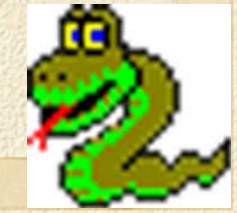
Python per Programmatori

Alex Martelli (aleaxit@yahoo.com)

Copyright©2005 Alex Martelli -- 27/01/2005



Python é (tante belle parole...)

- linguaggio altissimo livello (VHLL)
- sintassi pulita e spoglia
- semantica semplice e regolare
- estrema modularitá
- alta produttivitá
- open-source, cross-platform
- object-oriented
- ...

Python é... (come Java...)

- compilatore->bytecode + VM/interpreter
  - ma: compilazione implicita ("auto-make")
  - VM dedicata (o, con Jython, JVM)
  - compilatori JIT/SC (psyco, starkiller, pypy)
- tutto eredita da `object`
- semantica uniforme "object-reference":
  - assegnazione, passaggio argomenti, ...
  - anche per i numeri (immutabili, come le stringhe)
    -- piú uniforme
- vasta, potente libreria standard
- introspezione, serializzazione, thread, ...

Python é... (come C++...)

- multi-paradigma
  - object-oriented, procedurale, generico, FP
- ereditá multipla (strutturale, mix-in)
- overloading degli operatori
- polimorfismo basato su signature
  - come se “tutto fosse un template”... ma con sintassi semplice e pulita
- tantissime possibilitá di scelta per:
  - GUI, framework x Web server, accesso a database, COM/Corba/...

Python é... (come C...)

"lo spirito del C"... all'87% (+ di Java/C++...)
dal "Rationale" dello Standard C ISO:

1. fidati del programmatore
2. non impedire al programmatore di fare quello che occorre
3. mantieni il linguaggio piccolo e semplice
4. offri un solo modo di eseguire ciascuna operazione
5. (preferisci l'efficienza alla portabilitá)
   - ❖ non al 100% in Python, ma, ad es: float == quelli offerti dalla macchina

Python é... (molto diverso...)

- tipizzazione forte ma dinamica
  - gli **oggetti** hanno tipi (forti), i **nomi** no
  - niente dichiarazioni: solo istruzioni
- sintassi pulita, minima "ornamentazione"
  - blocchi senza { } -- solo indentazione
  - if e while senza ( )
- un po' tutto é un **oggetto di prima classe**
  - anche: classi, funzioni, metodi, moduli, package, ...
- focus ad alto/altissimo livello
  - metaclassi, generatori, descrittori, ...

Versioni e release di Python

- Python **classico**: oggi 2.4 (2.5 "in cottura")
  - implementato in C ISO (livello 1990)
- **Jython**: oggi 2.2 (2.3/2.4 "quasi pronto")
  - implementato in "Java puro al 100%"
  - deployment come Java, su di una JVM
  - puoi usare/estendere/implementare trasparentemente arbitrarie classi e interfacce Java / compilare a Java
- Altri: a livello sperimentale o di ricerca
  - Stackless, IronPython (.NET),Vyper (in O'CAML), **pypy** (progetto ricerca, finanziamento EU), ...

Risorse Python in rete

- http://www.zonapython.it
- http://www.python.it
  - il centro della comunitá italiana di Python
- news:it.comp.lang.python
  - domande, discussioni, richieste Python in italiano
- http://www.python.org
- news:comp.lang.python
- http://www.jython.org
- http://www.google.com
  - no, **davvero!!!**

Fondamenti di Python

- interprete interattivo (testo, IDLE, ...)
  - per provare cose, o come “calcolatrice”
  - prompt >>>, mostra valori di espressioni
- file di programma (afile.py, afile.pyc, ...)
  - per la maggior parte degli usi
  - la compilazione é automatica (all’**import**)
- assegnazione (la forma piú semplice):

  `nome = <qualsiasi espressione>`
  - crea il nome se occorre, lo lega al valore
  - i nomi non vengono dichiarati, e, di per se, non hanno alcun tipo

Assegnazioni e print

```
myvar = 'hello'                    # creo nome
myvar = 23                         # rilego nome
domanda = risposta = 42
myvar, risposta = risposta, myvar
print myvar, risposta, domanda
42, 23, 42

if myvar<20: myvar = 10            # non esegue
if myvar>40: myvar = 50            # esegue
print myvar
50
```

