Una presentazione di leonardo maffi

(V.1.0, 10 Dic 2006)

Indirizzo di questo documento PDF
(potete scaricarlo anche adesso):
`www.fantascienza.net/leonardo/ar/seminario_py2.pdf`

Con molto materiale di Federico Nati:
`www.orangecoffee.it/federico/corso_python`

Python è un linguaggio:
- Interpretato
- Ad alto livello
- Multipiattaforma
- Object oriented
- Open Source

Vantaggi per questa presentazione:
- Spiegazione rapida
- Già nel vostro account (V.2.1 o 2.3)
- Python on a Stick

Sito ufficiale:
`www.python.org`

Usato per:
- Siti Web
- Data munging
- Interfaccie grafiche
- Collante
- Scripting di applicazioni
- Test e prototipazione
- Ricerca scientifica

Caratteristiche:
- Elegante e molto leggibile (codice indentato)
- Utile come collante
- Estendibilità: albero di moduli
- Facilità di apprendimento
- Massimo dinamismo
- Shell integrata

Timeline:
- Perl apparso nel 1987
- Python creato da Guido nel 1990
- Java creato nei primi anni 90
- PHP apparso attorno al 1995
- Ruby apparso nel 1995
- Versione attuale di Python: 2.5

Guido Van Rossum
Python's BDFL
(Benevolent
Dictator For Life)

Multipiattaforma:
- Linux (incluso)
- Windows
- Macintosh
- Sparc Solaris
- PalmOS
- Psion
- Sharp Zaurus
- WindowsCE
- Play Station 2
- Nokia C60
- etc.

Origini del nome:
Monty Python's Flying Circus
Ma spesso oggi viene associato al pitone

Usato da:

- Google
- Industrial Light+Magic (Star Wars Episode II...)
- Zope corporation (application server)
- Infoseek (search engine)
- Yahoo! Groups (mailing list manager)
- Nasa (vari utilizzi)
- Red Hat (installation software)
- SGI Inc. (vari utilizzi)
- IBM (test automatici di strumenti internet)
- RealNetworks (vari utilizzi)
- Star Trek Bridge Commander
- Blender 3D (scripting language)

Ecc.

La licenza di Python:

- La licenza di Python è Open Source, non è GPL, ma compatibile.

- Python può essere incorporato o modificato a piacere, e il codice Python può essere distribuito liberamente. Può essere usato anche in progetti commerciali.

- È possibile ***non*** distribuire il proprio codice sorgente creando eseguibili che contengono anche l'interprete. Comunque Python è nato con una filosofia OS, e in generale non è molto adatto per la protezione del proprio codice.

`>>> license()` stampa tutta la storia delle licenze.

Perl ha **CPAN**, ***Comprehensive Perl Archive Network***
Python ha le *batterie incluse*, e ***Cheese Shop***:
`cheeseshop.python.org/pypi`
E alcuni altri archivi come il ***Python Cookbook***:
`aspn.activestate.com/ASPN/Python/Cookbook`

```
>>> import this
```

The Zen of Python, by Tim Peters

Definisce in gran parte cosa si intende con ***pythonico***

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Sguardo dall'alto - sintassi semplicissima!

```
>>> print "Hello World!"
Hello World
>>> # Questo è un commento
>>> larghezza = 20
>>> altezza = 5 * 9
>>> larghezza * altezza
900
```

```
>>> a = 1.5 + 0.5j
>>> a
(1.5+0.5j)
>>> a.real
1.5
>>> a.imag
0.5
>>> abs(a)
5.0
```

Python shell come calcolatrice

```
>>> from math import *
>>> 1 + 1
2
>>> sin(2)
0.9092974268256817
>>> log(e)
1.0
>>> sqrt(4)
2.0
>>> # Multiprecisione
>>> 3 ** 45
29543127065508336986 43L
```

Liste

```
>>> a = ['spam', 'eggs', 1, 2]
>>> a[0]
'spam'
>>> a[3]
2
>>> a[-2]
1
>>> a[1:-1]
['eggs', 1]
>>> len(a)
4
```

Stringhe

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[-1]
'A'
```

Il ciclo FOR

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Il ciclo WHILE

```
# Fibonacci series:
>>> a, b = 0, 1
>>> while b < 10:
...         print b
...         a, b = b, a+b
...
1
1
2
3
5
8
```

Comando IF

```
x = int(raw_input("Enter an integer: "))
if x < 0:
     x = 0
     print 'Negative changed to zero'
elif x == 0:
     print 'Zero'
elif x == 1:
     print 'Single'
else:
     print 'More'
```

Funzioni, docstring

```
>>> def fib(n):
...     """Stampa la serie di Fibonacci fino a n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>> help(fib)
Help on function fib in module __main__:

fib(n)
    Stampa la serie di Fibonacci fino a n.
```

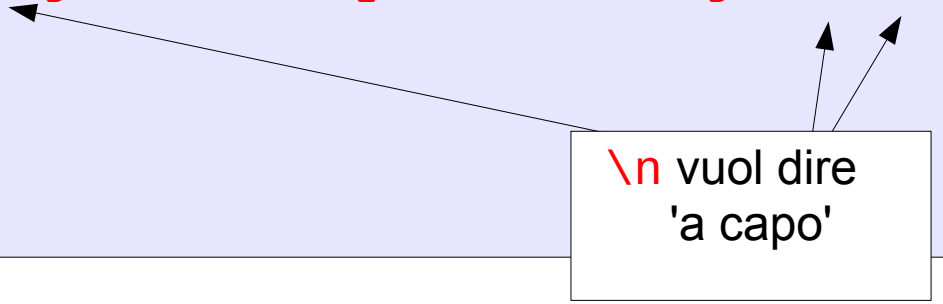
Caricare e usare moduli

```
>>> import math
>>> math.sin(1)
0.8414709848078965
>>> from math import *
>>> log(10, 2)
3.3219280948873626
```

Gestione file

Leggere un file

```
>>> f = open('/etc/lilo.conf')
>>> lc = f.read()
>>> lc[:80]
"# LILO configuration file\n# generated by 'liloconfig'\n#\n#
Start LILO global sect"
>>> lc = lc.split('\n')
>>> lc[0]
'# LILO configuration file'
```



Scrivere su un file

```
>>> miofile = open('dati.txt', 'w')
>>> dati = ['ciao', 'come', 'va']
>>> miofile.writelines(dati) # No newlines added!
>>> miofile.close()
```

Scrittura dati su un file

```
>>> dati1 = [1, 2, 3]
>>> dati2 = [10, 20, 30]
>>> miofile = open('dati.txt', 'w')
>>> for i in range(len(dati1)):
...     miofile.write(str(dati1[i]) + '\t' + str(dati2[i]) + '\n')
...
>>> miofile.close()
```

Output *dati.txt*

```
1       10
2       20
3       30
```

O codice Python più moderno:

```
>>> dati1 = [1, 2, 3]
>>> dati2 = [10, 20, 30]
>>> zip(dati1, dati2)
[(1, 10), (2, 20), (3, 30)]
>>> miofile = open('dati.txt', 'w')
>>> for el1, el2 in zip(dati1, dati2):
...     print >>miofile, "%d\t%d" % (el1, el2)
...
>>> miofile.close()
```

Le classi

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...     def metodomio(self):
...         print "Questo e' il mio metodo..."
...
>>> x = Complex(3.0, 4.5)
>>> x.r
3.0
>>> x.i
4.5
>>> x.metodomio()
Questo e' il mio metodo...
```

Le basi del linguaggio - L'interprete

Nota 1: comodo, non esce dalla shell dopo aver eseguito il modulo:

```
python -i script.py
```

Nota 2: per uso su Linux, aggiungere in `.pythonrc`

```
import rlcompleter
readline.parse_and_bind("tab: complete")
```

Nota 3: esiste anche *ipython*, con varie comodità extra per shell:

```
http://ipython.scipy.org
```

Nota4: Python può essere editato con qualunque editor per testo puro, ma esistono sintassi/modalità per *Emacs, Vim, TextPad, Ultraedit*, *Eclipse*, ecc. Inoltre esistono varie IDE, come *IDLE* (integrata), *Boa Constructor, Eric, SPE*, etc.

Editare gli script

Nota per chi usa Emacs:

www.python.org/cgi-bin/moinmoin/EmacsPythonMode

www.iro.umontreal.ca/~pinard/pymacs/

Nota per chi usa VIM:

www.vex.net/~x/python_and_vim.html

aggiungere in .vimrc :

```
" Indent options:
set tabstop=4
set shiftwidth=4
set smarttab
set expandtab
set softtabstop=4
set ai
```

Per tutti:

MAI MISCHIARE SPAZI E TAB

(di solito si usano 4 spazi)

```
:'<,'>s/^/
/g
```

Nelle righe selezionate sostituisci all'inizio quattro spazi

Alcuni tipi di dato *built-in*

- **Numbers** `3.1415  1234  999L  3+4j`
- **Strings** `'spam' "guido's" '''spam''' """guido's"""`
- **Lists** `[1, 2]  ['a', 'b']  [1, [2, 'three'], 4]`
- **Dictionaries** `{'food':'spam', 'taste':'yum'}`
- **Tuples** `(1, 2, 3)  (1, 'spam', 4.1, 'U')`
- **Files**
```
for line in open('eggs'):
```
- **Sets**
```
s = set('eggs')  s.intersection('as')
```

***I tipi built-in di Python sono sufficienti per la maggior parte degli scopi*, e sono implementati in C. NON** è necessario implementare oggetti e strutture curando la gestione della memoria, implementando routine di accesso e ricerca o altre cose che appesantiscono e distolgono l'attenzione dal vero fine del programma.
Varie altre strutture dati sono nella libreria (biblioteca) standard (pile, array monotipo, dizionari on default, heap, deque, ecc).

