Introduction

Simone Campagna

filosofia/1

Python è un linguaggio di programmazione con molti aspetti positivi:

- Grande semplicità d'uso
- Grande semplicità di apprendimento (assomiglia alla pseudocodifica)
- Grande leggibilità (c'è un solo modo per fare qualsiasi cosa)
- Grande portabilità

filosofia/2

- Per favore, non definiamolo un linguaggio di scripting! Anche se può essere utilizzato come tale, è molto, molto di più.
- È un linguaggio di altissimo livello, moderno, completo, con il quale è possibile realizzare software di notevole complessità.
- È un linguaggio interpretato, ma sarebbe più appropriato definirlo “linguaggio dinamico”.

filosofia/3

- Un linguaggio dinamico è un linguaggio di alto livello in cui vengono eseguiti run-time molti dei controlli che altri linguaggi eseguono in compilazione.
- In effetti python “compila” il sorgente in bytecode, che viene eseguito su una virtual machine (come Java).

filosofia/4

È un linguaggio multiparadigma:

- Imperative
- Object-oriented
- Functional
- Structural
- Aspect-oriented
- Design by contract (con una estensione)
- ...

python vs perl/1

- C'è un'antica disputa fra sostenitori di python e sostenitori di perl

python vs perl/2

Hanno ragione i sostenitori di python!

python vs perl/3

- La filosofia di perl è opposta a quella di python:
- PERL:
  - TMTOWTDI: There's more than one way to do it
  - TIMTOWTDIBSCINABTE: There's more than one way to do it, but sometimes consistency is not a bad thing either
- PYTHON:
  - There should be one—and preferably only one—obvious way to do it.

python vs perl/4

- PERL:
  - C'è una nuova esigenza? Definiamo un nuovo operatore!
- PYTHON:
  - Meno operatori e parole chiave ci sono, meglio è.

python vs perl/5

- The Zen of Python:
  - "beautiful", "explicit" and "simple"
  - To describe something as clever is NOT considered a compliment in the Python culture
  - There should be one—and preferably only one—obvious way to do it

The zen of python

>>> import this
The Zen of Python, by Tim Peters

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!

svantaggi

Python è spesso considerato un linguaggio *lento*. In buona misura questo è vero: è più lento di java, ad esempio.

Ma la velocità non è sempre il collo di bottiglia. Spesso la gestione della complessità è un problema più importante della velocità.

Vi sono comunque vari modi per rendere più veloci le parti “critiche” di un programma python.

performance

People are able to code complex algorithms in much less time by using a high-level language like Python (e.g., also C++). There can be a performance penalty in the most pure sense of the term.

optimization

"The best performance improvement is the transition from the nonworking to the working state."

--John Ousterhout

"Premature optimization is the root of all evil."

--Donald Knuth

"You can always optimize it later."

-- Unknown

l'interprete/1

- Python è un linguaggio interpretato
- L'interprete esegue una compilazione del sorgente in bytecode, che viene poi eseguito su una virtual machine, come in Java
- L'interprete è anche un eccellente “calcolatore”, da usare in interattivo!

```
>>> 2**1024
```

```
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602112011387987139335765878976881441662249284743063947412437776789342486548527630221960124609411945308295208500576883815068234246288147391311054082723716335051068458629823994724593847971630483535632962422413721 6L
```

l'interprete/2

- Quando usato in interattivo, l'interprete agisce in maniera leggermente diversa.
- Il prompt è “>>>”
- Se una espressione ha un valore, esso viene stampato automaticamente:

```
>>> 34 + 55 – 2

87
```

l'interprete/3

- Qualsiasi errore possa avvenire nel corso dell'esecuzione dell'interprete in interattivo, l'interprete sopravvive, anche in caso di SyntaxError:

```
>>> 5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> fact(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fact' is not defined
>>> @@@ ausdf9 !?;<j
  File "<stdin>", line 1
    @@@ ausdf9 !?;<j
     ^
SyntaxError: invalid syntax
>>>
```

print

Per stampare si usa *print*.

Può prendere un numero arbitrario di argomenti:

```
>>> print a, b, c
```

Normalmente stampa un newline al termine; non lo fa se la lista degli argomenti termina con una virgola:

```
>>> print a, b, c,
```

numeri interi

- In python <3.0 vi sono due tipi di interi:
  - *int* per interi fino a 2**63-1 (vedi *sys.maxint*)
  - *long* per interi di qualsiasi dimensione!
- Gli int vengono automaticamente trasformati in long quando necessario:

```
>>> print a, type(a)
9223372036854775807 <type 'int'>
>>> a += 1
>>> print a, type(a)
9223372036854775808 <type 'long'>
```

numeri floating point

- I numeri floating point sono rappresentati dal tipo float:

```
>>> 2.0**-1024
5.562684646268003e-309
>>>
```

test *(interactive)*

- Eseguire le seguenti operazioni:
  - 2**1024
  - 100/3
  - 100//3
  - 100.0/3
  - 100.0//3
  - 100%3
  - divmod(100, 3)

numeri complessi

- Esiste il tipo *complex*:

```
>>> z = 3.0 + 4.0j
>>> w = 3.0 - 4.0j
>>> z+w
(6+0j)
>>> z*w
(25+0j)
>>> type(z)
<type 'complex'>
>>> print z.real, z.imag, abs(z)
3.0 4.0 5.0
>>>
```

variabili/1

È possibile assegnare ad un qualsiasi oggetto un nome simbolico, che non necessita di essere dichiarato.

```
>>> a = 5
>>> b = 3.2
>>> c = a
>>> C = b   # non è “c”, è un'altra variabile
>>>
```

variabili/2

Anche se è un po' prematuro spiegarlo ora, in realtà i nomi simbolici *a*, *b, c* e *C* non sono affatto *variabili*.

Inoltre, anche il simbolo “=” non è affatto quello che sembra!

Per ora comunque possiamo proseguire “facendo finta” che questi simboli siano variabili e che “=” esegua un assegnamento o copia, come in C,

C++ o Fortran.

operatori

Sono disponibili i comuni operatori:

- + (somma, concatenazione)
- - (differenza)
- * (prodotto)
- / (divisione)
- // (divisione intera)
- % (modulo)
- ...

operatori binari

Come in C, agli operatori binari sono associati operatori di assegnamento:

- += (incremento)
- -= (decremento)
- …

Dunque,

```
>>> a = 10
>>> a += 4
>>> print a
14
>>>
```

stringhe/1

- Il tipo *str* è comunemente utilizzato per le stringhe. Esiste anche il tipo *unicode*.
- Possono essere create indifferentemente con apici singoli (`'alfa'`) o doppi (`“alfa”`)

```
>>> "alfa" + 'beta'
'alfabeta'
>>>
```

stringhe/2

- Le sequenze di apici tripli “”” o ''' possono essere utilizzate per stringhe che spaziano su più righe, o che contengono apici singoli o doppi (o tripli dell'altro tipo):

```
>>> a = """Questa stringa occupa due righe,
... e contiene apici 'singoli', "doppi" e
'''tripli'''"""
>>> print a
Questa stringa occupa due righe,
e contiene apici 'singoli', "doppi" e '''tripli'''
>>>
```

stringhe/3

- I caratteri di escape sono più o meno come in C:

```
>>> print "alfa\nbeta\tgamma"
alfa
beta    gamma
>>>
```

stringhe/4

- È possibile creare stringhe *raw* (senza sostituzione di *escape*) con costanti letterali come:

```
>>> print r"alfa\nbeta\tgamma"

alfa\nbeta\tgamma

>>>
```

- Questo risulta particolarmente utile per definire le *regular expression*

stringhe/5

È possibile compiere varie operazioni sulle stringhe:

```
>>> s = "Ciao, mondo!"
>>> print s.lower()
ciao, mondo!
>>> print s.upper()
CIAO, MONDO!
>>> print s.title()
Ciao, Mondo!
>>> print s.replace('o', 'x')
Ciax, mxndx!
>>>
```

stringhe/6

```
>>> print s.find('nd')
8
>>> print len(s)
12
>>> print “Ciao, mondo!”.upper()
CIAO, MONDO!
>>>
```

stringhe/7

```
>>> print “Ciao, mondo!”.toenglish()
Hello, world!
>>> print “Ciao, mondo!”.tofrench()
Bonjour, tout le monde!
>>>
```

D'accordo, scherzavo... queste non esistono!

stringhe/8

È possibile accedere ai singoli elementi della stringa, o a sottostringhe:

```
>>> hi_folk = "Hi, folk!"
>>> print hi_folk[0]
H
>>> print hi_folk[4]
f
>>> print hi_folk[-1]
!
>>> print hi_folk[2:]
, folk!
>>> print hi_folk[:3]
Hi,
>>> print hi_folk[2:5]
, f
>>>
```

stringhe/9

“Stranamente”, le stringhe non sono modificabili:

```
>>> hi_folk[1] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not
support item assignment
>>>
```

python program file

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


a = 10

print a
```

test *(nome.py)*

- Definire una stringa *nome* contenente il proprio nome, una stringa *cognome* contenente il proprio cognome;
- Stampare la lunghezza delle due stringhe;
- Concatenare le due stringhe formando *nome_cognome;*
- Stampare la lunghezza di nome_cognome.

contenitori

- Uno dei punti di forza di python è nei contenitori disponibili, che sono molto efficienti, comodi da usare, e versatili:
  - *tuple ()*
  - *list []*
  - *dict {}*
  - *set*

tuple/1

```
>>> a = (3, 4, 5)
>>> print a
(3, 4, 5)
>>> print a[1] # indici da 0 a len-1!
4
>>> b = 2, 3
>>> print b
(2, 3)
```

tuple/2

- Non sono necessariamente omogenee!