Condizionali

```
if domanda>30:              # 'if' é la “guardia”
   domanda = 20             #   di un blocco che
   x = 33                   #   é indentato a dx
else:                       # 'else' facoltativo
   x = 99                   #   indentazione

if x<30: myvar = 10         # non soddisfatto
elif x<40: myvar = 20       # soddisfatto
elif x<40: myvar = 40       # neppure testato
else: myvar = 80            # nemmeno questo

print x, domanda, myvar
33 20 20
```

Paragoni, test, veritá

```
eguaglianza, identitá:      ==  !=  is  is not
ordine:                     <  >  <=  >=
contenimento:               in  not in
paragoni e test “a catena”: 5<x<9  a==b==c

sono falsi: tutti i numeri == 0, “”, None,
          tutti i contenitori vuoti
sono veri:   tutti gli altri valori
il tipo Booleano (sottotipo di int):
    False==0,  True==1
    bool(x)   True o False (qualsiasi x)
not x == not bool(x)       (qualsiasi x)
```

and/or ("corto-circuito" logico)

and e **or** "corto-circuitano", e tornano come risultato *uno dei propri operandi*:

```
x = y and z  é come: if y: x=z
                     else: x=y
x = y or  z  é come: if y: x=y
                     else: x=z
print 0 and 0j, 0j and 0, 0 or 0j, 0j or 0
0 0j 0j 0
```

numeri

int (normalmente 32 bit) e **long** (illimitato):

```
print 2**100
```

1267650600228229401496703205376

float (normalmente 64 bit, IEEE):

```
print 2**100.0
```

1.26765060023e+030

complex (fatto di due **float**, **.real** e **.imag**):

```
print 2**100.0j
```

(0.980130165912+0.19835538276j)

aritmetica

addiz, sottraz, moltip, potenza: + - * **
divisione (vera, tronca, resto): / // %
bit-per-bit e shift: ~ & | ^ << >>
built-in: **abs divmod max min pow round sum**
conversioni di tipo: **complex float int long**

vedere anche i moduli: **math**, **cmath** e **operator**

```
import math, cmath
print math.sin(0.1)
0.0998334166468
print cmath.sin(0.3+0.2j)
(0.301450338429+0.192343629802j)
```

cicli

```
while myvar>10: myvar -= 7
print myvar
3
for i in 0, 1, 2, 3: print i**2
0 1 4 9
for i in range(4): print i**2 # lim.sup escl.
0 1 4 9
while e for di solito controllano blocchi
    indentati (standard: 4 a destra)
    possono contenere break e/o continue
possono avere un else (=="termine naturale")
```

i file (es: copia di un file)

```
fin = file('in', 'r')       # o solo file('in')
fou = file('ou', 'w')       # 'a', 'r+', 'wb'..

fou.write(fin.read())                    # o...
data = fin.read(); fou.write(data)     # o...
fou.writelines(fin.readlines())        # o...
for line in fin: fou.write(line)

fin.close()      # consigliabile, ma non
fou.close()      # obbligatorio (in CPython)
```

le stringhe (es: listato di un file)

```
for n, r in enumerate(fin):
    fou.write('Riga %s: %s' % (n+1, r))
```

o 1 singola istruzione (“list comprehension”):

```
fou.writelines([ 'Riga %s: %s' % (n+1, r)
                 for n, r in enumerate(fin) ])
```

o (Python 2.4) “generator expression”:

```
fou.writelines('Riga %s: %s' % (n+1, r)
               for n, r in enumerate(fin))
```

le stringhe sono sequenze

```
for c in 'ciao': print c,
```
c i a o

```
print len('cip'), 'i' in 'cip', 'x' in 'cip'
```
3 True False

```
print 'Amore'[2], 'Amore'[1:4], 'Amore'[::-1]
```
o mor eromA

```
print 'ci'+'ao', 'cip'*3, 4*'pic'
```
ciao cipcipcip picpicpicpic