Più in dettaglio

Numeri

- Normal integers (C longs) 124 -24 0
- Long integers (unlim. Size) 99999999999L
- Floating-point (C doubles) 1.23 3.14e-10 4E21 4.0e+210
- Complex numbers 3+4j 3.0+4.0j 3J
- Octal and hex constants 0177 0x9ff (attenzione)
- decimal module

```
>>> from math import sin, exp
>>> x = 14./3
>>> sin(x)
-0.9989549170979282 7
>>> exp(-x ** 2)
3.483624072895621e-10
>>> y = 18
>>> x + y
22.666666666666668
```

```
>>> c = 3+4j
>>> c ** 2
(-7+24j)
>>> c.imag
4.0
>>> c.real
3.0
>>> abs(c)
5.0
```

```
>>> x = 3.456
>>> round(x, 1)
3.5
>>> round(x, 2)
3.46
>>> int(x)
3
>>> floor(x)
3.0
>>> ceil(x)
4.0
>>> str(x)
'3.456'
```

Numeri e Operatori

```
>>> x, y = 0, 1
>>> x and y
0
>>> x or y
1
>>> not x
True
>>> x < y
True
>>> x == y
False
>>> x != y
True
```

```
>>> x != y
True
>>> x >= y
False
>>> x * y, x + y, 12*y%5
(0, 1, 2)
>>> y / x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or
modulo by zero
>>> y / (x + 1)
1
>>> 1/2, 1//2, 1.0/2, 1.0//2
(0, 0, 0.5, 0.0)
```

Stringhe

String formatting

```
>>> "Il prezzo di %s mele e' %s" % (5, "2euro")
"Il prezzo di 5 mele e' 2euro"
```

```
>>> s1 = ""
>>> s2 = "spam's"
>>> s1 + s2
"spam's"
>>> s2 * 3
"spam'sspam'sspam's"
>>> s2[0]
's'
>>> s2[1:3]
'pa'
>>> s2[-2:]
"'s"
>>> len(s2)
6
```

Iteration

```
>>> for c in s2: print c
...
s
p
a
m
'
s
>>> for c in s2: print c,
...
s p a m ' s
```

Appartenenza

```
>>> 'pa' in s2
True
>>> 'z' in s2
False
```

Stringhe

Caratteri speciali

\n	Ritorno a capo
\t	TAB
\a	Bell
\" \'	Quotes
\\	Backslash
\b	Backspace
\0XX	Octal XX
\xXX	Hex XX
\r	Carriage return

Caratteri speciali

```
>>> print
"1\t2\t3\n14\t15\t16"
1       2       3
14      15      16
```

```
>>> print "ciao" "come stai"
ciaocome stai
>>> print "ciao", "come stai"
ciao come stai
>>> print """ciao
...  come stai"""
ciao
 come stai
```

```
>>> c = str(3.1415)
>>> print c
3.1415
>>> print c+'ciccio'
3.1415ciccio
>>> ord('~')
126
>>> chr(126)
'~'
```

Editare le stringhe

String slicing

```
Start|  0  |  1  |...  | -2 | -1 |End
```

```
S='ciao'  S[:0] è ""  S[:1] è 'c'  S[-2:] è 'ao'
```

Metodi stringa

```
>>> s = "spammify"
>>> s.upper()
'SPAMMIFY'
>>> s.find('f')
6
>>> int("42")
42
>>> S = "Ciao come stai?"
>>> s.split()
['Ciao', 'come', 'stai?']
>>> s.replace('a', 'A')
'CiAo come stAi?'
```

Nota: le stringhe Python sono oggetti immutabili, non si possono modificare sul posto, es.: S[0]='a' dà errore!

Liste Python (array dinamici)

Elenchi ordinati di oggetti (anche misti)

- Lista vuota — L1 = []
- 4 el. (ind. 0, 1, 2, 3) — L2 = [34, 12, 45, 2]
- Sottoliste — L3 = ['abc', ['def', 'ghi']]
- Indici, slice, length — L2[i] L2[i:j] L3[i][j] len(L2)
- Unire e ripetere — L1+L2 L2*3
- Iterare, membership — for x in L2:... 3 in L2
- Aggiungere e togliere — L2.append(4) L2.pop(-1)
- Ordinare e cercare — L2.sort() sorted(L2) L2.index(1)
- Eliminare elementi — del L2[1] L2[0:2] = []
- Index/slice assignment — L[2] = 0 L[1:3] = [3,4,5]
- Lista di interi — range(10)

Liste: esempi (1)

```
>>> x = range(10)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[0:4]
[0, 1, 2, 3]
>>> x.reverse()
>>> x[0:4]
[9, 8, 7, 6]
>>> x.sort
<built-in method sort of list object at 0x81d46f4>
>>> x.sort()
>>> del x[-3:]
>>> x
[0, 1, 2, 3, 4, 5, 6]
```

```
>>> x.append(25)
>>> x
[0, 1, 2, 3, 4, 5, 6, 25]
>>> x.pop(-1)
25
>>> x
[0, 1, 2, 3, 4, 5, 6]
>>> del x[-1]
>>> x
[0, 1, 2, 3, 4, 5]
```

Liste: esempi (2)

```
>>> x = x + [12,13]
>>> x
[0, 1, 2, 3, 4, 5, 12, 13]
>>> len(x)
8
>>> y = range(len(x))
>>> y
[0, 1, 2, 3, 4, 5, 6, 7]
>>> z = [x, y]
>>> z
[[0, 1, 2, 3, 4, 5, 12, 13], [0, 1, 2, 3, 4, 5, 6, 7]]
>>> x[0] = 1000
>>> z
[[1000, 1, 2, 3, 4, 5, 12, 13], [0, 1, 2, 3, 4, 5, 6, 7]]
>>> x = 1000
>>> z
[[1000, 1, 2, 3, 4, 5, 12, 13], [0, 1, 2, 3, 4, 5, 6, 7]]
```

Liste: esempi (3)

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> sorted(a)
[-1, 1, 66.6, 333, 333, 1234.5]
```

Liste: esempi (4)
Functional programming

filter

```
>>> x = range(10)
>>> def f(x): return x>5
...
>>> filter(f,x)
[6, 7, 8, 9]
>>> filter(lambda(x): x>5, x)
[6, 7, 8, 9]
```

reduce

```
>>> def mult(x,y): return x*y
...
>>> reduce(mult, [1,2,3,4])
24
>>> def somma(x,y): return x+y
...
>>> reduce(somma, range(10))
45
```

map, Definizione di lista

```
>>> x = range(5)
>>> map(lambda(x): x>2, x)
[False, False, False, True, True]
>>> map(lambda(x): x**2, x)
[0, 1, 4, 9, 16]
>>> def f(x): return x % 2
...
>>> map(f, x)
[0, 1, 0, 1, 0]
>>> x = [1, 2, 3, 4]
>>> y = [11, 22, 33, 44]
>>> map(None, x, y)
[(1, 11), (2, 22), (3, 33), (4,
44)]
>>> # Definizione di lista
>>> [el**3 for el in x if el>2]
[27, 64]
```

Dizionari (dict)

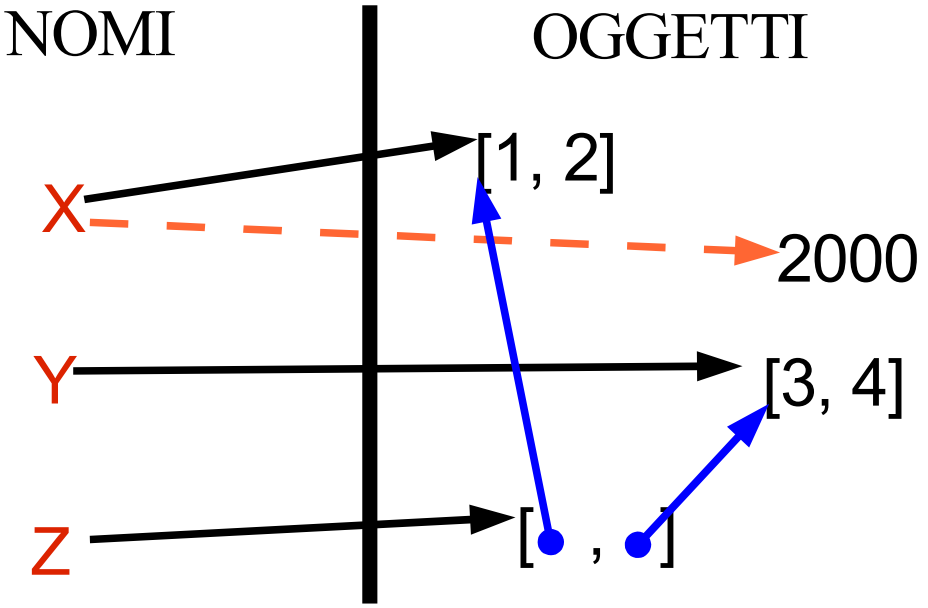
```
>>> dizionario = {'nome':'Calimero', 'anni':5, 'colore':'nero'}
>>> dizionario.items()
[('colore', 'nero'), ('anni', 5), ('nome', 'Calimero')]
>>> dizionario.keys()
['colore', 'anni', 'nome']
>>> dizionario.values()
['nero', 5, 'Calimero']
>>> dizionario['colore']
'nero'
>>> 'nome' in dizionario
True
>>> dizionario['anni'] = 6
>>> dizionario['anni']
6
>>> for k in dizionario: print k
...
colore
anni
nome
>>> [k for k,v in dizionario.iteritems() if v==5]
['anni']
```

Statements

- **Assignments – Create reference** `s = 'ciao'`
- **Calls – Running functions** `stdout.write('ciao\n')`
- **print – Print objects** `print "Hello World!"`
- **if/elif/else – Select action** `if x>10: print x`
- **for/else – Sequence iteration** `for i in range(10): print i`
- **while/else – General loop** `while 1: print "ciao"`
- **pass – Empty placeholder** `while 1: pass`
- **break/continue – Loop jumps** `while 1: if not line: break`
- **import, from – Module access** `from math import *`
- **def/return – Build functions** `def f(x): return x+1`
- **class – Building objects** `class MyClass(): MyData = []`
- **del – Deleting things** `del data[k]  del variabile`
- **exec – Running code strings** `exec "print 'ciao'"`
- **try/except/finally, raise – Catching and raising exceptions**

Assegnamenti

Nomi, Oggetti e *shared references*



Regole per i nomi delle variabili:

- CASE SENSITIVE (Z non è z)
- Deve iniziare con lettere o _ (underscore)
- Può contenere solo numeri, lettere o _
- Parole riservate sono OFF-LIMITS (and, if, not, print, is, pass, def...)