```
>>> a = (4, z, "alfa", b)
>>> a
(4, (3+4j), 'alfa', (2, 3))
```

tuple/3

- Possono anche stare alla sinistra dell'uguale:

```
>>> r, i = z.real, z.imag
>>> print z
3.0+4.0j
>>> print r
3.0
>>> print i
4.0
```

tuple/4

Definiscono altre operazioni:

```
>>> a = (1, 2, 3)
>>> b = (4, 5, 6)
>>> print a+b
(1, 2, 3, 4, 5, 6)
>>> print a*3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>>
```

tuple/5

La virgola è importante!

```
>>> x = 1
>>> print type(x), x
<type 'int'> 1
>>> y = 1,
>>> print type(y), y
<type 'tuple'> (1,)
>>>
```

tuple/6

Lo slicing consente di accedere a “porzioni” della tupla:

```
>>> a = (0, 1, 2, 3, 4)
>>> print a[1:3]  # dal secondo elemento (incluso)
                  # al quarto (escluso)
(1, 2)
>>> print a[:2]   # dal primo (incluso) al terzo (escluso)
(0, 1)
>>> print a[2:]   # dal terzo (incluso) all'ultimo (incluso)
(2, 3, 4)
>>> print a[2:] + a[:2]
(2, 3, 4, 0, 1)
>>>
```

test *(nome.py)*

Partendo dal file *nome.py*, definire una tupla contenente il proprio nome, il proprio cognome, e l'anno di nascita.

Costruire, a partire da essa, una nuova tupla contenente, nell'ordine, anno di nascita, nome, cognome).

liste/1

Le liste sono “tuple modificabili”:

```
>>> l = [1, 2, 3]
>>> print l
[1, 2, 3]
>>> l.append(4)
>>> print l
[1, 2, 3, 4]
>>> l.insert(2, "XYZ")
>>> print l
[1, 2, 'XYZ', 3, 4]
>>> print len(l)
5
```

liste/2

```
>>> print l[0], l[-1]
1 4
>>> print l[:2]
[1, 2]
>>> print l[-4:]
[2, 'XYZ', 3, 4]
>>> l[1] = 3, 2, 1, 0
>>> print l
[1, (3, 2, 1, 0), 'XYZ', 3, 4]
>>>
```

liste/3

```
>>> l = [3, 5, 7, 9, 11, 13]
>>> l.append(2)
>>> l.remove(9)
>>> print l
[3, 5, 7, 11, 13, 2]
>>> l.sort()
>>> print l
[2, 3, 5, 7, 11, 13]
>>> l.reverse()
>>> print l
[13, 11, 7, 5, 3, 2]
>>>
```

liste/4

Le liste possono essere utilizzate come stack:

```
>>> print l
[13, 11, 7, 5, 3, 2]
>>> l.pop()
2
>>> l.pop()
3
>>> l.pop()
5
>>> l.pop()
7
>>> print l
[13, 11]
>>>
```

liste/5

```
>>> l *= 3
>>> print l
[13, 11, 13, 11, 13, 11]
>>> l[1:4] = ['a', 'b', 'c']
>>> print l
[13, 'a', 'b', 'c', 13, 11]
>>> print l.count(13)
2
>>> print l.count(7)
0
>>> print l.index('c')
3
```

liste/6

```
>>> l.extend( (2, 3, 5) )
>>> print l
[13, 'a', 'b', 'c', 13, 11, 2, 3, 5]
>>> del l[:-4]
>>> print l
[11, 2, 3, 5]
>>>
```

range

- Range è una funzione per generare liste di interi:

```
>>> print range(3)
[0, 1, 2]
>>> print range(3, 7)
[3, 4, 5, 6]
>>> print range(3, 10, 2)
[3, 5, 7, 9]
```

test *(books.py)*

Definite una lista contenente alcuni nomi di libri che avete letto.

Aggiungete qualche altro elemento alla lista.

Stampate il numero di elementi della lista.

Riordinate la lista.

extended slices/1

La sintassi dello slicing accetta ora un parametro *stride*, sempre separato da :

```
>>> l = range(20)
>>> l[1:18]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> l[1:18:3]
[1, 4, 7, 10, 13, 16]
>>> l[1::3]
[1, 4, 7, 10, 13, 16, 19]
>>> l[:5:3]
[0, 3]
>>> l[::3]
[0, 3, 6, 9, 12, 15, 18]
>>>
```

extended slices/2

Lo *stride* può anche essere negativo

```
>>> l[3:18:3]
[3, 6, 9, 12, 15]
>>> l[18:3:-3]
[18, 15, 12, 9, 6]
>>> l[17:2:-3]
[17, 14, 11, 8, 5]
>>>  l[::-1]
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7,
6, 5, 4, 3, 2, 1, 0]
>>>
```

extended slices/3

Quando si assegna con lo slicing, le extended slices sono meno flessibili delle regular slices, infatti la lunghezza degli operandi deve essere identica:

```
>>> l = range(5)
>>> r = ['a', 'b', 'c', 'd']
>>> r[1:3] = l[:-1]  # len(r[1:3]) != len(l[:-1]), ok
>>> r
['a', 0, 1, 2, 3, 'd']
>>> r[::3] = l[::2]  # len(r[::2]) != len(l[::2]), KO!!!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
>>> r[::2] = l[::2]  # len(r[::2]) == len(l[::2]), ok
>>> r
[0, 0, 2, 2, 4, 'd']
>>>
```

extended slices/4

Quando si eliminano elementi con lo slicing, non ci sono problemi:

```
>>> l = range(11)
>>> del l[::3]
>>> l
[1, 2, 4, 5, 7, 8, 10]
>>> del l[::-2]
>>> l
[2, 5, 8]
>>>
```

set/1

- I set definiscono insiemi di elementi senza ripetizione. Non sono ordinati (nel senso che sono ordinati automaticamente al fine di rendere veloce la ricerca di elementi).

```
>>> s = set()
>>> s.add(2)
>>> s.add(3)
>>> s.add(2)
>>> s.add(4)
>>> s
set([2, 3, 4])
>>> u = set(['alfa', 2, 3.5])
>>> print u
set([3.5, 2, 'alfa'])
>
```

set/2

```
>>> print s
set([2, 3, 4])
>>> l=[1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
>>> t = set(l)
>>> print t
set([1, 2, 3, 4])
>>> s.intersection(t)
set([2, 3, 4])
>>> s.difference(t)
set([])
>>> t.difference(s)
set([1])
>>>
```

set/3

```
>>> s.symmetric_difference(t)
set([1])
>>> s.union(t)
set([1, 2, 3, 4])
>>> s.discard(3)
>>> print s
set([2, 4])
>>> s.clear()
>>> print s
set([])
>>>
```

frozenset/4

Per analogia con le tuple, che possono essere considerate “liste congelate”, esistono i frozenset:

```
>>> ft = frozenset(t)
>>> ft
frozenset([1, 2, 3, 4])
>>> ft.add(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute
'add'
>>>
```

test *(primes_1.py)*

Avendo a disposizione solo queste strutture dati:

```
>>> all = set(range(1, 20))
>>> primes = set([2, 3, 5, 7, 11, 13, 17, 19])
>>> even = set([2, 4, 6, 8, 10, 12, 14, 16, 18])
```

stampare l'insieme dei numeri dispari, l'insieme dei numeri dispari primi, e l'insieme dei numeri dispari non primi.

dizionari/1

- I dizionari implementano array associativi; associano ad una chiave arbitraria un valore arbitrario:

```
>>> numero_zampe = {'gatto': 4, 'oca': 2,
'formica': 6, 'ragno': 8}

>>> print numero_zampe['formica']

6

>>> print numero_zampe

{'oca': 2, 'ragno': 8, 'formica': 6, 'gatto':
4}

>>>
```

dizionari/2

- Non sono necessariamente omogenei:

```
>>> d = {}
>>> d['alfa'] = 3
>>> d[2.5] = 'xyz'
>>> d[3+4j] = [3, 4, 5]
>>> d[(1,2,3)] = { 'x': 2, 'y': 3, 'z': 1 }
>>> print d
{2.5: 'xyz', (1, 2, 3): {'y': 3, 'x': 2, 'z':
1}, 'alfa': 3, (3+4j): [3, 4, 5]}
>>>
```

dizionari/3

```
>>> print d.keys()
[2.5, (1, 2, 3), 'alfa', (3+4j)]
>>> print d.values()
['xyz', {'y': 3, 'x': 2, 'z': 1}, 3, [3, 4, 5]]
>>> print d.items()
[(2.5, 'xyz'), ((1, 2, 3), {'y': 3, 'x': 2, 'z': 1}),
('alfa', 3), ((3+4j), [3, 4, 5])]
>>> d.has_key('alfa')
True
>>> d.has_key('beta')
False
>>>
```

dizionari/4

```
>>> d.get('alfa', -1999)
3
>>> d.get('beta', -1999)
-1999
>>> print d
{2.5: 'xyz', (1, 2, 3): {'y': 3, 'x': 2, 'z': 1}, 'alfa': 3, (3+4j):
[3, 4, 5]}
>>> d.setdefault('alfa', -1999)
3
>>> d.setdefault('beta', -1999)
-1999
>>> print d
{2.5: 'xyz', 'beta': -1999, (1, 2, 3): {'y': 3, 'x': 2, 'z': 1},
'alfa': 3, (3+4j): [3, 4, 5]}
>>>
```

dizionari/5

Possono essere usati come stack:

```
>>> d.pop('alfa', -5)
3
>>> d.pop('gamma', -5)
-5
>>> d.popitem()
(2.5, 'xyz')
>>> d.popitem()
('beta', -1999)
>>> print d
{(1, 2, 3): {'y': 3, 'x': 2, 'z': 1}, (3+4j): [3, 4, 5]}
>>>
```

dizionari/6

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = {1: 1.0, 2: 2.0}
>>> d1.update(d2)
>>> print d1
{'a': 1, 1: 1.0, 'c': 3, 'b': 2, 'd': 4, 2:
2.0}
>>> d1.clear()
>>> print d1
{}
>>>
```

test *(books.py)*

Definite un dizionario che metta in relazione alcuni dei libri che avete letto con il nome del relativo autore.

Stampare i nomi di tutti gli autori (senza ripetizioni).

costrutti per il controllo del flusso

- Python ha pochi costrutti per il controllo del flusso, secondo la filosofia della massima semplicità.

indentazione

- In python l'indentazione è sintattica, vale a dire, determina l'annidamento degli statement.
- Questo è parte integrante della filosofia di python: siccome indentare è un bene, e siccome un programma senza indentazione è un male, perché non obbligare ad indentare?
- Diventa quindi inutile l'uso delle parentesi graffe per racchiudere blocchi, come in C/C++, o di statement di chiusura, come l'END DO del Fortran.

if-elif-else

```
>>> if a == b:
...     print a
... elif a > b:
...     print b
... else:
...     print a
...
```

for/1

- Il ciclo for consente di iterare su “oggetti iterabili”, come liste, tuple, set, dizionari, …

```
>>> for i in range(3):
...     print i
...
0
1
2
>>>
```

for/2

```
>>> t = ('a', 'b', 10, 5.5)
>>> for i in t:
...     print i
...
a
b
10
5.5
>>>
```

for/3

```
>>> d = {'a': 0, 'b': 1, 'c': 2}
>>> for key in d.keys():
...     print key
...
a
c
b
>>>
```

for/4

```
>>> for val in d.values():
...     print val
...
0
2
1
>>>
```

for/5

```
>>> for key, val in d.items():
...     print key, '=', val
...
a = 0
c = 2
b = 1
>>>
```

for/6

```
>>> for key in d:
...     print key
...
a
c
b
>>>
```

for/7

```
>>> for key in ('a', 'd', 'c'):
...     print d.get(key, None)
...
0
None
2
>>>
```

for/8

```
>>> s = set(range(0, 10, 2)+range(0, 10, 3))
>>> print s
set([0, 2, 3, 4, 6, 8, 9])
>>> for i in s:
...   print i
...
0
2
3
4
6
8
9
>>>
```

test *(books.py)*

Partendo dal dizionario libri : autori, scrivere il codice per stampare, per ciascun autore, il numero di libri presenti nel dizionario.

while/1

- Il ciclo while è il generico ciclo con una condizione:

```
>>> i = 0
>>> while i < 4:
...     print i
...     i += 1
...
0
1
2
3
>>>
```

while/2

- Il ciclo precedente non è tuttavia un buon esempio, molto meglio questo:

```
>>> for i in range(4):
...     print i
...
0
1
2
3
>>>
```

break

Break permette di uscire dal loop:

```
>>> for i in range(10000):
...     print i
...     if i%3 == 2:
...         break
...
0
1
2
>>>
```

continue

Continue permette di passare all'iterazione successiva:

```
>>> for i in range(4):
...     if i == 1:
...         continue
...     print i
...
0
2
3
>>>
```

loops: else clause

- I loop (for e while) possono avere una clausola *else*, che viene eseguita solo se non si esce dal loop con un *break*; ovvero, se il loop completa naturalmente:

```
>>> for i in range(10):
...   if i > 3: break
...   print i
... else:
...   print "finito!"
...
0
1
2
3
>>>
```

```
>>> for i in range(2):
...   if i > 3: break
...   print i
... else:
...   print "finito!"
...
0
1
finito!
>>>
```

switch

Non esiste un costrutto switch, basta una serie di blocchi *if-elif-else.*

operatori

Vi sono vari operatori:

- Operatori di comparazione: *==, !=, <, <=, >, >=, is, in*
- Operatori logici: *and, or, not*

is

L'espressione *a == b* restituisce *True* se il valore di *a* è identico al valore di *b*.

L'espressione *a is b* restituisce *True* se *a* e *b* si riferiscono allo stesso oggetto fisico

```
>>> l1 = [1, 3, 8]
>>> l2 = [1, 3, 8]
>>> l3 = l1
>>> print l1 == l2, l1 is l2
True False
>>> print l1 == l3, l1 is l3
True True
>>> print l2 == l3, l2 is l3
True False
>>>
```

in

L'espressione *a in lst* restituisce *True* se l'oggetto *a* è contenuto in *lst.*

```
>>> print 2 in l1
False
>>> print 3 in l1
True
>>>
```

None/1

- None è un oggetto particolare del linguaggio, che viene utilizzato per indicare l'assenza di un valore o un valore indefinito

```
>>> a = None
>>> print a
None
>>>
```

bool/1

- Il tipo bool viene utilizzato per i valori logici di verità/falsità
- Un oggetto bool può assumere uno di questi valori:
  - `True`
  - `False`

bool/2

```
>>> a = 1 > 5
>>> print a
False
>>> b = 1 <= 5
>>> print b
True
>>>
```

conversioni a bool/1

- I tipi predefiniti possono essere convertiti a bool, nel senso che possono essere usati in espressioni condizionali.
  - Un *int* uguale a 0 equivale a *False*, altrimenti *True*
  - Un *float* uguale a 0.0 equivale a *False*, altrimenti *True*
  - Una stringa vuota *“”* equivale a *False*, altrimenti *True*
  - Un contenitore vuoto([], (), set(), {}, ...) equivale a *False*, altrimenti *True*
  - *None* equivale a *False*

test *(divisors.py)*

- Scrivete un codice per stampare tutti i divisori primi del numero 2009

funzioni/1

- Le funzioni si dichiarano con *def*, seguito dal nome della funzione e dalla lista dei parametri.
- Come al solito, non si dichiara nulla, né il tipo del valore di ritorno, né il tipo dei parametri: *duck typing!*

funzioni/2

```
>>> def sum(a, b):
...   return a+b
...
>>>
>>> print sum(3, 4)
7
>>> print sum("Ciao, ", "mondo!")
Ciao, mondo!
>>> print sum(3.2, 9.1)
12.3
>>> print sum(6, "ccc")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

funzioni/3

