## 2.4: built-in nuovi/migliorati

- `reversed(sequenza)` → iteratore sulla sequenza in ordine inverso
- `sorted(iterabile)` → lista sortata
  - `list.sort` e `sorted` hanno facoltative `key` e `reverse`
  - ad esempio, `sort case-insensitive`:
  - `L.sort(key=str.lower)`
  - `operator.attrgetter` e `itemgetter` x uso come `key`
- filosoficamente allineati all'idea: cicli piú semplici e ottimizzati

## 2.4: ottimizzazioni

- bisect e heapq circa 5-6 volte + veloci
- circa 30% di accelerazione x slice liste e tuple, vari cicli su dict, list comprehension
- accelerazione sino a 3 volte di list(), &c, su argomenti che forniscono `__len__`
- di 2 volte su `__getitem__` di list e dict
- al netto su pystone, circa 15%-20%





# Filosofia dell'ottimizzazione

---

- non farlo
- non farlo ancora
- Python + potente e ottimizzato significa:
  - programma con la massima semplicità
  - sempre meno necessario "fare i furbi"
- make it work, make it right, make it fast
  - con Python 2.3, e ancor + con 2.4...
  - ...sempre + spesso puoi evitare la terza fase!

# Implementazioni alternative

## ■ Esistenti:

- Jython (per JVM) stabile
- IronPython (per .NET) sperimentale/pre-alpha
- Stackless (microthreading) molto stabile
- Psyco (genera codice macchina JIT) abb. stabile
  - per ora: solo per CPU di tipo x86

## ■ Prospettiva di convergenza futura:

- **pypy** (reimplementazione in Python)
- facilita: alterazioni semantiche (stackless &c)
- facilita: generazione di codice (pyrex/psyco/...)

business

collaboration

people

AMK's "What's New in Python":

<http://www.amk.ca/python/>

GvR sul nuovo object-model e i descrittori:

<http://www.python.org/2.2/descriintro.html>

AM, DA, ML, MLH, and molti altri, su tutto questo....:

Python in a Nutshell, Learning Python,  
The Python Cookbook, Practical Python

AM, JH su iteratori, generatori, metaclassi, ....:

<http://www.strakt.com>