le liste (vettori eterogenei)

```
x = [1, 2, 'beep', 94]

x[1] = 'plik'          # le liste sono mutabili
print x
[1, 'plik', 'beep', 94]

x[1:2] = [6,3,9]       # slice assegnabili
print x
[1, 6, 3, 9, 'beep', 94]

print [it*it for it in x[:4]]
[1, 36, 9, 81]
```

le liste sono sequenze

```
print x
[1, 6, 3, 9, 'beep', 94]

print len(x), 6 in x, 99 in x
6 True False

print x[2], x[1:5], x[1::2]
3 [6, 3, 9, 'beep'] [1, 9, 94]

print [1]+[2], [3, 4] * 3
[1, 2] [3, 4, 3, 4, 3, 4]
```

sequenze: indici e slice

```
x = 'ciaomondo'
print x[1], x[-3]
i n

print x[:2], x[2:], x[:-3], x[-3:]
ci aomondo ciaomo ndo

print x[2:6], x[2:-3], x[5:99]
aomo aomo ondo

print x[::2], x[-3:4:-1]
camno nom
```

sequenze: packing e unpacking

```
x = 2, 3, 6, 9      # tupla (immutabile)
print x
(2, 3, 6, 9)

a, b, c, d = x      # unpacking
print c, d, b, a
6 9 3 2

ROSSO, GIALLO, VERDE = range(3) # come enum

a, b, c, d = 'ciao' # unpacking
print c, d, b, a
a o i c
```

metodi delle stringhe

```
x = 'ciao mondo'
print x.upper(), x.title(), x.isupper()
CIAO MONDO Ciao Mondo False
print x.find('o'), x.count('o'), x.find('z')
3 3 -1
print x.replace('o', 'e')
ciae mende
print ', '.join(x.split())
ciao, mondo
print x.join('bah')
bciao mondoaciao mondoh
```

metodi delle liste

```
x = list('ciao')
print x
['c', 'i', 'a', 'o']
print x.sort()
None
print x
['a', 'c', 'i', 'o']
print ''.join(x)
acio
x.append(23); print x
['a', 'c', 'i', 'o', 23]
```

list comprehensions

```
[ <expr> for v in seq ]
[ <expr> for v in seq if <cond> ]

def divisori(x):
  """ la lista dei divisori dell'intero x """
  return [ n for n in range(2,x) if x%n==0 ]
def primo(x):
  """ x é primo? """
  return not divisori(x)
# quadrati dei primi compresi fra 3 e 33
print [x*x for x in range(3,33) if primo(x)]
[9, 25, 49, 121, 169, 289, 361, 529, 861, 961]
```

semantica di riferimento

```
x = [a', 'b', 'c']
y = x
x[1] = 'zz'
print x, y
['a', 'zz', 'c'] ['a', 'zz', 'c']

# se vuoi una copia, chiedila esplicitamente:
y = list(x)      # o x[:], x*1, copy.copy(x),
x[2] = 9999
print x, y
['a', 'zz', 9999] ['a', 'zz', 'c']
```

i dict sono *mappe*

```
x = {1:2, 'beep':94}
x[1] = 'plik'          # i dict sono mutabili
print x
{1:'plik', 'beep':94}
x['z'] = [6, 3, 9]  # aggiunta di un elemento
print x                # NB: ordine arbitrario
{1:'plik', 'z':[6, 3, 9], 'beep':94}

# costruire un dict da una seq. di coppie:
print dict([ (i, i*i) for i in range(6) ])
{0:0, 1:1, 5:25, 2:4, 3:9, 4:16}
```

le chiavi dei dict

Sempre *hashabili* (di solito, immutabili):

```
x = {}
x[[1,2]] = 'una lista non puo essere chiave'
```

TypeErrpr: list objects are unhashable

```
x[{1:2}] = 'un dict non puo essere chiave'
```