```
>>> def factorial(n):
...   if n < 2:
...     return 1
...   else:
...     return n*factorial(n-1)
...
>>> for i in (0, 1, 5, 10, 200):
...   print factorial(i)
...
1
1
120
3628800
788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579568660226031904170324062351700858796178922227896237038973747200000000000000000000000000000000000000000000000000
>>>
```

funzioni/4

Qualsiasi funzione ha un valore di ritorno. Per default, questo valore è None. Se si vuole dare un valore di ritorno specifico, ad esempio 4, ad una funzione, basta aggiungere lo statement *return 4*.

Uno statement *return* senza espressioni alla destra equivale a *return None*.

Se una funzione termina senza aver incontrato alcun *return*, esegue implicitamente un *return None.*

test *(divisors.py)*

- Scrivere una funzione che determini tutti i divisori di un numero intero arbitrario

test *(divisors.py -> primes.py)*

- Scrivere una funzione che verifichi se un numero è primo (non importa che sia efficiente!)

passaggio di parametri/1

- Gli argomenti di una funzione possono essere passati per posizione o per nome:

```
>>> def count(lst, val):
...     c = 0
...     for el in lst:
...         if el == val: c += 1
...     return c
...
>>> print count([1,2,1,3,2,4], 2)
2
>>> print count(val=2, lst=[1,2,1,3,2,4])
2
```

passaggio di parametri/2

- Dopo aver passato almeno un argomento per nome, non è più possibile passarli per posizione:

```
>>> print count(val=2, [1,2,1,3,2,4,1])
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword
arg
>>>
```

argomenti di default/1

- Gli argomenti di una funzione possono avere valori di default:

```
>>> def count(lst, val=1):
...     c = 0
...     for el in lst:
...         if el == val: c += 1
...     return c
...
>>> print count([1,2,1,3,2,4,1], 2)
2
>>> print count([1,2,1,3,2,4,1])
3
>>>
```

argomenti di default/2

- Se un argomento accetta un valore di default, anche tutti i successivi argomenti devono avere un valore di default:

```
>>> def f(a, b=0, c):
...     print a+b
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows
default argument
>>>
```

arbitrary argument list/1

Una funzione può avere argomenti posizionali opzionali; questi argomenti vengono inseriti in una tupla:

```
>>> def f(a, *l):
...   print a
...   print l
...
>>> f("a")
a
()
>>> f("a", 2)
a
(2,)
>>> f("a", 2, 5, 'y')
a
(2, 5, 'y')
>>>
```

arbitrary argument list/2

```
>>> def sum(a, *l):
...   for i in l:
...     a += i
...   return a
...
>>> print sum(10)
10
>>> print sum(10, 1, 100)
111
>>> print sum(“a”, “bc”, “d”)
abcd
>>>
>>> print sum([1], [], [2, 3], [4, 5, 6], range(7, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

arbitrary keyword arguments

Gli argomenti opzionali possono essere previsti anche per nome; in tal caso, vengono mantenuti in un dizionario:

```
>>> def g(a, **kw):
...   print a
...   print kw
...
>>> g(5, y=9, z=5, x=1)
5
{'y': 9, 'x': 1, 'z': 5}
>>> g(5, y=9, z=5, x=1, a=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: g() got multiple values for keyword argument 'a'
>>>
```

forma generale di una funzione

- Riassumendo, la forma generale di una funzione è:
- >>> def f(a, b, c=0, *l, **kw):
- ... print a, b, c, l, kw
- ...
- >>> f(1, 2)
- 1 2 0 () {}
- >>> f(1, 2, 3)
- 1 2 3 () {}
- >>> f(1, 2, 3, 4, 5, 6, 7)
- 1 2 3 (4, 5, 6, 7) {}
- >>> f(1, 2, 3, 4, 5, 6, 7, x=0, y=1, alfa=2)
- 1 2 3 (4, 5, 6, 7) {'y': 1, 'x': 0, 'alfa': 2}
- >>>

unpacking argument list

A volte può capitare di avere, in una lista, il valore degli argomenti posizionali di una funzione:

```
>>> def f(a, b, c):
...     return a*b-c
...
>>> args = [3, 5, -2]
>>> print f(*args) ### => f(3, 5, -2)
17
>>>
```

unpacking keyword arguments

A volte può capitare di avere, in una tupla, una lista di argomenti da passare per nome ad una funzione:

```
>>> def f(x, y, z):
...     return x**2 + y**2 + z**2
...
>>> args = {'x': 3, 'y': 4, 'z': 5}
>>> print f(**args) ### => f(x=3, y=4, z=5)
50
>>>
```

doc string/1

È possibile associare ad una funzione una *doc string*, una stringa arbitraria di documentazione per la funzione. La stringa diventa parte integrante della funzione; se il nome della funzione è *fun*, *fun.__doc__* è la sua *doc string*.

Per default, la doc string vale *None*.

doc string/2

```
>>> def sum(a, b, *l):
...    """Somma due o piu' numeri; ad esempio,
...       >>> print sum(2, 4, 10, 20, 30)
...       66
...       >>>"""
...    r = a + b
...    for el in l:
...      r += el
...    return r
...
>>> print sum.__doc__
Somma due o piu' numeri; ad esempio,
     >>> print sum(2, 4, 10, 20, 30)
     66
     >>>
>>>
```

doc string/3

```
>>> print range.__doc__ # le funzioni built-in hanno una doc!
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of
integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!)
defaults to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is
omitted!
These are exactly the valid indices for a list of 4 elements.
>>>
```

pass

- Pass è una generica istruzione che non fa niente, e può stare in qualsiasi posizione del codice. È utile perché in alcune situazioni è necessario uno statement per ragioni sintattiche. Ad esempio, il corpo di una funzione non può essere omesso. Se si vuole una funzione vuota, si fa così:

```
>>> def f(a, b, c, d=0):
...     pass
...
>>>
```

tutto è oggetto

Tutto è un oggetto, anche una funzione. Pertanto, anche alle funzioni possiamo associare nomi simbolici:

```
>>> def sum(a, b):
...     return a+b
...
>>> somma = sum
>>> print somma(10, 12)
22
>>>
```

funzioni come argomento di funzioni

Poiché si può assegnare un nome simbolico ad una funzione, si può anche passare una funzione come argomento di un'altra funzione:

```
>>> def sum(a, b): return a+b
...
>>> def sub(a, b): return a-b
...
>>> def g(x, f): return f(x, x+1)
...
>>> g(1, sum)
3
>>> g(1, sub)
-1
>>>
```

sort/1

Un buon esempio di funzione che riceve una funzione come argomento è costituito dall'ordinamento delle liste.

Il metodo *sort* di *list*, che abbiamo visto, può ricevere un parametro opzionale di tipo funzione. Se questo parametro viene fornito, deve essere una funzione di due argomenti, che restituisca

- -1, se il primo argomento è strettamente minore del secondo;
- 0, se i due argomenti sono uguali
- +1, se il primo argomento è strettamente maggiore del secondo.

La funzione builtin *cmp* ha proprio questo effetto:

$$\text{cmp}(a, b) = \begin{cases} -1 & \textit{se } a < b \\ 0 & \textit{se } a == b \\ +1 & \textit{se } a > b \end{cases}$$

sort/2

```
>>> l = [(1, 2), (-1, 9), (10, 0), (2, -3)]
>>>
>>> def compare(x, y): return cmp(x[1], y[1])
...
>>> l.sort(compare)
>>> print l
[(2, -3), (10, 0), (1, 2), (-1, 9)]
>>> def compare2(x, y): return cmp(x[0]-x[1], y[0]-y[1])
...
>>> l.sort(compare2)
>>> print l
[(-1, 9), (1, 2), (2, -3), (10, 0)]
>>>
```

lambda functions/1

Nell'ambito del paradigma di programmazione funzionale, rispetto al quale python offre un discreto supporto, è spesso necessario definire piccole funzioni da passare come argomento ad altre funzioni.

Le lambda function sono funzioni “anonime” fatte di una sola riga, semplici e compatte.

Hanno alcune limitazioni: non possono far altro che restituire il risultato di un'unica espressione, per quanto complessa. Non possono eseguire statement (come *print*, o come *a = 10*).

lambda functions/2

```
>>> def compare(a, b):
...    return cmp(a[1], b[1])
...
>>> print compare((10, 1), (0, 2))
True
>>> print compare((10, 10), (0, 2))
False
>>> lambda_compare = lambda a, b: cmp(a[1], b[1])
>>> print lambda_compare((10, 1), (0, 2))
True
>>> print lambda_compare((10, 10), (0, 2))
False
```

lambda functions/3

```
>>> l = [(1, 2), (-1, 9), (10, 0), (2, -3)]
>>> l.sort(lambda a, b: cmp(a[1], b[1]))
>>> print l
[(1, 2), (-1, 9), (10, 0), (2, -3)]
>>>
>>> l.sort(lambda x, y: cmp(x[0]-x[1], y[0]-y[1]))
>>> print l
[(-1, 9), (1, 2), (2, -3), (10, 0)]
```

programmazione funzionale/1

Il paradigma di programmazione funzionale si basa principalmente sull'applicazione di funzioni a sequenze.