```
>>> x, y = [1, 2], [3, 4]
>>> z = [x, y]
>>> z
[[1, 2], [3, 4]]
>>> x[0] = 1000
>>> z
[[1000, 2], [3, 4]]
>>> x = 2000
>>> z
[[1000, 2], [3, 4]]
>>> y.append('ciao')
>>> z
[[1000, 2], [3, 4, 'ciao']]
>>> y.reverse()
>>> z
[[1000, 2], ['ciao', 4, 3]]
>>> del x
>>> del y
>>> z
[[1000, 2], ['ciao', 4, 3]]
```

if...elif...else

Formato generale

```
if <expression1>:
INDENT<statements>
elif <expression2>:
INDENT<statements>
else:
INDENT<statements>
```

Dopo i DUE PUNTI il codice va indentato

```
anni = input('Quanti anni hai? ')

if not isinstance(anni, (int, long)):
    print "Devi inserire un numero intero!"
elif anni < 0:
    print 'Mi prendi in giro?'
elif anni <= 3:
    print 'Non credo che tu sappia già leggere...'
elif anni >= 120:
    print 'Scusa se nutro qualche dubbio'
else:
    print "Hai %s anni, be', ti facevo più vecchio." % anni
```

Nota: non esiste case/switch. Si usano gli elif.

Andare a capo e commentare

\ (backslash) permette di andare a capo
Ma dentro le parentesi () {} [] non serve

```
                    A CAPO
print 'linea molto lunga voglio\
 andare a capo!'
if (condizione molto lunga voglio
    andare a capo):
x = [1, 2, 3, 4
     5, 6]
```

commenta tutto ciò che segue

```
                   COMMENTI
# questo e' un commento
print 'ciao' # anche questo
```

Test di verità

- True significa qualunque numero non nullo o oggetto non vuoto.
- False significa non vero: sono False il numero 0, oggetti vuoti e la parola chiave None.
- Test di uguaglianza o confronto hanno come risultato True o False.
- Gli operatori booleani and, not e or hanno come risultato un oggetto vero o falso (Not restituisce un bool, gli altri possono restituire oggetti veri).

```
>>> 10 < 1
False
>>> 10 > 3
True
>>> 10 or 3
10
>>> 10 and 3
3
```

```
>>> 10 and 0
False
>>> 10 or 0
10
>>> 0 or 10
10
>>> not 0 or 10
True
>>> "" or 10
10
>>> "ciao" or 10
'ciao'
```

```
>>> x = range(10)
>>> y = range(10)
>>> x == y
True
>>> x is y
False
>>> z = x
>>> z == y
True
>>> z is x
True
>>> z is y
False
```

Cicli while

Formato generale

```
while <test>:
    <statements1>
else:
    <statements2>
```

Es. 1

```
>>> while 1:
...     print "non mi fermo piu'!"
...
non mi fermo piu'!
non mi fermo piu'!
non mi fermo piu'!
non mi fermo piu'!
 mi fermo piu'!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
>>>
```

Es. 2

```
>>> stringa = "ciao"
>>> while stringa:
...     print stringa
...     stringa = stringa[:-1]
...
ciao
cia
ci
c
>>>
```

Es. 3

```
>>> a, b = 0, 10
>>> while a < b:
...     print a,
...     a += 1
...
0 1 2 3 4 5 6 7 8 9
>>>
```

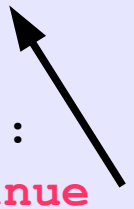
While / break / else

```
>>> a, b = 0, 10
>>> while a < b:
...     print a,
...     a += 1
... else:
...     print "a e' diventato piu' grande di b!"
...
0 1 2 3 4 5 6 7 8 9 a e' diventato piu' grande di b!
>>>
```

```
>>> a, b = 0, 10
>>> while a < b:
...     print a,
...     a = a+1
...     if a == 7:
...         break
... else:
...     print "a e' diventato piu' grande di b!"
...
0 1 2 3 4 5 6
>>>
```

While / continue

```
>>> a, b = 0, 10
>>> while a < b:
...     a = a + 1
...     if a == 7:
...         continue
...     print a,
... else:
...     print "a e' diventato piu' grande di b!"
...
1 2 3 4 5 6 8 9 10 a e' diventato piu' grande di b!
>>>
```



Cicli for

Formato generale

```
for <target> in <object>:
    <statements1>
```

```
for <target> in <object>:
    <statements1>
else:
    <statements2>
```

```
for <target> in <object>:
    <statements1>
    if <test1>: break
    if <test2>: continue
else:
    <statements2>
```

for si può fare sugli oggetti *ITERABILI* (stringhe, liste, tuple, set, e altri data-type che costituiscono sequenze.)

for: esempi (1)

```
>>> for el in ['spam', 'eggs', 'ham']:
...     print el
...
spam
eggs
ham
```

```
>>> stringa = "ciao"
>>> for carattere in stringa:
...     print carattere * 3
...
ccc
iii
aaa
ooo
```

```
>>> somma = 0
>>> for x in [1, 2, 3, 4]:
...     somma += x # Significa: somma = somma + x
...
>>> somma
10
>>> sum([1, 2, 3, 4])
10
```

for: esempi (2)

```
>>> txt = 'ciao'
>>> for i in range(len(txt)):
...     print i, txt[i]
...
0 c
1 i
2 a
3 o
```

```
>>> txt = 'ciao'
>>> for i,c in enumerate(txt):
...     if i > 2: continue
...     print i, c
... else:
...     print "finito!"
...
0 c
1 i
2 a
finito!
```

```
>>> # Oggi si userebbe:
>>> txt = 'ciao'
>>> for i,c in enumerate(txt):
...     if i > 2: break
...     print i, c
... else:
...     print "finito!"
...
0 c
1 i
2 a
```

Funzioni

Permettono il **riutilizzo del codice** e di decomporre il programma in **procedure**.

```
def <nomefunzione>(<arg1>, <arg2>, <arg3>,..., <argN>):
    <statements>
    return <value>
```

crea una funzione di nome <nomefunzione>

restituisce un oggetto al codice che chiama la funzione

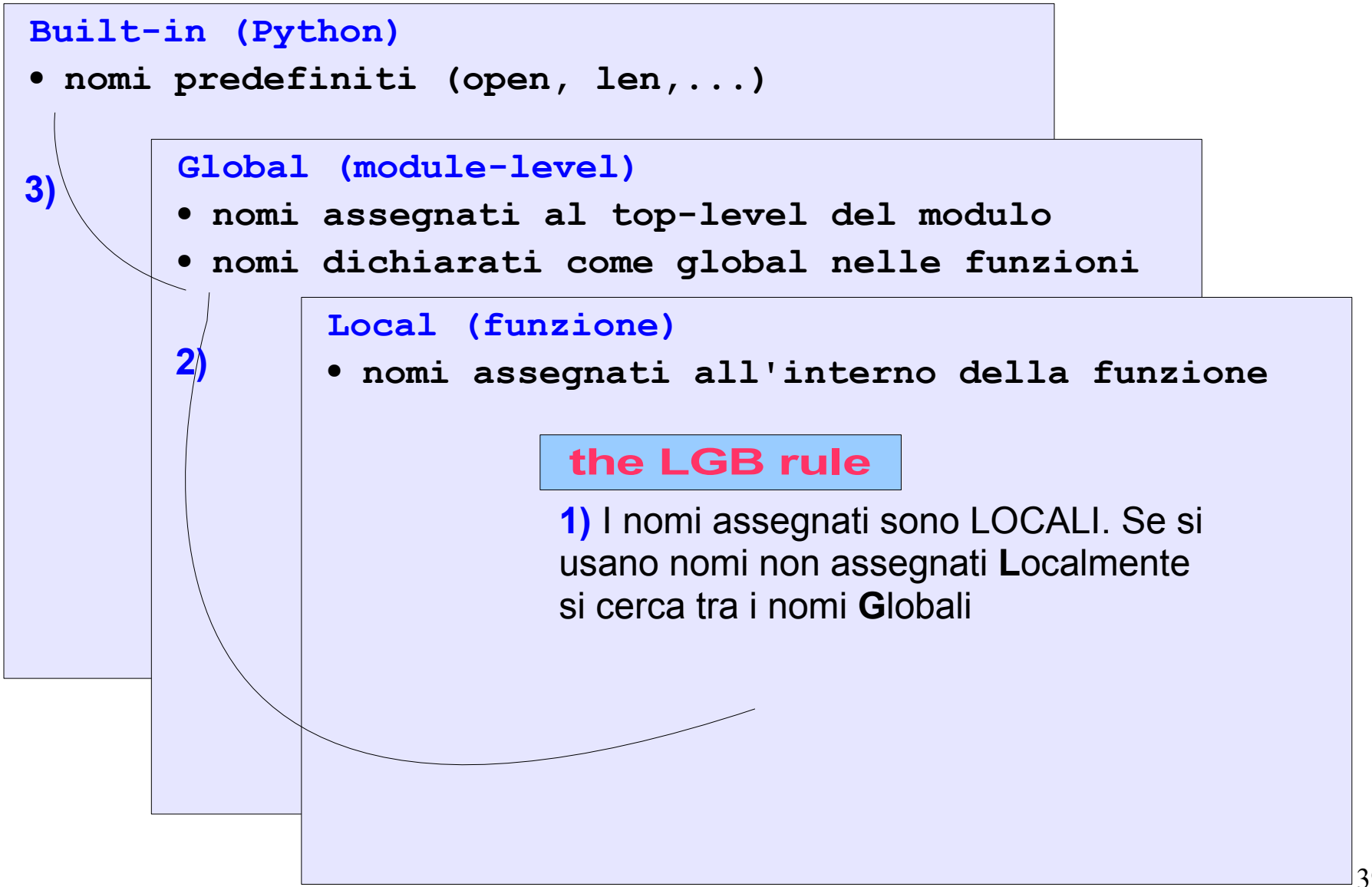
```
def <nomefunzione>(<arg1>, <arg2>, <arg3>,..., <argN>):
    <statements>
    global <name>
    return <value>
```

definisce il nome come a *module-level*

I namespace (Spazi di nomi)