TypeErrpr: dict objects are unhashable

```
x[1,2] = 'una tupla va bene' # tuple hashabili
x[0j] = 'un complex va bene' # idem numeri,MA:
print x[0.0], x[0]  # 0==0.0==0.j, quindi...:
```

un complex va bene un complex va bene

i dict *non* sono sequenze, ma...

```
print x
{1:'plik', 'z':[6. 3, 9], 'beep':94}
for k in x: print k,
1 z beep
print x.keys(), x.values()
[1, 'z', 'beep'] ['plik', [6, 3, 9], 94]
print x.items()
[(1,'plik'), ('z',[6,3,9]), ('heep',94)]
# identico a: zip(x.keys(), x.values())
print len(x), 'z' in x, 99 in x
3 True False
```

metodi dei dict

```
print x.get(1), x.get(23), x.get(45, 'bu')
plik None bu
print x.setdefault(1, 'bah')
plik
print x.setdefault(9, 'wo')
wo
print x
{1:'plik',9:'wo','z':[6,3,9],'beep':94}
```

esempio: l'indice di un testo

```
# costruisco mappa parole->numeri di riga
idx = {}
for n, riga in enumerate(file('xx.txt')):
    for parola in riga.split():
        idx.setdefault(parola, []).append(n)
# mostro l'indice in ordine alfabetico
parole = idx.keys(); parole.sort()
for parola in parole:
    print "%s: " % parola,
    for n in idx[parola]: print n,
    print
```

l'equivalente in C++ standard

```
#include <string>
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
typedef std::vector<int> svi;

int main() {
  std::map<std::string, svi> idx;
  std::string line;
  int n = 0;
  while(getline(std::cin, line)) {
    std::istringstream sline(line);
    std::string word;
    while(sline>>word)
      idx[word].push_back(n);
    n += 1;
  }
  for(std::map<std::string, svi>::iterator i = idx.begin(); i != idx.end(); ++i) {
    std::cout<< i->first << ": ";
    for(svi::iterator j = i->second.begin(); j != i->second.end(); ++j)
      std::cout<< ' ' << *j;
    std::cout<< "\n";
  }
  return 0;
}
```

sulla KJB, 4.4MB di testo: C++ 8.5/17.40 secondi (ottimizzato, 7.38/15.01)

Python 5.4/11.22 (ottimizzato, 3.85/8.09)

le funzioni

```
def sumquadrati(x, y): return x*x+y*y
print sumquadrati(1, 2)
5
def sq1(x, y=1): return sumquadrati(x, y)
print sq1(1, 2), sq(3)
5 10
def sq(*args):      # numero variabile di arg.
    totale = 0
    for a in args: totale += a*a
    return totale
```

le funzioni con *chiusura* lessicale

```
def faiSommatore(addendo):
    def sommatore(augendo):
        return augendo + addendo
    return sommatore

piu23 = faiSommatore(23)
piu42 = faiSommatore(42)

print piu23(100),piu42(100),piu23(piu42(100))
```

123 142 165

le classi

```
class sico:
    cla = []                # attributo di classe
    def __init__(self):     # costruttore
        self.ins = {}       # attrib. d'istanza
    def meth1(self, x):
        self.cla.append(x)
    def meth2(self, y, z):
        self.ins[y] = z
# chiamando la classe si creano le istanze:
es1 = sico()
es2 = sico()
```

classi e istanze

```
print es1.cla, es2.cla, es1.ins, es2.ins
[] [] {} {}

es1.meth1(1); es1.meth2(2, 3)
es2.meth1(4); es2.meth2(5, 6)

print es1.cla, es2.cla, es1.ins, es2.ins
[1, 4] [1, 4] {2: 3} {5: 6}

print es1.cla is es2.cla, es1.ins is es2.ins
True False
```

sottoclassi

```
class sub(sico):
    def meth2(self, x, y=1):   # override
        sico.meth2(self, x, y) # call a super

class repetita(list):
    def append(self, x):
        for i in 1, 2:
            list.append(self, x)

class override_di_un_dato(sub):
    cla = repetita()
```

classi “nuovo stile”

```
class ns(object):
    def ciao(): return 'salve'
    ciao = staticmethod(ciao)
    def hi(cls): return 'hi,%s'%cls.__name__
    hi = classmethod(hi)
class sn(ns): pass
print ns.ciao(), sn.ciao(), ns.hi(), sn.hi()
salve salve hi,ns hi,sn
x = ns(); y = sn()
print x.ciao(), y.ciao(), x.hi(), y.hi()
salve salve hi,ns hi,sn
```

Python 2.4: i decoratori

```
class ns(object):
    @staticmethod
    def ciao(): return 'salve'
    @classmethod
    def hi(cls): return 'hi,%s'%cls.__name__
```

