CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 1

Python

Python è in questo momento forse il miglior linguaggio di programmazione: per la facilità di apprendimento e di utilizzo, per le caratteristiche di linguaggio ad altissimo livello che realizza i concetti sia della programmazione funzionale che della programmazione orientata agli oggetti, per il recente perfezionamento della libreria per la programmazione insiemistica, per il supporto da parte di numerosi programmatori, per l'ottima documentazione disponibile in rete e la ricerca riuscita di meccanismi di leggibilità, per la grafica con Tkinter, per la semplicità dell'aggancio ad altri linguaggi, di cui il modulo RPy per il collegamento con R è un esempio meraviglioso.

A. Martelli: Python in a nutshell. O'Reilly 2003. Un ottimo compendio. L'autore, piuttosto famoso, è italiano e risiede a Bologna. Ha insegnato a Ferrara e lavora per Google.

M. Lutz/D. Ascher: Programmare con Python. Hoepli 2004.

J. Kiusalaas: Numerical methods in engineering with Python. Cambridge UP 2005. Un corso elementare e molto leggibile di analisi numerica con Python.

H. Langtangen: Python scripting for computational science. Springer 2004. Scritto da un matematico, il libro è piuttosto denso e ricco di temi, utilizza forse un po' troppo librerie esterne che bisogna installare.

S. Holden: Python Web programming. New Riders 2002. Dedicato alla programmazione per il Web, questo libro fornisce anche un'ottima introduzione alla programmazione orientata agli oggetti con Python.

D. Mertz: Text processing in Python. Addison-Wesley 2003. Un'introduzione, a livello piuttosto elevato, all'elaborazione di testi con Python.

www.python.it. Ottimo sito italiano.

www.python.org. Sito ufficiale.

www.aleax.it. Sito di Alex Martelli.

it.wikipedia.org/wiki/Python. Wikipedia italiana.

wiki.python.org/moin/. Wikipedia sul sito ufficiale.

felix.unife.it. Seguire la voce Python.

Oltre ad Alex Martelli, dal dicembre 2005 anche Guido van Rossum, inventore di Python, lavora per Google.

Combinare Python ed R con RPy

Il modulo RPy è un piccolo miracolo e permette una quasi perfetta e semplicissima collaborazione tra Python ed R.

Sotto Linux il pacchetto va installato nel modo seguente: In primo luogo è necessario che R sia stato creato in modo che si possano utilizzare le librerie condivise:

```
./configure --enable-R-shlib
make
make install
```

Successivamente va aggiunta la riga

```
/usr/local/lib/R/lib
```

nel file */etc/ld.so.conf* ed eseguito il comando `ldconfig`.

A questo punto, per installare RPy stesso, è sufficiente

```
/usr/local/bin/python setup.py install
```

Sotto Windows sembra che attualmente non si possa ancora utilizzare RPy con le ultime versioni di Python ed R; useremo perciò Python 2.3.5 (di Enthought) ed R 2.1.1 (peraltro del 2005). Bisogna (forse) poi installare la libreria *PyWin32* e infine RPy per Python 2.3, come indicato sul sito del corso.

Per importare il pacchetto scriviamo

```
from rpy import r
```

nel programma in Python. Le funzioni di R possono allora essere usate anteponendo il prefisso `r.`, come in `r.fun(argomenti)`. Sotto Linux bisogna (per un piccolo errore contenuto nel modulo) inserire l'istruzione `r.q()` alla fine del programma.

Definiamo ad esempio una funzione in Python che utilizza la funzione `mean` di R per calcolare la media di un vettore:

```
def Media (x): return r.mean(x)
```

Per provare la funzione usiamo

```
print Media([1,5,8,6,3,1])
# output: 4.0
```

In particolare possiamo usare la funzione `source` di R. Ciò significa che possiamo creare una raccolta di funzioni in R da noi programmate; queste funzioni possono a loro volta utilizzare (come se fossimo in una libreria creata per R) le altre funzioni di quella raccolta e allo stesso tempo essere usate, nella sintassi indicata, nei programmi in Python! Creiamo ad esempio un file *funz.r*:

```
# funz.r

cubo = function (x): x^2
```

In uno script di Python scriviamo poi

```
r.source('funz.r')

print r.cubo(13)
# output: 2197.0
```

www.math.tamu.edu/~wmoreira. Sito di Walter Moreira, uruguaiano, dottorando in matematica (algebre di Hopf con Marcelo Aguiar) a College Station nel Texas e creatore di RPy.