```
>>> x = 10
>>> x
10
>>> import sys
>>> sys.platform
'win32'
>>> platform
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'platform' is not defined
>>> from sys import platform
>>> platform
'win32'
```

I namespace (spazi di nomi)



Funzioni: esempi

```
>>> def mult(x, y):
...     return x * y
...
>>> mult(3, 6)
18
>>> mult(3, 'Ciao')
'CiaoCiaoCiao'
```

```
>>> def fatt(n):
...     if n <= 1: return 1
...     return n * fatt(n-1)
...
>>> fatt(0)
1
>>> fatt(1)
1
>>> fatt(4)
24
```

```
>>> def intersection(seq1, seq2):
...     result = []
...     for el in seq1:
...         if el in seq2:
...             result.append(el)
...     return result
...
>>> s1 = 'Ciao xyz123'
>>> s2 = 'Ciao pqr2'
>>> intersection(s1, s2)
['C', 'i', 'a', 'o', ' ', '2']
>>> set(s1).intersection(s2)
set(['a', ' ', 'C', 'i', 'o', '2'])
```

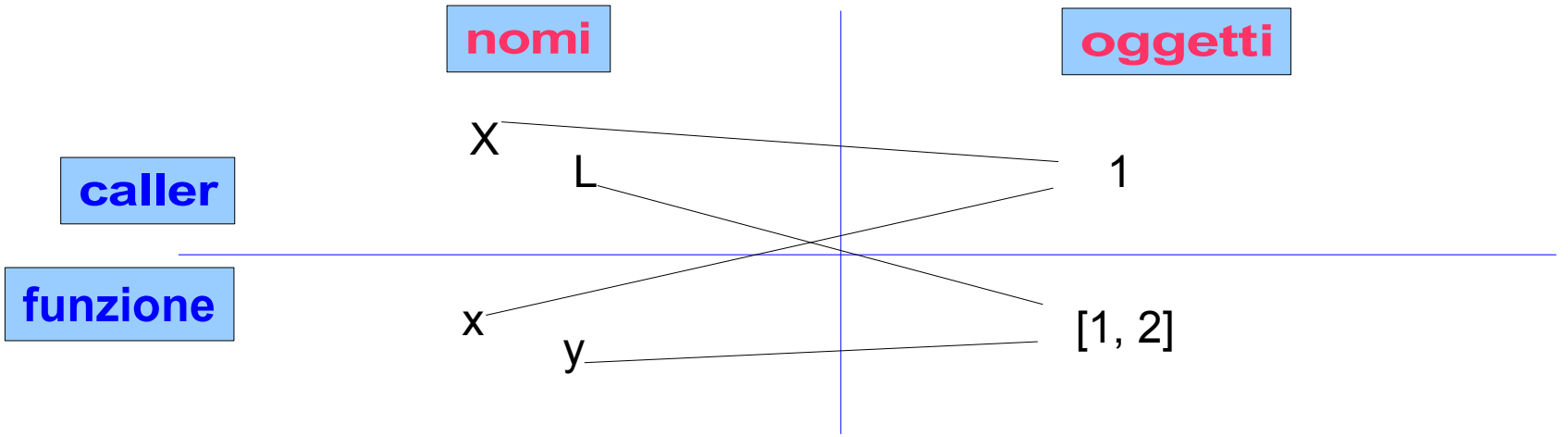
```
>>> a = 10
>>> def f(x):
...     global g
...     g = 14
...     return x * a
...
>>> f(2)
20
>>> g
14
```

Passaggio argomenti

- **In Python gli argomenti sono passati *by object assignment***, ovvero assegnando gli oggetti ai nomi locali
- Gli assegnamenti effettuati dentro la funzione non influenzano il codice chiamante.
- Modificando un oggetto mutabile (es. una lista) dentro una funzione si può influenzare il codice chiamante (gli argomenti immutabili si comportano come al C "*by value*", quelli modificabili al C "*by pointer*")
- Comunque si può passare una copia dell'oggetto.

Passaggio argomenti: un esempio

```
>>> def changer(x, y):
...     x = 2
...     y[0] = 'spam'
...
>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)
>>> X
1
>>> L
['spam', 2]
```



Argument matching

```
>>> def f(x, y): return x / y
...
>>> f(30.0, 10.0)
3.0
>>> f(10.0, 30.0)
0.33333333333333331
>>> f(y=10.0, x=30.0)
3.0
```

```
>>> def stampa(s, t, u):
...     print s, t, u
...
>>> stampa('ciao', 'come', 'stai')
ciao come stai
>>> stampa(u='ciao', s='come', t='stai')
come stai ciao
>>> stampa('ciao', u='come', t='stai')
ciao stai come
```

Argument defaults

```
>>> def stampa(s, t='', u='.'):
...     print s + t + u
...
>>> stampa('ciao')
ciao.
>>> stampa('ciccio')
ciccio.
>>> stampa('ciccio', 'bello')
cicciobello.
>>> stampa('ciccio', 'bello', '\n')
cicciobello

>>> stampa()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: stampa() takes at least 1 argument (0 given)
```

Un def *senza return* definisce quello che talvolta si chiama *procedura*.
In Python viene comunque resituito l'oggetto None

Numero variabile di argomenti

```
>>> def stampa_primo_carattere(*stringhe):
...     for s in stringhe:
...         print s[0]
...
>>> stampa_primo_carattere('ciao')
c
>>> stampa_primo_carattere('ciao', 'ciccio')
c
c
>>> stampa_primo_carattere('ciao', 'ciccio', 'bello')
c
c
b
>>> varie = ['foo', 'bar', 'spam']
>>> stampa_primo_carattere(*varie)
f
b
s
```

Esiste anche la sintassi ** per passare/ricevere parametri con nome.

lambda

```
>>> lambda x, y, z: x ** y - z
<function <lambda> at 0x812653c>
>>> f = lambda x, y, z: x**y - z
>>> f(2, 3, 4)
4
```

- lambda crea una funzione anonima
- va scritta su una sola riga e senza return
- per il resto vale quanto detto per def
- lambda *è una espressione*, non uno statement, e può contenere solo espressioni

```
>>> L = [lambda x: x**2, lambda x: x**3, lambda x: x**4]
>>> for funzione in L:
...     print funzione(3)
...
9
27
81
```

Esempio: fattoriale con lambda

```
>>> fatt = lambda n: n-1 + abs(n-1) and fatt(n-1)*n or 1
>>> fatt(0)
1
>>> fatt(1)
1
>>> fatt(2)
2
>>> fatt(6)
720
```

Le monolinee sono sconsigliate per molte ragioni, inoltre tale monolinea è poco leggibile. Oggi volendo scrivere ciò come monolinea su Python 2.5 si userebbe:

```
>>> fatt = lambda n: fatt(n-1)*n if n>1 else 1
```

Ancora sulle funzioni

- Le funzioni sono oggetti come tutti gli altri; i loro nomi sono solo references a tali oggetti, e possono essere riassegnati;
- Dal momento che gli argomenti sono passati assegnando gli oggetti, le funzioni possono essere argomenti;
- * (apply) è utile per applicare funzioni arbitrarie su argomenti arbitrari;
- map è una funzione built-in che permette di applicare una funzione su di una lista (ma spesso nel Python moderno si usano i generatori di lista o descrittori di lista).