In generale, in Python 2.4:

```
@qualcosa
def f(...
    ...
```

é come (costrutto valido in 2.4, 2.3, 2.2, ...):

```
def f(...
    ...
f = qualcosa(f)
```

proprietá

```
class fapari(object):
    def __init__(self, num): self.num = num
    def getNum(self): return self.x * 2
    def setNum(self, num): self.x = num // 2
    num = property(getNum, setNum)

x = fapari(23); print x.num
22
x.num = 27.12; print x.num
26.0
```

Perché le proprietá **contano**

permettono di esporre attributi d'istanza:

```
x.foo = y.bar + z.baz
```

senza perdere **nessun** beneficio dell'incapsulamento, poichè, se e solo se occorre:

```
def setFoo(self, afoo): ...
foo = property(getFoo, setFoo)
```

e quindi risparmiano di esporre al codice cliente goffe sintassi alla `getThis`, `setThat`; soprattutto, evitano la jattura **boilerplate**:

```
def getPurancoQuesto(self):
    return self._puranco_questo
```

overload di operatori

```
class bugiardo(object):
    def __add__(self, altro): return 23
    def __mul__(self, altro): return 42
x = bugiardo()
print x+5, x+x, x+99, x*12, x*None, x*x
```
23 23 23 42 42 42

Overload possibile per: aritmetica, indici (e slice), accessi ad attributi, lunghezza, veritá, creazione, inizializz., copia...
ma **non** per “assegnazione ad un nome” (non c'è assegn. “A oggetti”, ma DI *oggetti* **a** *NOMI*)

eccezioni

A fronte di errori, Python solleva eccezioni; es:

```
x = [1, 2, 3]; x[3] = 99
Traceback (most recent call last):
   ...
IndexError: list assignment out of range
```

Posso definire nuove classi di eccezioni:

```
class ErroreStrano(Exception): pass
```

Posso sollevare eccezioni esplicitamente:

```
raise ErroreStrano, 223961
```

gestire le eccezioni

```
try:
    x[n] = unvalore
except IndexError:
    x.extend((n-len(x))*[None])
    x.append(unvalore)
else:
    print “tutto OK senza fatica”

f = file('uncertofile')
try: elabora(f)
finally: f.close()
```

gli iteratori

Un iteratore incapsula la logica di un ciclo, e permette di usare un semplice `for` al posto di un ciclo complicato

Creato chiamando (`for` lo fa implicitamente) la built-in `iter` su di un oggetto iterabile

Un iteratore espone un metodo `next` che torna "il prossimo elemento" a ogni chiamata

Quando non restano + elementi, `next` solleva `StopIteration` (significa "fine iterazione")

un iteratore non-terminante

```
class fiboniter(object):
    def __init__(self): self.i=self.j=1
    def __iter__(self): return self
    def next(self):
        r, self.i = self.i, self.j
        self.j += r
        return r
for conigli in fiboniter():
    if conigli > 100: break
    print conigli,
```

1 1 2 3 5 8 13 21 34 55 89

un iteratore terminante

```
class fiboniter_lim(object):
    def __init__(self, max):
        self.i=self.j=1
        self.max = max
    def __iter__(self): return self
    def next(self):
        r, self.i = self.i, self.j
        self.j += r
        if r>self.max: raise StopIteration
        return r
for conigli in fiboniter_lim(100):
    print conigli,
```

1 1 2 3 5 8 13 21 34 55 89

i generatori producono iteratori

```
def fiboniter_gen(max=None):
    r, i, j = 0, 1, 1
    while max is None or r <= max:
        if r: yield r
        r, i, j = i, j, i+j

for conigli in fiboniter_gen(100):
    print conigli,
```

1 1 2 3 5 8 13 21 34 55 89

un utile generatore: enumerate

```
# é built-in, ma, se non lo fosse...:
def enumerate(iterabile):
    n = 0
    for item in iterabile:
        yield n, item
        n += 1

print list(enumerate('ciao'))
[(0, 'c'), (1, 'i'), (2, 'a'), (3, 'o')]
```