Una sequenza è un qualsiasi oggetto iterabile.

programmazione funzionale/2

Il paradigma di programmazione funzionale si basa principalmente su tre funzioni built-in:

- *filter*: applica un filtro ad una sequenza
- *map*: applica una funzione a tutti gli elementi di una sequenza
- *reduce*: esegue una operazione di riduzione su tutti gli elementi di una sequenza

filter

```
>>> l = range(30)
>>> print l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29]
>>> print filter(lambda x: x%5 == 0, l)
[0, 5, 10, 15, 20, 25]
>>> print filter(lambda x: x < 3, l)
[0, 1, 2]
>>>
```

map

```
>>> l = range(10)
>>> print map(lambda x: x**2, l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print map(lambda x: x+100, l)
[100, 101, 102, 103, 104, 105, 106,
107, 108, 109]
>>>
```

reduce/1

```
>>> l = range(10)
>>> print reduce(lambda x, y: x+y,
l)
45
>>> print reduce(lambda x, y: x+y,
l, 1000)
1045
>>>
```

reduce/2

Reduce usa il primo argomento della lista come inizializzazione, se non viene dato un inizializzatore (terzo argomento). Talvolta questo non va bene:

```
>>> reduce(lambda x, y: x+y, [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
>>> reduce(lambda x, y: x+y, [], 0)
0
>>>
```

sum, min, max

Sono varianti di reduce per ottenere la somma, il minimo ed il massimo degli elementi:

```
>>> l = range(10)
>>> print sum(l)
45
>>> print min(l)
0
>>> print max(l)
9
>>>
```

test *(books.py)*

- A partire dal dizionario libri->autori, costruire una lista di tutti i libri il cui titolo non contiene il carattere *“a”*.
- A partire dal dizionario libri->autori, costruire una lista di tutti i libri ordinati per nome dell'autore

list comprehension/1

Viene detta list comprehension la possibilità di eseguire in un unica espressione operazioni complesse sulle liste.

```
>>> l = range(10)
>>>
>>> print [i+1 for i in l]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> print [i**2 for i in l if i%3==0]
[0, 9, 36, 81]
>>> print [[0 for i in range(10)] for i in range(7)]
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0,
0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
>>>
```

list comprehension/2

```
>>> l1 = [1, 2, 4]
>>> l2 = ['a', 'b']
>>> l3 = [9, 8, 7]
>>> [(e1,e2,e3) for e1 in l1 for e2 in l2
for e3 in l3]
[(1, 'a', 9), (1, 'a', 8), (1, 'a', 7), (1,
'b', 9), (1, 'b', 8), (1, 'b', 7), (2, 'a',
9), (2, 'a', 8), (2, 'a', 7), (2, 'b', 9),
(2, 'b', 8), (2, 'b', 7), (4, 'a', 9), (4,
'a', 8), (4, 'a', 7), (4, 'b', 9), (4, 'b',
8), (4, 'b', 7)]
```

test *(books.py)*

- A partire dal dizionario libri->autori, costruire una lista di tutti i libri il cui titolo non contiene il carattere *“a”* usando la list comprehension.

str e repr/1

Vi sono due funzioni per convertire un oggetto in stringa: *str* e *repr*.

Per capire come funzionano, bisogna conoscere un po' la programmazione object-oriented.

La chiamata *str(x)* equivale a `x.__str__()` se l'oggetto *x* ha un metodo `__str__`, altrimenti equivale a `x.__repr__()`. Tutti gli oggetti hanno automaticamente un metodo `__repr__`.

str e repr/2

Quando viene stampato un oggetto, ad esempio

```
>>> print x
```

viene in realtà eseguito

```
>>> print str(x)
```

e dunque viene chiamato il metodo `__str__` se x lo possiede, `__repr__` in alternativa.

str e repr/3

Possiamo forzare l'uso di *str* o *repr*; con le stringhe è possibile vedere la differenza:

```
>>> a = "Ciao, mondo!"
>>> print a
Ciao, mondo!
>>> print str(a)
Ciao, mondo!
>>> print repr(a)
'Ciao, mondo!'
>>>
```

str e repr/4

Quando viene stampato un contenitore standard (liste, tuple, dizionari o set), gli elementi contenuti vengono sempre stampati attraverso *repr:*

```
>>> print a
Ciao, mondo!          # str(a)
>>> print [a]
['Ciao, mondo!']      # repr(a)
>>>
```

str e repr/5

In pratica, *repr* dovrebbe restituire una rappresentazione dell'oggetto più vicina a quella “interna”. Per i tipi predefiniti, l'output di repr equivale alla costante letterale che consente di definire un oggetto equivalente:

```
>>> a = 'ciao'
>>> print str(a)
ciao
>>> print repr(a)
'ciao'
>>>
```

string formatting

Python supporta due modalità di formattazione di stringhe.

“old-style” string formatting/1

La modalità più vecchia è simile a quella utilizzata per la funzione C printf.

```
>>> a = 10
>>> print "Il valore di a  e': %d" % a
Il valore di a  e': 10
>>>
>>> b = 0.2
>>> print "a==<%3d>, b==<%10.4f>" % (a, b)
a==< 10>, b==<    0.2000>
>>>
```

“old-style” string formatting/2

L'associazione fra specificatori di formato e parametri può avvenire anche per nome:

```
>>> dct = {
...     'a': 10,
...     'b': 20,
...     'c': 30,
... }
>>>
>>> print "%(a)d + %(b)d + %(c)d" % dct
10 + 20 + 30
>>>
```

“old-style” string formatting/3

A differenza della printf C, lo specificatore di formato *%s* stampa qualsiasi cosa:

```
>>> a, b, c = 10, "ciao", 5.4
>>> print "%s, %s, %s" % (a, b, c)
10, ciao, 5.4
>>>
```

In pratica, viene forzata la chiamata a str.

“new-style” string formatting/1

Si usa il metodo *format* delle stringhe.

```
>>> print "My name is {0} {1}".format("Simone",
"Campagna")
My name is Simone Campagna
>>> print "My name is {1} {0}".format("Simone",
"Campagna")
My name is Campagna Simone
>>> print "My Name is {name}
{surname}".format(surname="Campagna", name="Simone")
My Name is Simone Campagna
>>>
```

“new-style” string formatting/2

```
>>> l = ["zero", "one", "two", "three", "four",
"five", "six", "seven", "eight", "nine", "ten"]
>>> print "{array[2]} + {array[1]} + {array[0]}
+ {array[4]} = {array[7]}".format(array=l)
two + one + zero + four = seven


>>> abc = {'a': 0, 'b':1, 'c':2}
>>> print "{d[a]}, {d[c]}".format(d=abc)
0, 2
>>>
```

test *(books.py)*

- Stampare tutto il contenuto del dizionario

  libri->autori con un formato simile a questo:

  Umberto Eco ha scritto il libro 'Il nome della rosa'.

classi (breve introduzione)

Le classi sono lo strumento per definire nuovi tipi. Immaginiamo di non avere *complex*:

```
class Complex(object):
  def __init__(self, r=0.0, i=0.0):
    self.re = r
    self.im = i
  def __mul__(self, other):
    return Complex(self.re*other.re-self.im*other.im,
self.re*other.im+self.im*other.re)
  def __imul__(self, other):
    self.re = self.re*other.re-self.im*other.im
    self.im = self.re*other.im+self.im*other.re
    return self
  def __rmul__(self, other):
    return self.__mul__(other)
  def __str__(self):
    return "(%s+%si)" % (self.re, self.im)
```

istanze

Abbiamo detto che tutto è oggetto. Ora possiamo dire che un oggetto è una istanza di una classe.

4 è una istanza della classe *int*.

metodi

Le classi hanno metodi, ovvero funzioni che si applicano su istanze di quella classe.

Il primo argomento di qualsiasi metodo è l'istanza su cui si applica; convenzionalmente lo si chiama *self*.

```
>>> class A(object):
...   def f(self, i):
...     print "A::", i
...
>>> a = A()
>>> a.f(10)
A:: 10
>>>
```

attributi/1

Ogni istanza (ogni oggetto) ha vari attributi. Anche i metodi sono attributi, di tipo funzione. Possono essere aggiunti in qualsiasi momento:

```
>>> print a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
>>> a.x = 10.2
>>> print a.x
10.2
>>>
```

attributi/2

Normalmente comunque gli attributi sono definiti attraverso i metodi della classe. In particolare, attraverso il metodo __init__, il costruttore della classe, che viene chiamato quando deve essere costruita una istanza della classe.

```
class Complex(object):
  def __init__(self, r=0.0, i=0.0):
    self.re = r
    self.im = i
```

attributi speciali

Tutte le classi hanno l'attributo __init__, perché le istanze devono poter essere costruite. Analogamente, vi sarà un __del__, il distruttore, automaticamente chiamato quando l'oggetto deve essere distrutto.