```
>>> def f(x): return x**2
...
>>> g = f
>>> g(5)
25
>>> def h(funzione, arg):
...     return funzione(arg)**2
...
>>> h(g, 5)
625
>>> def mul(x, y): return x*y
>>> l = [3, 2]
>>> mul(*l)
6
>>> map(f, [1, 2, 3, 4])
[1, 4, 9, 16]
>>> [x*x for x in [1, 2, 3, 4]]
[1, 4, 9, 16]
>>> l= (x*x for x in [1,2,3,4])
>>> list(l)
[1, 4, 9, 16]
```

Garbage Collection

- Il garbage collector si occupa di liberare la memoria da tutti gli oggetti che non sono più utilizzati dal programma
- In vari linguaggi si deve invece farlo a mano (e se si sbagla sono guai)
- Python in genere *non fa copie* dello stesso oggetto ma solo riferimenti
- Se si vuole una copia si deve chiedere esplicitamente con copy
- Il garbage collector è un sistema 'asincrono' che di tanto in tanto in modo automatico fa questo controllo e libera eventualmente la memoria. Un oggetto viene deallocato se e solo se non ci sono più riferimenti ad esso (per usi più sofisticati sono disponibili anche riferimenti weak).
- Si può controllare, forzare o disabilitare esplicitamente il garbage collector attraverso l'apposito modulo gc
- Il garbage collector di Python è nato semplice, ma poi è stato raffinato fino ad ottenere un risultato adeguato. È un *reference count* unito ad un algoritmo per individuare riferimenti circolari

Moduli

- In termini pratici, i moduli sono file contenenti codice
- Il codice può essere sia in Python che compilato in estensioni C
- A cosa servono:
  - Riutilizzo del codice
  - Raggruppamento dei componenti
  - Condivisione di servizi, dati, o strutture di dati
- In termini più astratti, essi definiscono dei *namespace,* e i nomi corrispondenti si chiamano attributi. Come anche le funzioni, oggetti, classi, numeri, ecc, anche i moduli sono oggetti.

I moduli sono *namespace*

- Ogni nome assegnato in un modulo diventa un suo attributo
- Ogni statement in un modulo viene eseguito una sola volta quando il modulo viene importato. import è quindi un comando, e la sua collocazione nel codice chiamante non è arbitraria
- Il modulo definisce un namespace. I namespace in Python sono quasi sempre dei dizionari. Questi dizionari diventano attributi del modulo dopo l'import. In altre parole:
- Un namespace è un dizionario. In un modulo si chiama `__dict__`. Le chiavi sono i nomi, i valori sono gli oggetti ad essi assegnati

Attributi di un modulo

Modulo *miomodulo.py*

```
x = 20
y = 30
```

```
>>> import miomodulo
>>> miomodulo.x
20
>>> miomodulo.y
30
>>> dir(temp)
['__builtins__', '__doc__', '__file__', '__name__', 'x', 'y']
```

Moduli: esempi (1)

Modulo *miomodulo.py*

```
import math
def miafunzione(x):
    """Somma ad x il seno di x."""
    return x + math.sin(x)
```

```
>>> import miomodulo
>>> miomodulo.miafunzione(140)
140.98023965944031
>>> from miomodulo import miafunzione
>>> miafunzione(140)
140.98023965944031
>>> from miomodulo import miafunzione as f
>>> f(140)
140.98023965944031
>>> reload (miomodulo)
<module 'miomodulo' from 'miomodulo.pyc'>
```

Moduli: esempi (2)

Modulo *miomodulo.py*

```
from math import sin

x = 10
_x = 10

def miafunzione(x):
    return x + sin(x)

def _miafunzione(x):
    return x + sin(x)
```

Ciò che inizia con undescore non viene importato quando si fa un import *

```
>>> from miomodulo import *
>>> x
10
>>> _x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name '_x' is not defined
>>> miafunzione(10)
9.4559788891106304
>>> _miafunzione(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name '_miafunzione' is not
defined
>>> import miomodulo
>>> miomodulo._x
10
>>> from miomodulo import _miafunzione
>>> _miafunzione(10)
9.4559788891106304
```

Note sui moduli (3)

Modulo *miomodulo.py*

```
from math import sin

def miafunzione(x):
    """Somma ad x il seno di x."""
    return x + sin(x)

if __name__ == "__main__":
    import sys
    x = float(sys.argv[1])
    print miafunzione(x)
```

```
C:\tests>python miomodulo.py 10
9.45597888911
```

Su Linux/mac si può inserire in testa uno speciale 'commento', per far eseguire il codice Python come script:

```
#!/usr/bin/env python
```

Note sui moduli (4)

```
>del miomodulo.py
>python miomodulo.pyc 10
9.45597888911
```

- Quando si fa import il codice viene compilato in *bytecode*, un file contenente codice binario che gira sulla macchina virtuale di Python, come in Java.
- Quando un modulo viene eseguito direttamente, il bytecode non viene salvato su disco.
- I file .pyc sono sufficienti per eseguire l'import
- i moduli vengono cercati nei path specificati nella variabile d'ambiente PYTHONPATH e nella directory corrente. Es. su Linux:

```
export PYTHONPATH=$PYTHONPATH:$HOME/python
```

File

- Crea output file `output = open('tmp/spam', 'w')`
- Crea input file `input = open('data', 'r')`
- Legge tutto il file `S = input.read()`
- Legge N bytes `S = input.read(N)`
- Legge la riga succ. `S = input.readline()`
- Lista di righe `L = input.readlines()`
- Scrive stringhe su file `output.write(S)`
- Scrive tutti gli el. in L `output.writelines(L)`
- Chiude manualmente `output.close()`
- Sposta il puntatore `file.seek(0)`
- Itera sulle linee `for l in file: ...`

File: esempi (1)

```
>>> miofile = open('miofile.txt', 'w')
>>> miofile.write('Ciao, come stai?\n')
>>> miofile.close()
>>> miofile = open('miofile.txt')
>>> miofile.readline()
'Ciao, come stai?\n'
>>> miofile.readline()
''
```

File: esempi (2)

File *mieidati.dat*

```
1    20
2    23
3    26
4    29
5    30
```

```
>>> file = open('mieidati.dat')
>>> file.readlines()
['1   20\n', '2   23\n', '3   26\n', '4   29\n', '5   30\n']
>>> file.readlines()
[]
>>> file.seek(0)
>>> datistring = file.readlines()
>>> line = datistring[0]
>>> line
'1   20\n'
>>> line.split()
['1', '20']
>>> line.split()[0]
'1'
>>> float(line.split()[0])
1.0
```

File: esempi (2)

File *mieidati.dat*

```
1    20
2    23
3    26
4    29
5    30
```

```
>>> dati = [map(float, l.split()) for l in open('mieidati.dat')]
>>> dati
[[1.0, 20.0], [2.0, 23.0], [3.0, 26.0], [4.0, 29.0], [5.0, 30.0]]
>>> dati[0]
[1.0, 20.0]
>>> dati[0][0]
1.0
```

File: esempi (3)

```
>>> file = open('mieidati.bin', 'rb')
>>> file.read(4)
';\xad\x0c\x00'
>>> file.read(4)
'<\xad\x0c\x00'
>>> file.read(4)
'=\xad\x0c\x00'
>>> import struct
>>> file.seek(0)
>>> x = struct.unpack('i', file.read(4))
>>> x
(830779,)
>>> x = struct.unpack('10i', file.read(40))
>>> x
(830780, 830781, 830782, 830783, 830784, 830785, 830786, 830787,
830788, 830789)
```

File: note

- I moduli marshal e pickle permettono di salvare e ricaricare qualunque oggetto in un file
- Il modulo os fornisce anche la possibilità di operare con le pipe

```
>>> import pickle
>>> x = arange(10) # From SciPy
>>> file = open('miofile.dat', 'wb')
>>> pickle.dump(x, file)
```

```
>>> import pickle
>>> x=pickle.load(open('miofile.dat'))
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Nota: marshal funziona con la stessa sintassi, è ottimizzato per la serializzazione di oggetti semplici

Intercettare gli errori: le *exceptions*

```
>>> def fun(x):
...     return 1.0 / (5 - x)
...
>>> for i in xrange(20):
...     print fun(i)
...
0.2
0.25
0.333333333333
0.5
1.0
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  File "<stdin>", line 2, in fun
ZeroDivisionError: float division
```

Intercettare gli errori: le *exceptions*

```
>>> try:
...     for i in xrange(20):
...         print fun(i)
... except ZeroDivisionError:
...     print "Mi spiace, non si puo' dividere per zero..."
...
0.2
0.25
0.333333333333
0.5
1.0
Mi spiace, non si puo' dividere per zero...
```

Exceptions

Primo tipo

```
try:
    <statements>
except <name1>:
    <statements1>
except <name2>, <data2>:
    <statements2>
else:
    <statements>
```

Secondo tipo

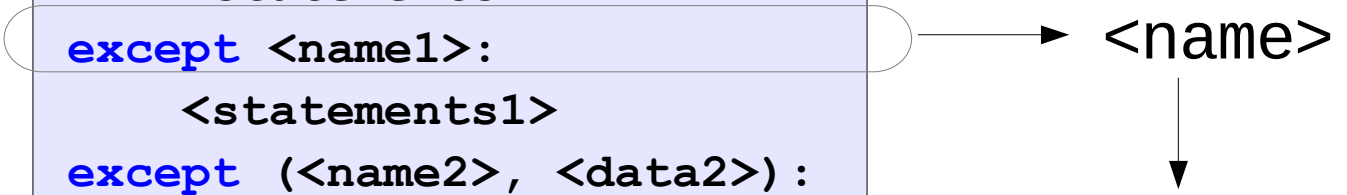
```
try:
    <statements1>
finally:
    <statements2>
```

- In Python 2.5 i due tipi sono stati *fusi assieme*, semplificandone così l'uso.
- In Python le eccezioni vengono usate molto liberamente, talvolta anche per effettuare controlli o per organizzare il programma, ad esempio al posto di codici d'errore restituiti da funzioni.