Noi useremo R per le sue funzioni statistiche su dati che spesso verranno preparati in Python; avremo così bisogno solo di un repertorio minimo di funzioni di R. Sono molto utilizzabili le funzioni d'aiuto interattive che possono essere richiamate o durante una sessione di R oppure anche da Python, ad esempio con `print r.help('cos')`.

In questo numero

Enthought Python

Questa è una raccolta molto ricca (di 95 MB) e preferita anche da H. Langtangen (pag. 660 del libro) che non comprende soltanto il linguaggio Python (attualmente nella versione 2.3.5), ma anche numerose librerie scientifiche:

wxPython. Un'interfaccia alla libreria grafica *wxWidgets*.

PIL. La *Python Imaging Library*, una libreria per l'elaborazione delle immagini.

VTK. Il *3D Visualization Toolkit*, un sofisticato pacchetto di grafica 3-dimensionale.

MayaVi. Una raccolta di programmi per la visualizzazione 3-dimensionale di dati scientifici basata su VTK.

Numeric. Un pacchetto di calcolo numerico, soprattutto matriciale.

ScientificPython. Libreria per il calcolo numerico scientifico.

SciPy. Una nuova libreria per il calcolo scientifico.

F2Py. Un'interfaccia tra Fortran e Python.

Chaco. Una libreria per la grafica scientifica sviluppata dalla Enthought.

Traits. Uno strumento di sviluppo per la programmazione orientata agli oggetti.

PyCrust. Una shell per Python che fa parte di *wxPython*.

ZODB3. Due sistemi di basi di dati orientate agli oggetti.

Gadfly. Un sistema di basi di dati relazionali.

PySQLite. Un'estensione per SQLite.

ctypes. Un pacchetto per la creazione di tipi di dati per il C.

Nel corso useremo soprattutto i pacchetti scientifici e grafici.

Commenti

Se una riga contiene, al di fuori di una stringa, il carattere `#`, tutto il resto della riga è considerato un commento, compreso il carattere `#` stesso.

Molti altri linguaggi interpretati (Perl, R, la shell di Unix) usano questo simbolo per i commenti. In C e C++ una funzione analoga è svolta dalla sequenza `//`.

Primi esempi

```
a=range(5,13,2)
print a
# output: [5, 7, 9, 11]

a=range(5,13)
print a
# output: [5, 6, 7, 8, 9, 10, 11, 12]

a=range(11)
print a
# output:
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Si noti che il limite destro non viene raggiunto.

```
a=xrange(5,13,2)
print a
# output: xrange(5, 13, 2)

for x in a: print x,
# output: 5 7 9 11
```

La differenza tra `range` e `xrange` è questa: Mentre `range(1000000)` genera una lista di un milione di elementi, `xrange(1000000)` è un iteratore (pag. 3) e crea questi elementi uno allo volta in ogni passaggio di un ciclo in cui il comando viene utilizzato.

Si noti il doppio punto (`:`) alla fine del comando `for`.

Sono possibili assegnazioni e confronti simultanei:

```
if 3<5<9: print 'o.k.'
# output: o.k.

a=b=c=4
for x in [a,b,c]: print x,
print
# output: 4 4 4
```

Funzioni in Python:

```
def f(x): return 2*x+1

def g(x):
  if (x>0): return x
  else: return -x

for x in xrange(0,10): print f(x),
print
# output: 1 3 5 7 9 11 13 15 17 19

for x in xrange(-5,5): print g(x),
print
# output: 5 4 3 2 1 0 1 2 3 4
```

A differenza da R, il `return` è obbligatorio.

La virgola alla fine di un comando `print` fa in modo che la stampa continui sulla stessa riga. Come si vede nella definizione di `g`, Python utilizza l'indentazione per strutturare il programma. Anche le istruzioni `if` ed `else` richiedono il doppio punto.

Una funzione di due variabili:

```
import math

def raggio (x,y):
  return math.sqrt(x**2+y**2)

print raggio(2,3)
# output: 3.60555127546

print math.sqrt(13)
# output: 3.60555127546
```

Funzioni possono essere non solo argomenti, ma anche risultati di altre funzioni:

```
def sommax (f,g,x): return f(x)+g(x)

def compx (f,g,x): return f(g(x))

def u(x): return x**2

def v(x): return 4*x+1

print sommax(u,v,5)
# output: 46

print compx(u,v,5)
# output: 441
```

Possiamo però anche definire

```
def somma (f,g):
  def s(x): return f(x)+g(x)
  return s

def comp (f,g):
  def c(x): return f(g(x))
  return c

def u(x): return x**2

def v(x): return 4*x+1

print somma(u,v)(5)
# output: 46

print comp(u,v)(5)
# output: 441
```

Queste costruzioni significano che Python appartiene (come R, Perl e Lisp) alla famiglia dei potenti linguaggi *funzionali*.

Stringhe sono racchiuse tra apici o virgolette, stringhe su più di una riga tra triplici apici o virgolette:

```
print 'Carlo era bravissimo.'
# output: Carlo era bravissimo.

print "Carlo e' bravissimo."
# output: Carlo e' bravissimo.

print '''Stringhe a piu' righe si
usano talvolta nei commenti.'''
# output:
# Stringhe a piu' righe si
# usano talvolta nei commenti.
```

Funzioni con un numero variabile di argomenti: Se una funzione è dichiarata nella forma `def f(x,y,*a):`, l'espressione `f(2,4,5,7,10,8)` viene calcolata in modo che gli ultimi argomenti vengano riuniti in una tupla `(5,7,10,8)` che nel corpo del programma può essere vista come se questa tupla fosse a.

```
def somma (*a):
  s=0
  for x in a: s+=x
  return s

print somma(1,2,3,10,5)
# output: 21
```

Ricordiamo che lo schema di Horner per il calcolo dei valori $f(\alpha)$ di un polinomio $f = a_0x^n + \ldots + a_n$ consiste nella ricorsione

$$b_{-1} = 0$$
$$b_k = \alpha b_{k-1} + a_k$$

per $k = 0, \ldots, n$. Possiamo quindi definire

```
def horner (alfa,*a):
  alfa=float(alfa); b=0
  for t in a: b=b*alfa+t
  return b

print horner(10,6,2,0,8)
# output: 6208.0
```

Vettori associativi (dizionari o tabelle di hash) vengono definiti nel modo seguente:

```
latino = {'casa': 'domus',
  'villaggio': 'pagus',
  'nave':'navis', 'campo':'ager'}
voci=sorted(latino.keys())
for x in voci:
  print '%-9s = %s' %(x,latino[x])
# output:
# campo     = ager
# casa      = domus
# nave      = navis
# villaggio = pagus
```

Scambi simultanei:

```
a=5; b=3; a,b=b,a; print [a,b]
# output: [3, 5]
```

Esecuzione

In una cartella *Python* (oppure, per progetti più importanti, in un'apposita cartella per quel progetto) creiamo i files sorgente come files di testo puro con l'estensione *.py*, utilizzando l'editor incorporato di Python. Con lo stesso editor, piuttosto comodo, scriviamo anche, usando l'estensione *.r*, le sorgenti in R che vogliamo affiancare ai programmi in Python. I programmi possono essere eseguiti mediante il tasto *F5* nella finestra dell'editor. Soprattutto in fase di sviluppo sceglieremo questa modalità di esecuzione, perché così vengono visualizzati anche i messaggi d'errore.

Successivamente i programmi possono essere eseguiti anche tramite il clic sull'icona del file oppure, in un terminale (Prompt dei comandi) e se il file si chiama *alfa.py*, con il comando `python alfa.py`.

Teoricamente i programmi possono essere scritti con un qualsiasi editor che crea files in formato testo puro, ad esempio il *Blocco note* di Windows, ma preferiamo utilizzare l'editor di Python per una più agevole correzione degli errori, per l'indentazione automatica e perché prevede la possibilità di usare combinazioni di tasti più comode di quelle disponibili per il Blocco note.

L'editor di Python può sostituire anche per altri compiti il Blocco note, per cui piazziamo la sua icona sulla scrivania di Windows.

Elenchiamo alcune combinazioni di tasti:

Ctrl Q	uscire
F5	esecuzione
Ctrl N	nuovo file
Ctrl O	aprire un file
Alt G	trova una riga
Ctrl A	seleziona tutto
Ctrl C	copia il testo selezionato
Ctrl V	incolla
Ctrl X	cancella il testo selezionato
Ctrl K	cancella il resto della riga
Inizio	inizio della riga
Fine	fine della riga
Ctrl E	fine della riga
Alt P	lista dei comandi dati: indietro
Alt N	lista dei comandi dati: avanti

Input dalla tastiera

Sono previsti due comandi per l'input dalla tastiera, `raw_input` e `input`, effettuabili anche nella finestra principale dell'editor. Entrambe accettano un argomento facoltativo che, quando è presente, viene visualizzato sullo schermo prima dell'input dell'utente. Mentre `input` aspetta dalla tastiera una espressione valida in Python che viene calcolata come se facesse parte di un programma, `raw_input` tratta la risposta dell'utente come una stringa. Se ad esempio vogliamo immettere una stringa con `input`, la dobbiamo racchiudere tra apici, mentre con `raw_input` gli apici verrebbero considerati come lettere della stringa.

Esempi:

```
>>> x=input('nome: ')
nome: Giacomo
...
NameError: name 'Giacomo' is not defined
>>> # Giacomo non e' una stringa.

>>> x=input('nome: ')
nome: 'Giacomo'
>>> print x
Giacomo

>>> x=raw_input('nome: ')
nome: Giacomo
>>> print x
Giacomo

>>> x=raw_input('nome: ')
nome: 'Giacomo'
>>> print x
'Giacomo'

>>> def f(x): return x**2

>>> x=input('espressione: ')
espressione: f(9)
>>> print x
81

>>> x=raw_input('espressione: ')
espressione: f(9)
>>> print x
f(9)
```

os.system

Il comando `os.system` permette di eseguire, dall'interno di un programma in Python, comandi di Windows (o comunque del sistema operativo), ad esempio

```
import os

os.system('dir > catalogo')
```

per scrivere l'elenco dei files nella cartella attiva nel file *catalogo*.

`os.system` apparentemente funziona solo all'interno dei programmi, non nell'editor di Python.

„Python is a general-purpose programming language ... This stable and mature language is very high level, dynamic, object-oriented, and cross-platform - all characteristics that are very attractive to developers. Python runs on all major hardware platforms and operating systems ... Python provides a unique mix of elegance, simplicity, and power.“ (Martelli, pag. 3)

Liste

Liste sono successioni finite modificabili di elementi non necessariamente dello stesso tipo. Python fornisce numerose funzioni per liste che non esistono per le tuple. Come le tuple anche le liste possono essere annidate, la lunghezza di una lista `v` la si ottiene con `len(v)`, l'`i`-esimo elemento è `v[i]`. A differenza dalle tuple, la lista che consiste solo di un singolo elemento `x` è denotata con `[x]`.

```
v=[1,2,5,8,7]

for i in xrange(5): print v[i],
print
# 1 2 5 8 7

for x in v: print x,
print
# 1 2 5 8 7

a=[1,2,3]; b=[4,5,6]
v=[a,b]

print v
# [[1, 2, 3], [4, 5, 6]]

for x in v: print x
# [1, 2, 3]
# [4, 5, 6]

print len(v)
# 2
```

Sequenze

Successioni finite in Python vengono dette *sequenze*, di cui i tipi più importanti sono liste, tuple e stringhe. Tuple e stringhe sono sequenze non modificabili. Esistono alcune operazioni comuni a tutte le sequenze che adesso elenchiamo. `a` e `b` siano sequenze:

`x in a`	Vero, se `x` coincide con un elemento di `a`.
`x not in a`	Vero, se `x` non coincide con nessun elemento di `a`.
`a + b`	Concatenazione di `a` e `b`.
`a * k`	Concatenazione di `k` copie di `a`.
`a[i]`	`i`-esimo elemento di `a`.
`a[-1]`	Ultimo elemento di `a`.
`a[i:j]`	Sequenza che consiste degli elementi `a[i]`, ..., `a[j-1]` di `a`.
`a[:]`	Copia di `a`.
`len(a)`	Lunghezza di `a`.
`min(a)`	Più piccolo elemento di `a`. Per elementi non numerici viene utilizzato l'ordine alfabetico.
`max(a)`	Più grande elemento di `a`.
`sorted(a)`[2.4]	*Lista* che contiene gli elementi di `a` in ordine crescente. Si noti che il risultato è sempre una lista, anche quando `a` è una stringa o una tupla.
`reversed(a)`[2.4]	*iteratore* che corrisponde agli elementi di `a` elencati in ordine invertito.
`list(a)`	Converte la sequenza in una lista con gli stessi elementi.

Come abbiamo visto, l'espressione `x in a` può apparire anche in un ciclo `for`.

Iteratori sono oggetti che forniscono uno dopo l'altro tutti gli elementi di una sequenza senza creare questa sequenza in memoria. Ad esempio sono iteratori gli oggetti che vengono creati tramite l'istruzione `xrange`.

La funzione `list` può essere applicata anche agli iteratori, per cui per generare la lista `b` che si ottiene da una sequenza `a` invertendo l'ordine in cui sono elencati i suoi elementi, possiamo utilizzare `b=list(reversed(a))`.

Come indicato, `sorted` e `reversed` esistono soltanto dalla versione 2.4 di Python e quindi non sono ancora disponibili nella 2.3.5 di Enthought.

Esempi:

```
if 'a' in 'nave': print "Si'."
# Si'.

if 7 in [3,5,1,7,10]: print "Si'."
# "Si'."

if 2 in [3,5,1,7,10]: print "Si'."
else: print "No."
# No.

a=[1,2,3,4]; b=[5,6,7,8,9]
print a+b
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

a='Mario'; b='Rossi'
print a+' '+b
# Mario Rossi

a=[1,2,3]*4
print a
# [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

a='==.==*3
print a
# ==.====.====.==

a=[10,11,12,13,14,15,16,17,18,19,20]
for i in xrange(0,11,3): print a[i],
print
# 10 13 16 19

a='01234567'
print a[2:5]
# 234

a=[3,5,3,1,0,1,2,1]
b=sorted(a)
print b
# [0, 1, 1, 1, 2, 3, 3, 5]
# Non funziona nella 2.3.5.

a=[1,2,3,4,5]
c=reversed(a)
print c
# <listreverseiterator object at ...>
# L'indirizzo dopo at cambia ogni volta.
# Non funziona nella 2.3.5.

d=list(reversed(a))
print d
# [5, 4, 3, 2, 1]
# Non funziona nella 2.3.5.

a='MARIO'
print list(a)
# ['M', 'A', 'R', 'I', 'O']
```

„Python stands out as the language of choice for scripting in computational science because of its very clean syntax, rich modularization features, good support for numerical computing, and rapidly growing popularity.“ (Langtangen, pag. v)

Funzioni per liste

Per le liste sono disponibili alcune funzioni speciali che non possono essere utilizzate per tuple o stringhe. `v` sia una lista.

`v.append(x)`	`x` viene aggiunto alla fine di `v`. Equivalente a `v=v+[x]`, ma più veloce.
`v.extend(w)`	Aggiunge la lista `w` a `v`. Equivalente a `v=v+w`, ma più veloce.
`v.count(x)`	Il risultato indica quante volte `x` appare in `v`.
`v.index(x)`	Indice della prima posizione in cui `x` appare in `v`; provoca un errore se `x` non è elemento di `v`.
`v.insert(i,x)`	Inserisce l'elemento `x` come `i`-esimo elemento della lista.
`v.remove(x)`	Elimina `x` nella sua prima apparizione in `v`; provoca un errore se `x` non è elemento di `v`.
`v.pop()`	Toglie dalla lista il suo ultimo elemento che restituisce come risultato; errore, se il comando viene applicato alla lista vuota.
`v.sort()`	Ordina la lista che viene modificata.
`v.reverse()`	Inverte l'ordine degli elementi in `v`. La lista viene modificata.

Esempi:

```
v=[1,2,3,4,5,6]
v.append(7)
print v
# [1, 2, 3, 4, 5, 6, 7]

v=[2,8,2,7,2,2,3,3,5,2]
print v.count(2)
# 5
print v.count(77)
# 0

print v.index(7)
# 3

v=[10,11,12,13,14,15,16]
v.insert(4,99)
print v
# [10, 11, 12, 13, 99, 14, 15, 16]

v=[2,3,8,3,7,6,3,9]
v.remove(3)
print v
# [2, 8, 3, 7, 6, 3,9]

v.pop()
print v
# [2, 8, 3, 7, 6, 3]

v.sort()
print v
# [2, 3, 3, 6, 7, 8]

v=[7,0,1,0,2,3,3,0,5]
v.sort()
print v
# [0, 0, 0, 1, 2, 3, 3, 5, 7]

v=[0,1,2,3,4,5,6,7]
v.reverse()
print v
# [7, 6, 5, 4, 3, 2, 1, 0]
```

Tuple

Tuple sono successioni finite *non modificabili* di elementi non necessariamente dello stesso tipo che sono elencati separati da virgole e possono facoltativamente essere incluse tra parentesi tonde. Per ragioni misteriose una tupla con un solo elemento deve essere scritta nella forma `(x,)` perché `(x)` è lo stesso come `x`. Ciò non vale per le liste: `[x]` è una lista. Tuple possono essere annidate.

Esempi:

```
x=3,5,8,9
print x
# (3, 5, 8, 9)

y=(3,5,8,9)
print y
# (3, 5, 8, 9)

s=(7)
print s
# 7

t=(7,)
print t
# (7,)

z=(x)
print z
# (3, 5, 8, 9)

u=(x,)
print u
# ((3, 5, 8, 9),)

v=(x,y)
print v
# ((3, 5, 8, 9), (3, 5, 8, 9))

w=1,2,(3,4,5),6,7
print w
# (1, 2, (3, 4, 5), 6, 7)

vuota=()
print vuota
# ()
```

La lunghezza di una tupla `v` la otteniamo con `len(v)`; l'`i`-esimo elemento di `v` è `v[i]`, contando (come in C e a differenza da R!) cominciando da 0.

```
x=3,5,8,9
print len(x)
# 4

for i in xrange(0,4): print x[i],
print
# 3 5 8 9

for a in x: print a,
# 3 5 8 9
```

Tuple vengono elaborate più velocemente e consumano meno spazio in memoria delle liste; per sequenze molto grandi (con centinaia di migliaia di elementi) o sequenze che vengono usate in migliaia di operazioni le tuple sono perciò talvolta perferibili alle liste. Liste d'altra parte non solo possono essere modificate, ma prevedono anche molte operazioni flessibili che non sono disponibili per le tuple. Le liste costituiscono una delle strutture fondamentali di Python. Con dati molto grandi comunque l'utilizzo di liste, soprattutto nei calcoli intermedi, può effettivamente rallentare notevolmente l'esecuzione di un programma.

Nomi ed assegnamento

A differenza dal C, il Python non distingue tra il nome `a` di una variabile e il suo indirizzo (che in C viene denotato con `&a`). Ciò ha implicazioni a prima vista sorprendenti nelle assegnazioni `b=a` in cui `a` è un oggetto mutabile (ad esempio una lista o un dizionario), mentre nel caso che `a` non sia mutabile (ad esempio un numero, una stringa o una tupla) non si avverte una differenza con quanto ci si aspetta.

Dopo `b=a` infatti `a` e `b` sono nomi diversi per lo stesso oggetto e se cambiamo l'oggetto che corrisponde ad `a`, lo troviamo cambiato anche quando usiamo il nome `b` proprio perché si tratta sempre dello stesso oggetto.

```
a=[1,5,0,2]; b=a; b.sort(); print a
# [0, 1, 2, 5]

b[2]=17; print a
# [0, 1, 17, 5]

b=a[:].sort(); print a
# [0, 1, 17, 5]
# a non e' cambiata.

b=a[:].reverse(); print a
# [0, 1, 17, 5]
# a non e' cambiata.
```

Se gli elementi di `a` sono immutabili, è sufficiente, come sopra, creare una copia `b=a[:]` oppure `b=list(a)` affinché cambiamenti in `b` non influenzino `a`. Ciò non basta più quando gli elementi di `a` sono mutabili, come ad esempio nel caso che `a` sia una lista annidata:

```
a=[[1,2],[3,4]]
b=a[:] # oppure b=list(a)
b[1][0]=17; print a
# [[1, 2], [17, 4]]
```

In questo caso bisogna creare una *copia profonda* utilizzando la funzione `copy.deepcopy` che naturalmente richiede il modulo `copy`:

```
import copy

a=[[1,2],[3,4]]
b=copy.deepcopy(a)
b[1][0]=17; print a
# [[1, 2], [3, 4]]
# a non e' cambiata.
```

Per verificare se due nomi denotano lo stesso oggetto, si può usare la funzione `is`, mentre l'operatore di uguaglianza `==` controlla soltanto l'uguaglianza degli elementi, non degli oggetti che corrispondono ai nomi `a` e `b`:

```
a=[1.5,0,2]; b=a
print b is a
# True

a=[[1,2],[3,4]]; b=a[:]
print b is a
# False
print b[0] is a[0]
# True

a=[[1,2],[3,4]]
b=copy.deepcopy(a)
print b is a
# False
print b[0] is a[0]
# False
print b==a
# True
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 2

Valori di verità

Vero e falso in Python sono rappresentati dai valori `True` e `False`. In un contesto logico, cioè nei confronti o quando sono argomenti degli operatori logici, anche ad altri oggetti è attribuito un valore di verità; come vedremo però, a differenza dal C, essi conservano il loro valore originale e il risultato di un'espressione logica in genere non è un valore di verità, ma uno degli argomenti da cui si partiva.