Per varie funzionalità vi sono metodi dal nome speciale: ad esempio, __len__, __str__, __repr__, __sort__, __get__, __setitem__, __delslice__, __add__, __iadd__, __radd__, __in__, ...

attributi di classe

Le classi possono avere attributi:

```
>>> class ALFA(object):
...   A = 10
...   def __init__(self):
...     self.x = 3
...
>>> a = ALFA()
>>> print a.A
10
>>> print a.x
3
>>> print a.__class__.A
10
>>> print ALFA.A
10
>>>
```

ereditarietà/1

Una classe può ereditare da un'altra classe; in tal caso, ne eredita tutto il contenuto. I metodi possono eventualmente essere ridefiniti.

ereditarietà/2

Una classe può ereditare da un'altra classe; in tal caso, ne eredita tutto il contenuto. I metodi possono eventualmente essere ridefiniti.

```
>>> class BETA(ALFA):
...   def __init__(self):
...     ALFA.__init__(self)
...     self.y = 5
...
>>> b = BETA()
>>> print b.A
10
>>> print b.x
3
>>> print b.y
5
>>>
```

oggetti iterabili/1

Abbiamo visto vari contenitori; tutti questi sono oggetti iterabili, ovvero, è possibile iterare sugli elementi di questi oggetti (ad esempio, con un ciclo *for*).

Vi sono altri oggetti su cui è possibile iterare, anche se a rigore non contengono alcunché.

oggetti iterabili/2

Consideriamo la funzione *range*; essa fornisce una lista.

```
>>> def sumn(N):
...     s = 0
...     for i in range(N+1): s+=i
...     return s
...
>>> print sumn(10)
55
>>> print sumn(10000)
50005000
>>>
```

Questo è inefficiente, perché viene creata una lista di N elementi, anche se ne viene sempre utilizzato uno alla volta.

oggetti iterabili/3

Sarebbe preferibile poter “generare” gli elementi della sequenza ad uno ad uno, senza doverli per forza memorizzare tutti assieme.

```
>>> def range_iter(N):
...   i=0
...   while i<N:
...     yield i
...     i += 1
...
>>> for i in range_iter(3):
...   print i
...
0
1
2
>>>
```

oggetti iterabili/4

```
>>> def sumn_iter(n):
...     return sum(range_iter(n+1))
...
>>> print sumn_iter(10)
55
>>> print sumn_iter(10000)
50005000
>>>
```

generatori ed iteratori/1

La funzione range_iter è un generatore. Essa produce un oggetto iterabile. Il trucco è sostituire *return* con *yield*; entrambi restituiscono al chiamante l'espressione a destra.

La differenza è che quando viene chiamato *yield*, la funzione “rimane sullo stack”, e la sua esecuzione può essere ripresa dal punto in cui si era interrotta.

generatori ed iteratori/2

Per essere più precisi (ma avremo bisogno di un po' di programmazione object-oriented), quando la funzione generatore viene chiamata, essa produce in output un iteratore. Un iteratore è un oggetto che possiede un metodo *next* (o *__next__*), il quale restituisce il successivo elemento della sequenza.

Generalmente un iteratore mantiene un qualche stato (nel nostro caso sarebbe il valore di *i*).

generatori ed iteratori/3

Gli iteratori possono facilmente essere costruiti a mano, con un po' di programmazione object-oriented. Non è difficile.

I generatori sono funzioni che consentono di evitare di scrivere manualmente un iteratore.

Quando una funzione usa *yield*, diventa un generatore. L'uso di *return* è allora proibito in quella funzione.

xrange

La funzione built-in *xrange* possiede la stessa identica interfaccia di range, ma non produce una lista, bensì un iteratore, in maniera simile alla nostra range_iter.

Con *xrange* è possibile iterare su una sequenza di 100000000 di interi, allocando memoria per un solo intero!

sequenze infinite

Il vantaggio degli iteratori è che consentono di realizzare sequenze infinite, con un impegno trascurabile di memoria:

```
>>> def even_numbers():
...   i=2
...   while True:
...     yield i
...     i += 2
...
>>> for n in even_numbers():
...   if not is_sum_of_two_primes(n):
...     print “Goldbach was wrong!”
...     break
...
>>>
```

iterazione sui contenitori/1

Cosa succede quando iteriamo su una lista?

```
>>> lst = [1, 2, 3]
>>> for i in lst:    # => for i in iter(lst):
                     # => for i in lst.__iter__():
...     print i
...
1
2
3
>>>
```

iterazione sui contenitori/2

Il metodo __iter__ restituisce un iteratore, uno strano oggetto che possiede un metodo next grazie al quale è possibile accedere in sequenza agli elementi della lista:

```
>>> ilst = iter(lst) # ilst = lst.__iter__()
>>> next(ilst)       #ilst.next()
1
>>> next(ilst)       #ilst.next()
2
>>> next(ilst)       #ilst.next()
3
>>> next(ilst)       # lista esaurita, errore!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

iterazione sui contenitori/3

Perché passare per un metodo __iter__ che restituisce un iteratore? Perché non mettere next direttamente nella lista? La ragione è che questo mantiene la possibilità di annidare gli iteratori su una stessa lista; ciascun iteratore mantiene il proprio stato, sono indipendenti anche se si riferiscono alla stessa lista.

```
>>> for i in lst:
...     for j in lst:
...         print i+j,
...
2 3 4 3 4 5 4 5 6
>>>
```

iterazione sui contenitori/4

In pratica, è come se l'iteratore fosse un fittizio “puntatore” ai singoli elementi della lista. Ogni volta che il puntatore è incrementato (next), esso punta all'elemento successivo. Su una medesima lista, posso avere quanti puntatori indipendenti voglio.

iterazione sui contenitori/5

Oltre ai metodi *keys, values, items*, i dizionari hanno anche i metodi *iterkeys, itervalues, iteritems*, che non restituiscono liste, ma iteratori sulle liste.

iteratore

Il concetto di iteratore è un tipico esempio di design pattern, ovvero una soluzione efficiente e diffusa ad un problema generale.

Concettualmente, è una estensione del concetto di puntatore C.

In python il concetto è pervasivo: tutto ciò che scorre su un oggetto da sinistra a destra lo fa attraverso iteratori.

funzioni built-in che operano su iteratori

Esistono molte funzioni built-in che operano su oggetti iterabili:

- *list*(*iterable*): costruisce una lista
- *tuple*(*iterable*): costruisce una tupla
- *dict*(*iterable*): costruisce un dizionario (attenzione: gli elementi di *iterable* devono essere coppie chiave/valore, ovvero tuple di 2 elementi)

generatori built-in/1

Esistono molti utili generatori built-in:

- *xrange*([start,] stop[, incr]): genera una sequenza
- *zip*(*it1, it2*): genera una sequenza di coppie I cui elementi sono presi da *it1* e da *it2*
- *enumerate*(*it*): genera una sequenza di coppie *(i, e)* dove *e* è un elemento di *it*, ed *i* il suo indice.

generatori built-in/2

```
>>> l1 = [1, 2, 3]
>>> l2 = ['a', 'b', 'c']
>>> print zip(l1, l2)
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> for a, b in zip(l1, l2):
...   print "%s=%s" % (b, a)
...
a=1
b=2
c=3
>>> l1 = ['a', 'b', 'c']
>>> l2 = [1, 2, 3]
>>> print zip(l1, l2)
[('a', 1), ('b', 2), ('c', 3)]
>>>
```

generatori built-in/3

```
>>> for k, v in zip(l1, l2):
...     print "%s==%s" % (k, v)
...
a==1
b==2
c==3
>>> print dict(zip(l1, l2))
{'a': 1, 'c': 3, 'b': 2}
>>>
```

generatori built-in/4

```
>>> for i, e in enumerate(l1):
...     print i, e
...
0 a
1 b
2 c
>>>
```

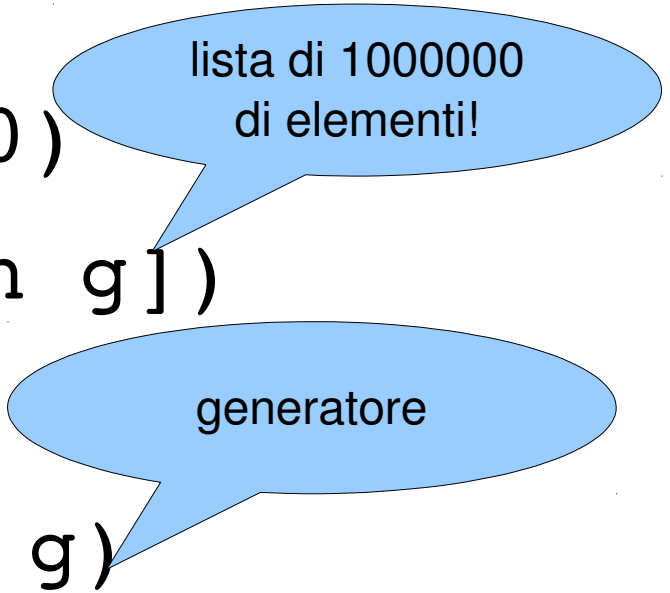
generator expressions/1

Se si usano le parentesi tonde invece delle parentesi quadrate, la stessa sintassi della list comprehension permette di definire generatori on-the-fly:

```
>>> print [i**3 for i in xrange(4)]
[0, 1, 8, 27]
>>> print (i**3 for i in xrange(4))
<generator object <genexpr> at 0x7f30e99d3fa0>
>>> print sum((i**3 for i in xrange(4)))
36
>>> print sum(i**3 for i in xrange(4))
36
>>>
```

generator expressions/2

```
>>> g = xrange(1000000)
>>> sum([e**2 for e in g])
333332833333500000
>>> sum(e**2 for e in g)
333332833333500000
```



test *(primes.py)*

- Scrivere un generatore “primes” che produca la successione dei numeri primi (tutti!). L'efficienza non è importante.
- Usando il generatore “primes”, creare una lista dei primi 250 numeri primi.
- Usando il generatore “primes”, definire una funzione “*pi*(N)” che restituisca il numero di numeri primi inferiori a N.

introspezione/1

L'introspezione è la capacità di un linguaggio di fornire varie informazioni sugli oggetti run-time.

Python ha un ottimo supporto per l'introspezione, a differenza di linguaggi come Fortran o C che non ne hanno alcuno, o C++ che ha un supporto estremamente limitato.

È utile per

- Debugging
- Apprendere più facilmente come usare le librerie
- Realizzare certi algoritmi

introspezione/2

Determinare il typo di un oggetto è estremamente facile: basta usare il comando *type*:

```
>>> l = [1, "alfa", 0.9, (1, 2, 3)]
>>> print [type(i) for i in l]
[<type 'int'>, <type 'str'>, <type 'float'>, <type 'tuple'>]
>>>
```

introspezione/3

Questo talvolta è utile nelle funzioni, perché il tipo degli argomenti di una funzione non è fissato, ma dipende da come la funzione viene chiamata.

Il comando *isinstance(object, type)* ci dice se l'oggetto *object* è del tipo *type*:

```
>>> def dupl(a):
...   if isinstance(a, list):
...     return [dupl(i) for i in a]
...   else:
...     return 2*a
...
>>> print dupl(10)
20
>>> print dupl(['a', 3, [1, 2, 3]])
['aa', 6, [2, 4, 6]]
>>>
```

introspezione/4

Ancora più utile, soprattutto quando dobbiamo imparare ad usare una libreria sviluppata da altri, è il comando *dir()*. Questo comando mostra il “contenuto” di un oggetto, vale a dire i suoi attributi.

```
>>> print l
[3, 4, 1]
>>> print type(l)
<type 'list'>
>>> print dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
>>> print l.sort.__doc__
L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
>>> l.sort(lambda x, y: cmp(x,y))
>>> print l
[1, 3, 4]
>>>
```

introspezione/5

Spesso gli oggetti hanno attributi “speciali” che contengono informazioni utili per l'introspezione:

```
>>> print l.__class__
<type 'list'>
>>> print l.__class__.__name__
list
>>> f = l.sort
>>> print f.__name__
sort
>>>
```

introspezione/6

Spesso si desidera sapere se una istanza possiede un certo attributo:

```
>>> if hasattr(a, 'x'):
...     print a.x
...
3
>>>
```

struttura degli oggetti/1

Per capire come funziona l'introspezione, è utile capire come sono strutturati gli oggetti.

```
>>> class ALFA(object):
...   A = 10
...   def __init__(self):
...     self.x = 3
...
>>> a = ALFA()
>>> print ALFA.__dict__
{'A': 10, '__module__': '__main__', '__dict__': <attribute
'__dict__' of 'ALFA' objects>, '__weakref__': <attribute
'__weakref__' of 'ALFA' objects>, '__doc__': None, '__init__':
<function __init__ at 0x18e1140>}
>>> print a.__dict__
{'x': 3}
>>>
```

struttura degli oggetti/2

Ogni oggetto ha un attributo __dict__ contenente un dizionario; __dict__ contiene tutti gli attributi dell'oggetto, indicizzati con il relativo nome:

```
>>> print a.__dict__['x']
3
>>>
```

struttura degli oggetti/3

Quando si accede ad un attributo di un oggetto, viene cercato prima nel dizionario dell'oggetto, poi nel dizionario della sua classe (ed eventualmente delle classi di base).

```
>>> print a.A
10
>>> print a.__dict__['A']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A'
>>> print a.__class__.__dict__['A']
10
>>>
```

test *(interactive)*

Usando l'interprete in interattivo, determinare come si usano le seguenti funzioni: ord, chr, callable.

moduli/1

Un file con terminazione .py costituisce un modulo python.

Un modulo può contenere qualsiasi tipo di codice python.

Il modulo può possedere una sua doc string.

moduli/1

Ad esempio:

```
$ cat my.py
def hello_world():
  print “Ciao, mondo!”
$ python
>>> import my
>>> dir(my)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'hello_world']
>>> print my.__file__
my.py
>>> my.hello_world()
Ciao, mondo!
>>>
```

moduli/2

Un modulo può essere eseguito direttamente:

```
$ cat my.py
def hello_world():
  print “Ciao, mondo!”

if __name__ == '__main__':
  hello_world()

$ ./my.py
Ciao, mondo!
```

moduli/3

Da un modulo è possibile importare in vari modi:

```
>>> import mymodule
>>> mymodule.myf1()
>>> from mymodule import myf1, myf2
>>> myf1()
>>> from mymodule import *
>>> myf10()
```

moduli/4

Quando da un modulo si importa '*', per default vengono importati tutti i simboli del modulo il cui nome non inizia con underscore.

Se il modulo contiene una variabile __all__, questa deve essere una lista dei nomi dei simboli importati quando si importa '*'.

```
$ cat my.py
__all__ = [ 'hi_folk']
...
```

packages/1

I moduli possono essere raggruppati in pacchetti, che hanno una struttura gerarchica rappresentata da directory. Una directory contenente un file __init__.py, eventualmente vuoto, è un pacchetto. Se la directory contiene altri pacchetti o moduli, essi sono accessibili come contenuto del pacchetto.

packages/2

```
sound/                  Top-level package
      __init__.py       Initialize the sound package
      formats/          Subpackage for file format conversions
              __init__.py wavread.py wavwrite.py
              aiffread.py aiffwrite.py auread.py
              auwrite.py  ...
      effects/          Subpackage for sound effects
              __init__.py echo.py surround.py
              reverse.py  ...
      filters/          Subpackage for filters
              __init__.py equalizer.py vocoder.py
              karaoke.py  ...
```

packages/3

```
>>> import sound.effects.echo.echofilter
>>> sound.effects.echo.echofilter(...)

>>> import sound.effects.echo
>>> sound.effects.echo.echofilter(...)

>>> from sound.effects import echo
>>> echo.echofilter(...)

>>> from sound.effects.echo import echofilter
>>> echofilter(...)
```

packages/4

Come per i moduli, si può importare '*' da un pacchetto, ma solo a condizione che __init__.py definisca __all__ = [...]; infatti la determinazione del nome del modulo dai file contenuti nella directory del pacchetto non è possibile sotto Windows (case insensitive)!

test *(interactive)*

- Dall'interprete in interattivo importare il modulo “primes”, scoprirne il contenuto e provare ad usare le funzioni del modulo.
- Provare ad importare il modulo “math” ed a scoprirne il contenuto.

pydoc/1

Pydoc è un tool, ovviamente scritto in python, che utilizza l'introspezione per fornire le informazioni racchiuse in un modulo in maniera chiara e compatta.

Pydoc utilizza le doc string __doc__ ed i vari altri attributi standard che gli oggetti hanno (__name__, __file__, ...).

pydoc/2

$ pydoc math

Help on built-in module math:

NAME

math

FILE

(built-in)

DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS

acos(...)

acos(x)

Return the arc cosine (measured in radians) of x.

...

help()

All'interno dell'interprete, la funzione *help* fa la stessa cosa che fa pydoc.

test *(interactive)*

- Usare pydoc o help() per scoprire il contenuto di primes

argomenti del programma/1

Per accedere agli argomenti con cui il programma è stato chiamato, bisogna importare il modulo *sys* ed accedere alla variabile *sys.argv*. Questa è una lista di stringhe; *sys.argv[0]* è il nome del programma, gli elementi restanti sono gli argomenti passati a riga di comando.

argomenti del programma/2

```
#!/usr/bin/env python
import sys
for arg in sys.argv:
   print arg
```

file I/O/1

Ovviamente un file è un oggetto:

```
>>> f = open("a.txt", "wb")
>>> f.write("Ciao,\n")
>>> f.write("mondo!\n")
>>> f.close()
>>> f = open("a.txt", "rb")
>>> a = f.read()
>>> print a
Ciao,
mondo!