Exceptions

Primo tipo

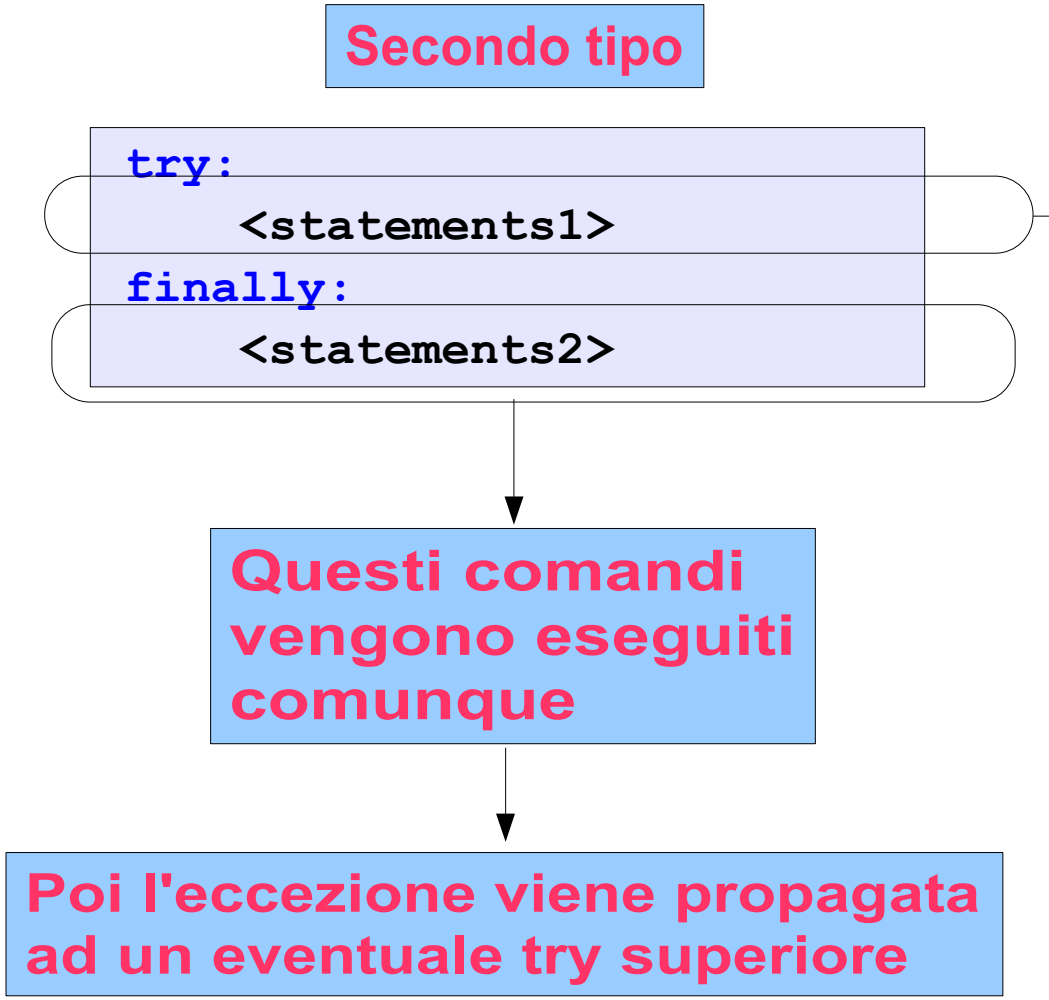
```
try:
    <statements>
except <name1>:
    <statements1>
except (<name2>, <data2>):
    <statements2>
else:
    <statements>
```



Si cerca il primo except <name> e si eseguono i comandi corrispondenti; se non è prevista l'eccezione giusta, essa viene sollevata. Se non viene sollevata nessuna eccezione si va ad else

- Il *solo except* non seguito da <name> intercetta *qualunque exception,* ***sconsigliabile!***
- *Se else non c'è*, e nessun except ha intercettato, allora *l'eccezione passa* all'eventuale blocco superiore *try*

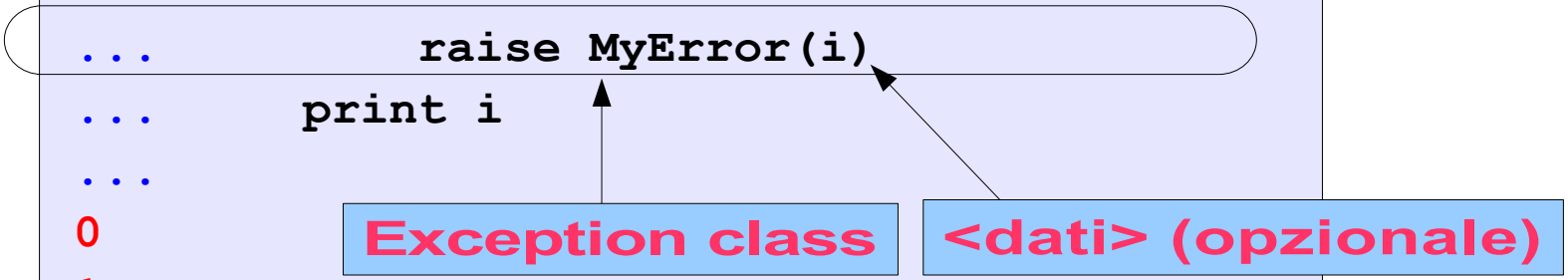
Exceptions



Primo caso: non dà errore: esegue prima try, e poi finally.
Secondo caso: dà errore: esce dal blocco try, ma vengono comunque eseguiti i comandi del blocco finally

Definire nuove exceptions – raise

```
>>> for i in range(10):
...     if i > 5:
...         raise MyError(i)
...     print i
...
0
1
2
3
4
5
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
MyError: 6
```



Nota: i debugger

- È possibile usare il Python debugger, pdb, per isolare i problemi. Come altri debugger (p. es. gdb) pdb permette di eseguire il codice passo dopo passo, controllando i valori delle variabili, impostare breakpoints e cosi' via...
- IDLE (e altre IDE migliori, come ActivePython, SPE, etc) facilitano l'uso del debugger attraverso una interfaccia grafica.

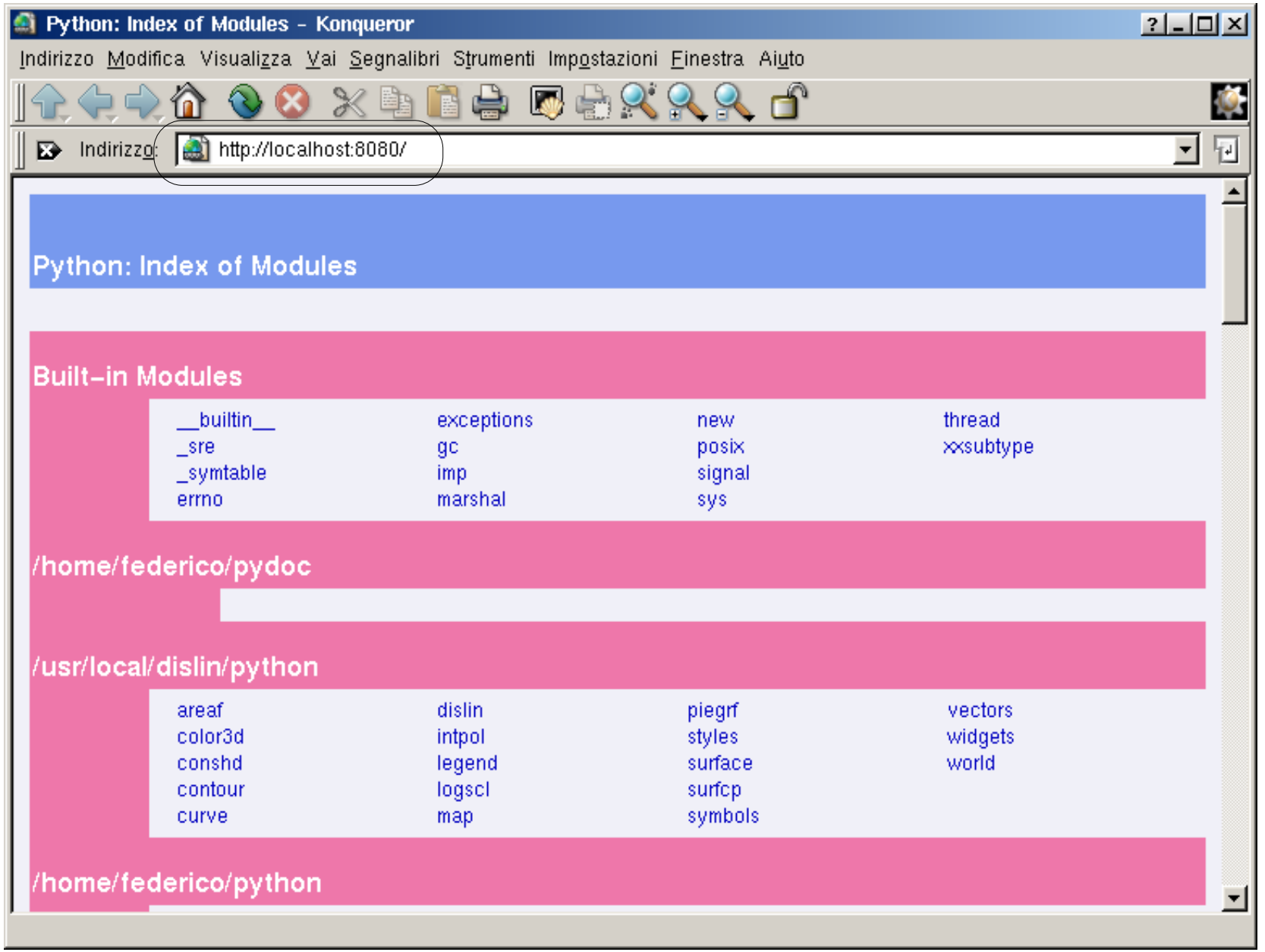
built-in tools

Esistono vari comandi/funzioni built-in che permettono di eseguire operazioni molto utili, ad esempio:

- dir(), help()
- str()
- int() e float()
- hex() ed oct()
- ord() e chr()
- min(i, j, k,...) e max (i, j, k,...)
- exec ed eval
- cgi ed urllib
- abs()
- enumerate(), reversed()
- any() e all() in Python 2.5
- Molti moduli come sys, os, time, etc.