2.4: generator expressions

```
# piú comunemente dette genexp
X = 'ciao'
x = ((n*n, x+x) for n,x in enumerate(X))
print list(x)
[(0, 'cc'), (1, 'ii'), (4, 'aa'), (9, 'oo')]

Come una list comprehension, ma un passo
alla volta ("lazy")
```

Il modulo itertools

Offre "mattoni" di alto livello per costruire e manipolare iteratori (inclusi generatori, genexp, ecc).
Ad esempio...

```
def enumerate(seq):
    import itertools as it
    return it.izip(it.count(), seq)
```

Genera un "*abstraction benefit*" (il contrario della comune "*abstraction penalty*" tipica di altri linguaggi...)

importare moduli

```
import math   # modulo di libreria standard
print math.atan2(1, 3)
0.321750554397
print atan2(1, 3)
Traceback (most recent call last):
  ...
NameError: name 'atan2' is not defined
atan2 = math.atan2
print atan2(1, 3)
0.321750554397
# o, scorciatoia: from math import atan2
```

definire moduli

Anche piú semplice...:

- ogni sorgente Python **wot.py é** un modulo
- importabile con **`import wot`**
- purché sia in un directory (o zipfile) elencato in **`sys.path`**
- **`sys.path.append('/aggiungi/directory')`**
- attributi del modulo sono i nomi che definisce
- ovvero le “variabili globali”del modulo
- NB: anche classi e funzioni sono “variabili”!

i package

un package é un modulo che puó contenere altri moduli (e recursivamente altri package)

- vive in un directory contenente **`__init__.py`**
- **`__init__.py`** é il “corpo”, puó essere vuoto
- i moduli del package sono i file nel directory
- i sub-package sono subdirectory
- ma solo quelli che hanno un **`__init__.py`**
- importato e usato con “nomi strutturati”:
- `import email.MIMEImage`
- `from email import MIMEImage`

“le pile sono comprese”

libreria standard Python (grosso modo)...:
180 mod. (`math, sys, os, sets, struct, re, random, pydoc, gzip, threading, socket, sele urllib, ftplib, rfc822, copy, pickle, SimpleXMLRPCServer, telnetlib, ...`)
8 package con altri 70 mod. (`bsddb, compiler, curses, distutils, email, logging, xml...`)
80 moduli codec, 280 unit-test, 180 demo
180 moduli in Tools (12 maggiori+60 minori)

...e non é finita...

“altre pile”: GUI e DB

GUI:

- Tkinter (con Tcl/Tk)
- wxPython (con wxWindows)
- PyQt (con Qt; anche, PyKDE)
- Pythonwin (con MFC – solo Windows)
- Cocoa (solo Mac OS X)
- PyGTK, PyUI, anygui, fltk, FxPy, EasyGUI, ...

DB (relazionali):

- Gadfly, PySQLite, MkSQL (con Metakit)
- MySQL, PostgreSQL, Oracle, SAP/DB, DB2...

“altre pile”: per il calcolo

Numeric (e numarray)
PIL (elaborazione di immagini)
SciPy
    weave (inline, blitz, ext_tools)
    fft, ga, special, integrate, interpolate, ...
    plotting: chaco, plt, xplt, gplt, ...
gmpy (multiprecisione, usa GMP)
pycrypto

“altre pile”: networking

Integrazione con Apache:
- `mod_python` (anche: PyApache)

Framework web di + alto livello (`Webware, Quixote, CherryPy, ...`)

“Application servers”/CMS (Zope, Plone)

...e un framework di rete asincrono con prestazioni, scalabilitá, potenza sempre piú sbalorditive...:
- **Twisted** (include: protocolli, web templating, integrazione con GUI, wrapping di thread / processi / DB, persistenza, config, ...)

integrazione con C/C++/...

Python C API
SWIG
Boost Python, sip, CXX/SCXX (C++)
PyObjC (Objective-C)
pyrex
pyfort, f2py

COM, XPCOM, Corba, AppleEvents,...

integrazione con Java

Jython: integrazione perfetta e trasparente (ma esclusiva -- usa una JVM, non il runtime + classico di Python): importa classi Java, implementa interfacce Java, genera bytecode JVM, interprete accessibile da Java, ...

JPE: integrazione “arm's length” di Python classico + Java

(prospettive analoghe per IronPython con C# e simili, entro .NET / Mono)