>>> f.close()
>>>
```

file I/O/2

Si può leggere riga per riga iterando sul file; attenzione, il newline '\n' fa parte della stringa letta!

```
>>> for line in open("a.txt", "rb"):
...   print line
...
Ciao,


mondo!


>>>
```

file I/O/3

Si può leggere una singola riga con readline, o tutte le righe con readlines:

```
>>> f = open("a.txt", "rb")
>>> print f.readline()
Ciao,

>>> f.close()
>>> f = open("a.txt", "rb")
>>> print f.readlines()
['Ciao,\n', 'mondo!\n']
>>> f.close()
>>>
```

test *(files.py)*

- Leggere la lista di argomenti a riga di comando, e stampare (un argomento per riga) tutti gli argomenti su un file il cui nome è uguale al nome del programma python più l'estensione '.out' (files.py.out, ma deve funzionare anche rinominando il programma!).
- Rileggere tutti gli argomenti dal file e stamparli.

gestione degli errori/1

La gestione degli errori è una problematica complessa, che i linguaggi moderni affrontano in maniera completamente diversa rispetto a quelli “vecchi”.

Innanzitutto, bisogna prendere coscienza del fatto che il luogo in cui un errore può essere individuato (ad esempio, una funzione di libreria come *sqrt*) non coincide con il luogo in cui l'errore può essere “trattato” (ad esempio, il *main program*).

gestione degli errori/2

I requisiti di un moderno sistema per la gestione degli errori sono:

- Basso o nessun impatto sulle performance, quando non vengono generati errori
- Poca invasività sul codice
- Non deve essere necessario “passare” l'errore a mano
- Deve essere possibile una gestione “parziale” dell'errore (a volte non è possibile risolvere completamente l'errore in un punto, ma si può applicare solo una parte della correzione)

gestione degli errori/3

Supponiamo di avere uno stack di chiamate di funzione come questo:

```
main

   compute_matrix <- qui l'errore ZeroDivisionError può essere
gestito completamente

      compute_cell <- qui l'errore BadStartingPoint può essere
gestito, e l'errore ZeroDivisionError può essere gestito parzialmente

         compute_radix_of_function

            newton -> qui l'errore BadStartingPoint può essere
individuato

               function_C -> qui l'errore ZeroDivisionError può
essere individuato
```

gestione degli errori/4

In Fortran, ad esempio, si usa una variabile restituita dalla funzione/subroutine per passare un eventuale errore.

In questo caso, la variabile contenente l'errore dovrebbe essere passata manualmente a ritroso, ad esempio da *function_C* a *newton* a *compute_radix_of_function* a *compute_cell* a *compute_matrix*.

gestione degli errori/5

Quali sono gli svantaggi?

- È una attività noiosa, e le attività noiose normalmente conducono ad errori
- Rende molto più lungo e complicato il codice
- Aggiunge overhead anche nel caso in cui non si verifica alcun errore (vi saranno degli *if* sul valore delle variabili di stato della funzione che genera l'errore e delle funzioni chiamanti, ad esempio).

gestione degli errori/6

Tutti i sistemi moderni usano un approccio differente.

Consideriamo l'errore BadStartingPoint.

Nel punto in cui l'errore può essere individuato (*newton*), viene lanciata una *eccezione* di tipo *BadStartingPoint*. Per ora non preoccupiamoci di cosa è *BadStartingPoint*: in effetti potrebbe essere qualsiasi cosa, un *int*, una stringa, una lista etc...

In python, le eccezioni si lanciano con il comando *raise*.

In fun_B comparirà dunque qualcosa come:

```
if ...:
  raise BadStartingPoint()
```

gestione degli errori/7

Quando viene lanciata una eccezione, il flusso del programma si interrompe, e lo stack viene automaticamente “srotolato” all'indietro, tornando alla funzione che ha chiamato quella che ha lanciato l'eccezione, ed ancora indietro se necessario, fino ad un punto in cui l'errore può essere “trattato”. La computazione riprende da questo punto.

gestione degli errori/8

Come si fa a stabilire chi può trattare un certo errore?

È semplice: il blocco di codice che *potrebbe* gestire una eccezione *BadStartingPoint* viene racchiuso in una sezione apposita; se si verifica quell'eccezione, viene eseguita la relativa sezione di gestione. La sintassi python si basa sul costrutto *try/except*.

gestione degli errori/9

Dunque, nella funzione dove abbiamo detto che l'errore può essere trattato (*compute_cell*) viene inserito un blocco try/catch:

```
def compute_cell(matrix, i, j):
...
      try:
        matrix[i][j] += compute_radix_of_function(f,
cell, x_0)
      except BadStartingPoint, e:
        print "ERR: %s: %s" % (e.__class__.__name__, e)
        X_0 += 0.4
...
```

gestione degli errori/10

Nelle funzioni intermedie nulla cambia: non sono coinvolte da quell'eccezione.

gestione degli errori/11

Nel caso di ZeroDivisionError, invece, la gestione dell'errore è più complessa: *compute_cell* può riparare parzialmente l'errore, ma non completamente. Il resto del lavoro lo fa *compute_matrix*.

In questo caso, l'eccezione viene raccolta, parzialmente gestita e rilanciata, con il solo comando *raise*:

```
...
      except ZeroDivisionError, e:
        print "ERR: ZeroDivisionError: resetting
cell"
        matrix[i][j] = 0.0
        raise
...
```

gestione degli errori/12

A questo punto lo stack viene nuovamente srotolato indietro fino a *compute_matrix*, che completa la gestione dell'errore.

gestione degli errori/13

Generalmente inoltre le eccezioni sono definite gerarchicamente. Nel nostro esempio, vi sono tre tipi di errori BadStartingPoint:

- StationaryStartingPoint
- CyclingStartingPoint
- SlowlyConvergingStartingPoint

newton lancia tutti e tre questi errori.

Queste tre tipologie di errore vengono gestite tutte allo stesso modo da *compute_cell*, ma non è escluso che altre funzioni che chiamano *newton* debbano gestirli diversamente.

Questo viene realizzato creando una classe BadStartingPoint, e le tre classi StationaryStartingPoint, CyclingStartingPoint, SlowlyConvergingStartingPoint che ereditano da essa.

gestione degli errori/14

Che cosa sono BadStartingPoint, StationaryStartingPoint, etc...? Sono “tipi” di eccezioni, o, più in generale, “tipi”: come *int*, *str*, …

Saranno però tipi definiti dall'utente, ovvero *classi*.

try/except

```
try:
  ...
  ...
except Exc0:
  ... # cattura le eccezioni di tipo Exc0
except Exc1, exc_instance:
  ... # cattura le eccezioni di tipo Exc1, e la relativa istanza e' exc
except (Exc2, Exc3):
  ... # cattura le eccezioni di tipo Exc2 o Exc3
except (Exc4, Exc5) as exc_instance:
  ... # cattura le eccezioni di tipo Exc2 o Exc3, e la relativa istanza e' exc
except:
  ... # cattura qualsiasi eccezione
else:
  ... # eseguito solo se non si sono catturate eccezioni
finally:
  ... # eseguito sempre e comunque
```

esempio else/finally

```
try:
   f = open(“a.txt”, “rb”)
   do_some_action_on_file(f)
except:
   print “ERROR”
else:
   print “OK”
finally:
   f.close()
```

eccezioni standard/1

Ora possiamo capire meglio cosa avviene quando c'è un errore in un programma:

```
>>> print 4/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division
or modulo by zero
>>>
```

eccezioni standard/2

```
>>> l = [1, 2, 3]
>>> print l[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> l.remove(444)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> d = {}
>>> print d['alfa']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'alfa'
>>>
```

eccezioni standard/3

```
>>> l = [1, 2]
>>> il = iter(l)
>>> il.next()
1
>>> il.next()
2
>>> il.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

eccezioni standard/4

Ora sappiamo che un ciclo for termina quando l'iteratore su cui opera lancia una eccezione *StopIterator*!

```
>>> def range_iter(n):
...   i = 0
...   while True:
...     if i >= n: raise StopIteration
...     yield i
...     i += 1
...
>>> for i in range_iter(3):
...   print i
...
0
1
2
>>>
```

test *(primes.py)*

- Modificare la funzione *is_prime* in modo da generare un errore se l'argomento non è un intero, oppure se è un intero negativo.

  Scrivere un programma di test con la gestione dell'errore.