Pydoc

Un modo diffuso in Python per la generazione automatica della documentazione



OOP
class: un esempio

```
class Fiat500:
    def __init__(self):
        self.colore = 'nero'
        self.cappotta = 'chiusa'
        self.stato_fari = False
    def apri_cappotta(self):
        self.cappotta = 'aperta'
    def accendi_fari(self):
        self.stato_fari = True
```

```
>>> miaauto = Fiat500()
>>> miaauto.colore
'nero'
>>> miaauto.cappotta
'chiusa'
>>> miaauto.apri_cappotta()
>>> miaauto.cappotta
'aperta'
>>> miaauto.colore = 'rosso'
>>> miaauto.colore
'rosso'
>>> tuaauto = Fiat500()
>>> tuaauto.colore
'nero'
```

miaauto = Fiat500()

metodo()

```
>>> miaauto.apri_cappotta()
```

attributo

```
>>> miaauto.stato_tergicristallo
False
```

attributo

```
>>> miaauto.colore
'nero'
```

metodo che agisce su attributo

```
>>> miaauto.stato_fari
False
>>> miaauto.accendi_fari()
>>> miaauto.stato_fari
True
```

```
>>> for ruota in miaauto:
...     print ruota.usura
```

oggetti con membership

Class – esempio (2)

```
class Sfera(object):
    pi = 3.1415
    def __init__(self, raggio):
        self.raggio = raggio
    def getVolume(self):
        return (4.0/3.0) * Sfera.pi * (self.raggio ** 3)
```

```
>>> s = Sfera(10)
>>> s.raggio
10
>>> s.getVolume()
4188.6666666666661
>>> s.volume
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Sfera' object has no attribute 'volume'
```

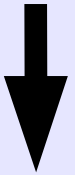
Class – esempio (3)

```
class Sfera(object):
    pi = 3.1415

    def __init__(self, raggio):
        self.raggio = raggio

    def getVolume(self):
        return (4.0/3.0) * Sfera.pi * self.raggio ** 3

    def setVolume(self, volume):
        self.raggio = (volume * 3.0/ 4.0 / Sfera.pi) ** (1.0 / 3.0)

    volume = property(getVolume, setVolume)
```

```
>>> s = Sfera(10)
>>> s.raggio
10
>>> s.volume
4188.6666666666661
>>> s.volume = 10
>>> s.raggio
1.33651775681
```

Ereditare le classi – esempio (1) (Mascheratura di metodo)

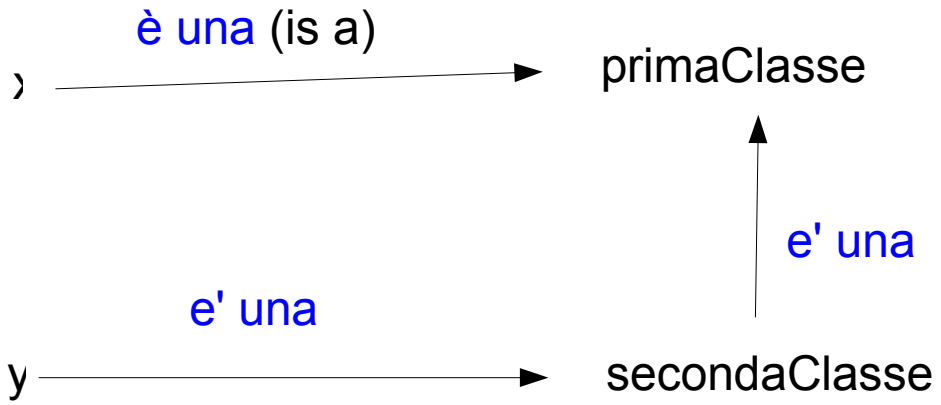
```
>>> class primaClasse:
...     def __init__(self):
...         self.attributo = 'Attributo 1'
...     def stampa_attributo(self):
...         print self.attributo
...
>>> x = primaClasse()
>>> x.stampa_attributo()
Attributo 1
```



```
>>> class secondaClasse(primaClasse):
...     def stampa_attributo(self):
...         print "L'attributo e'", self.attributo
...
>>> y = secondaClasse()
>>> y.stampa_attributo()
L'attributo e' Attributo 1
```

Ereditare le classi – esempio (2)

```
>>> y.attributo = 10
>>> y.stampa_attributo()
L'attributo è' 10
>>> x.stampa_attributo()
Attributo 1
```



**Ci sono 2 istanze,
2 classi,
4 namespaces**

Ereditarietà Multipla (1)

```
class Cubo_statico(object):
    def __init__(self, lato):
        self.lato = lato
    def setVolume(self, volume):
        self.lato = volume ** (1.0 / 3.0)
    def getVolume(self):
        return self.lato ** 3
    volume = property(getVolume, setVolume)

class Solido(object):
    def scalaVolume(self, s):
        self.volume = self.volume * s
    def scalaDimensione(self, s):
        self.volume = self.volume * (s ** 3)

class Sfera(Sfera_statica, Solido): pass
class Cubo(Cubo_statico, Solido): pass
```

polimorfismo

Ereditarietà Multipla (2)

```
>>> from geometria import Cubo, Sfera
>>> c, s = Cubo(10.), Sfera(10.)
>>> print "Volume Cubo: %s\t Volume Sfera: %s" % (c.volume, s.volume)
Volume Cubo: 1000.0      Volume Sfera: 4188.79020479
>>> print "Lato Cubo: %s\t Raggio Sfera: %s" % (c.lato, s.raggio)
Lato Cubo: 10.0  Raggio Sfera: 10.0
>>>
>>> c.scalaVolume(2)
>>> s.scalaVolume(2)
>>> print "Volume Cubo: %s\t Volume Sfera: %s" % (c.volume, s.volume)
Volume Cubo: 2000.0      Volume Sfera: 8377.58040957
>>> print "Lato Cubo: %s\t Raggio Sfera: %s" % (c.lato, s.raggio)
Lato Cubo: 12.5992104989         Raggio Sfera: 12.5992104989
>>> c.scalaDimensione(2)
>>> s.scalaDimensione(2)
>>> print "Volume Cubo: %s\t Volume Sfera: %s" % (c.volume, s.volume)
Volume Cubo: 16000.0     Volume Sfera: 67020.6432766
>>> print "Lato Cubo: %s\t Raggio Sfera: %s" % (c.lato, s.raggio)
Lato Cubo: 25.1984209979         Raggio Sfera: 25.1984209979
```

Operator Overloading

- Gli operatori (+, -, *,...) sono in realtà metodi con i quali si gestisce l'interazione tra oggetti
- Alcuni comandi agiscono in realtà su metodi dell'oggetto, come print, len...
- Slicing e indexing sono gestiti dai metodi dell'oggetto con cui si sta operando
- È quindi possibile ereditare le classi e ridefinire tali metodi. In questo modo si controlla il modo in cui si comportano i nostri ogetti nei confronti degli operatori che di solito compaiono con gli oggetti built-in

Operator Overloading

```
class Solido(object):
    def scalaVolume(self,s):
        self.volume=self.volume * s
    def scalaDimensione(self,s):
        self.volume=self.volume * (s ** 3)
    def __add__(self, other):
        return self.volume + other.volume

    __radd__ = __add__

    def __repr__(self):
        return "Sono un solido di volume " + `self.volume`
class Sfera(Sfera_statica, Solido): pass
class Cubo(Cubo_statico, Solido): pass
```

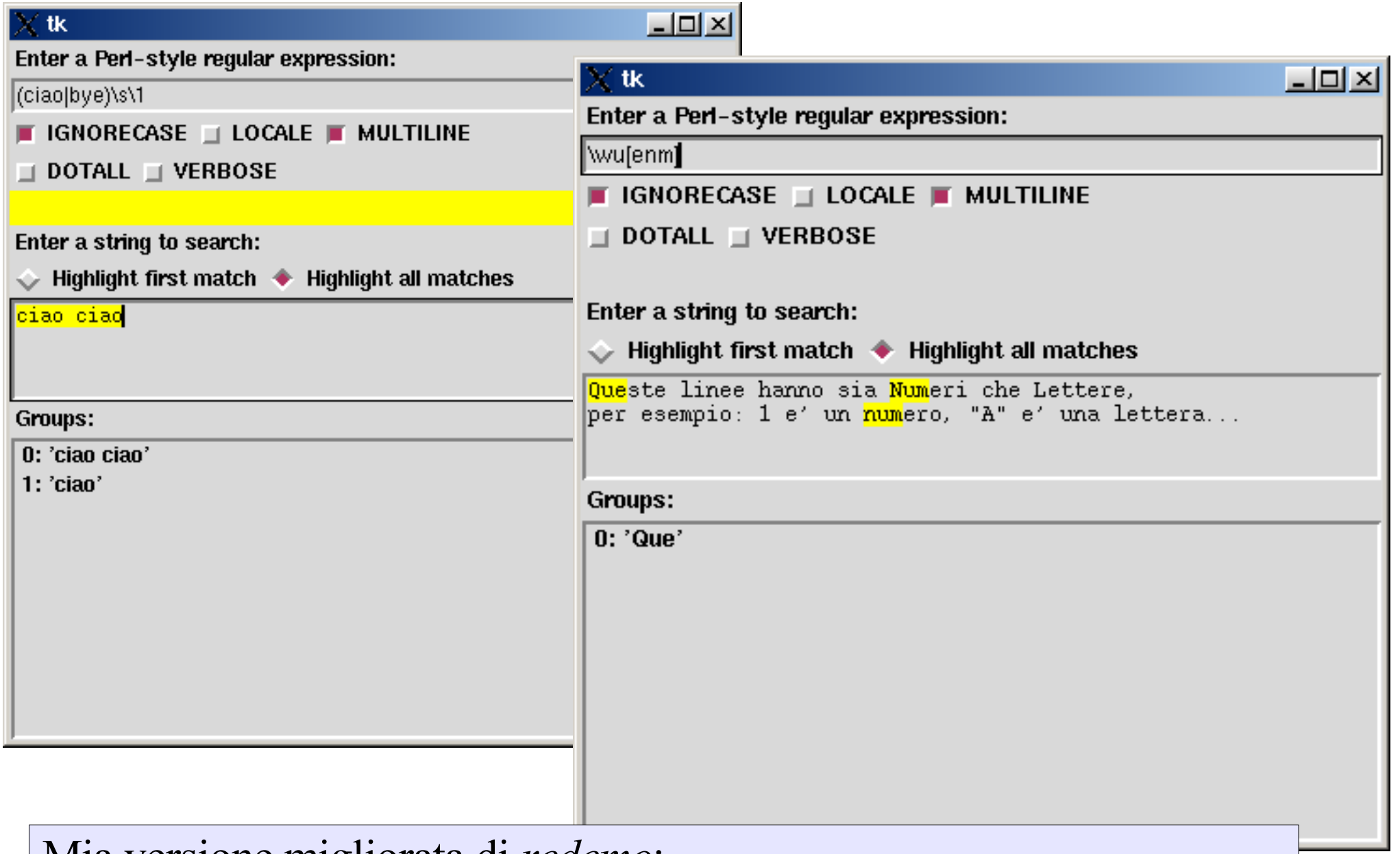
```
>>> c, s = Cubo(10.0), Sfera(10.0)
>>> c.volume, s.volume
(1000.0, 4188.7902047863909)
>>> c + s
5188.7902047863909
>>> print c
Sono un solido di volume 1000.0
```

Regular Expression

- Le RE sono un mini linguaggio molto specializzato disponibile in Python attraverso il modulo re
- Serve per indicare un pattern, una serie di caratteri, a cui si fanno corrispondere delle stringhe (una o più)
- La sintassi è molto simile ma non identica a quella del Perl
- Le routines che eseguono i matching sono ottimizzate in C
- Sono la maniera più potente e diffusa di operare con stringhe di caratteri (ricerca in file di testo o in database, sostituzione di caratteri, editing di stringhe...) in vari linguaggi e programmi
- Ma molto spesso in Python è preferibile usare i metodi stringa, più veloci, con una sintassi più chiara che produce meno bug. In Python sono uno strumento utile e potente che va usato solo quando serve.

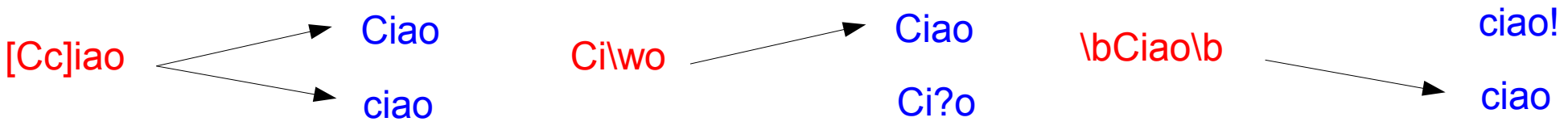
Hanno un mini-howto dedicato su:
`www.amk.ca/python/howto/`

redemo.py



Mia versione migliorata di *redemo*:
`www.fantascienza.net/leonardo/so/redemo2.pyw`

re – esempi (1)




```
>>> import re
>>> p = re.compile('selva')
>>> m = p.search('Nel mezzo del cammin di nostra vita')
>>> print m
None
>>> m = p.search('mi ritrovai per una selva oscura,')
>>> print m
<_sre.SRE_Match object at 0x8157078>
>>> m.span()
(20, 25)
>>> m.group()
'selva'
```

p.search trova e restituisce il PRIMO oggetto match per p dentro la stringa

re – esempi (2)

```
>>> p = re.compile('selva|tr') # la pipe equivale all'OR logico
>>> m = p.search('Nel mezzo del cammin di nostra vita')
>>> print m
<_sre.SRE_Match object at 0x81ce668>
>>> m.span()
(27, 29)
>>> m.group()
'tr'
>>> m = p.search('mi ritrovai per una selva oscura')
>>> m.span()
(5, 7)
>>> m.group()
'tr'
```

re – caratteri speciali

I caratteri speciali:

. ^ $ * + ? [] {} \ | () -

- [] individua una classe; - un range (es. [a-z])
- () individua un gruppo
- *, +, {}, ? servono per le ripetizioni dei match
- . qualunque carattere
- \d qualunque numero, equivalente a `[0-9]`
- \D e' il not di \d
- \s qualunque spazio
- \S e' il not di \s
- \w qual. car. alfanumerico, equivale a `[a-zA-Z0-9_]`
- \W e' il not di \w
- \b e' il bordo di una parola
- \B e' il not di \b
- | (pipe) e' l'OR logico
- ^ e' l'inizio della stringa (o della riga); $ la fine
- [^] dentro una classe e' la negazione (es. [^a] e' *not* a)
- \A e' l'inizio della stringa, \Z la fine
- \ e' il carattere di escape

re – ripetizioni

~= significa che si ha il match tra la RE e la stringa
!= significa che non c'e' il match

- `cat` ~= 'cat'
- `c.t` ~= 'cat', 'cAt', 'cCt' ma != 'ct'
- `c[Aa]t` ~= 'cat', 'cAt' ma != 'cCt'
- `ca*t` ~= 'ct', 'cat', 'caat', 'caaat'...
- `ca+t` ~= 'cat', 'caat', 'caaat'... ma != 'ct'
- `ca?t` ~= 'ct', 'cat' ma != 'caat', 'caaat'...
- `(ab)*` ~= '', 'ab', 'abab', 'abababab'...
- `(ab){1,2}` ~= 'ab', 'abab' ma != '', 'ababab'

re – esempi (3)

```
>>> p = re.compile('(ciao|bye)\s\1') # \1 è una backreference
>>> m = p.search('ciao ciao')
>>> print m
None
>>> p = re.compile(r'(ciao|bye)\s\1') # r = raw string
>>> m = p.search('ciao ciao')
>>> print m
<_sre.SRE_Match object at 0x81b80b8>
>>> p = re.compile(r'(ciao|bye)\s\1', re.IGNORECASE) # keywords
>>> m = p.search('ciAo Ciao')
>>> print m
<_sre.SRE_Match object at 0x81d1148>
>>> m.span()
(0, 9)
```

Esiste anche p.match() che equivale a p.search() ma richiede che il match sia *ALL'INIZIO* della stringa

re – esempi (4) - splitting

```
>>> p = re.compile(r'a\s')
>>> s = "voglio che venga divisa ad ogni a seguita da spazio"
>>> l = p.split(s)
>>> l
['voglio che veng', 'divis', 'ad ogni ', 'seguit', 'd', 'spazio']
>>> p1 = re.compile(r'(a\s)')
>>> l1 = p1.split(s)
>>> l1
['voglio che veng', 'a ', 'divis', 'a ', 'ad ogni ', 'a ',
'seguit', 'a ', 'd', 'a ', 'spazio']
```

re – esempi (5) – search & replace

```
>>> p = re.compile(r'(giallo|rosso)')
>>> s = "non so se preferire il giallo o il rosso."
>>> t = p.sub('giallorosso', s)
>>> t
'non so se preferire il giallorosso o il giallorosso.'
>>> t = p.sub('giallorosso', s, 1)
>>> t
'non so se preferire il giallorosso o il rosso.'
>>> p.findall(s)
['giallo', 'rosso']
```

re – esempi (6) – trovare tutti i match

```
>>> p = re.compile(r'ciao', re.IGNORECASE)
>>> s = "Ciao come stai? Ti ho detto ciao!!"
>>> def allmatches(p, astring):
...     result = []
...     def found(m): result.append(m)
...     p.sub(found, astring)
...     return result
...
>>> l = allmatches(p, s)
>>> l
[<_sre.SRE_Match object at 0x81d4f88>, <_sre.SRE_Match object at
0x81d4ff8>]
>>> for match in l:
...     print match.group(), match.span()
...
Ciao (0, 4)
ciao (28, 32)
```

Se il primo argomento di sub è una funzione, questa viene eseguita su ogni occorrenza *non-overlapping* del match

re – esempi (7) – scrmable (scramble)

Module *scrmable.py*

```
import re, random

def replacer(match_obj):
    word = match_obj.group()
    core = list(word[1:-1])
    random.shuffle(core)
    return word[0] + "".join(core) + word[-1]

def scrmable(text):
       return re.sub(r"[a-zA-Z]{4,}", replacer, text)
```

```
>>> import scrmable
>>> s = "This is just an example text, that shows"
>>> scrmable.scrmable(s)
'Tihs is just an emlxpae text, that sowhs'
```