Con

```
a="Roma"; b="Torino"
for x in (3<5, 3<5<7, 3<5<4,
  6==7, 6==6,a=='Roma',a<b):
  print x,
```

otteniamo

```
True True False False True True True
```

perché la stringa `"Roma"` precede alfabeticamente `"Torino"`. Con

```
for x in (0,1,0.0,[],(),[0],
  None,'','alfa'):
  print "%-6s%s" %(x, bool(x))
```

otteniamo

```
0     False
1     True
0.0   False
[]    False
()    False
[0]   True
None  False
      False
alfa  True
```

Vediamo così che il numero 0, l'oggetto `None`, la stringa vuota, e la lista o la tupla vuota hanno tutti il valore di verità falso, numeri diversi da zero, liste, tuple e stringhe non vuote il valore di verità vero.

In un contesto numerico i valori di verità vengono trasformati in 1 e 0:

```
print (3<4)+0.7
# 1.7

v=[3<4,3<0]
for x in v: print x>0.5,
# True False

print
from rpy import r

a=[True,True,False,True,False]
print r.sum(a)
# 3
```

Operatori logici

Come abbiamo accennato, gli operatori logici `and` e `or`, a differenza dal C, non convertono i loro argomenti in valori di verità. Inoltre questi operatori (come in C, ma a differenza dalla matematica) non sono simmetrici. Più precisamente

```
A1 and A2 and ... and An
```

è uguale ad `An`, se tutti gli `Ai` sono veri, altrimenti il valore dell'espressione è il primo `Ai` a cui corrisponde il valore di verità falso. Ad esempio:

```
print 2 and 3 and 8
# 8
print 2 and [] and 7
# []
```

Similmente

```
A1 or A2 or ... or An
```

è uguale ad `An`, se nessuno degli `Ai` è vero, altrimenti è uguale al primo `Ai` che è vero:

```
print 0 or '' or []
# []
print 0 or [] or 2 or 5
# 2
```

Se per qualche ragione (ad esempio nella visualizzazione di uno stato) si desidera come risultato di queste operazioni un valore di verità, è sufficiente usare la funzione `bool`:

```
print bool(2 and 3 and 8)
# True
print bool(2 and [] and 7)
# False
print bool(0 or '' or [])
# False
print bool(0 or [] or 2 or 5)
# True
```

È molto importante nell'interpretazione procedurale degli operatori logici che in queste espressioni i termini che non servono non vengono nemmeno calcolati.

```
print math.log(0)
# Errore - il logaritmo di 0
# non e' definito.

print 1 or math.log(0)
# 1 - In questo caso il logaritmo
# non viene calcolato.
```

L'operatore di negazione logica `not` restituisce invece sempre un valore di verita:

```
print not []
# True

print not 5
# False
```

`not` lega più fortemente di `and` e questo più fortemente di `or`. Perciò le espressioni

```
(a and b) or c
(not a) and b
(not a) or b
```

possono essere scritte senza parentesi:

```
a and b or c
not a and b
not a or b
```

In questo numero

L'operatore di decisione

L'operatore ternario di decisione (o di diramazione) in C è rappresentato da un'espressione della forma `A ? x : y`.

Questa costituisce *un valore* che è uguale ad `x`, se `A` è vera, altrimenti uguale ad `y`.

L'operatore di diramazione è molto importante nell'informatica teorica (ad esempio nella tecnica dei *diagrammi binari di decisione* nello studio delle funzioni booleane) ed è spesso scritto nella forma $[A, x, y]$.

Per imitare questo operatore in Python definiamo la funzione

```
def iftern (A,x,y): # if ternario
  if A: return x
  else: return y
```

Saremmo tentati a utilizzare questa funzione per il calcolo del fattoriale con

```
def f (n): # Brutta sorpresa!
  return iftern(n==0,1,n*f(n-1))
```

ma quando usiamo questa funzione, ci aspetta una brutta sorpresa: l'interprete ci segnala che abbiamo innescato una ricorsione infinita, come se mancasse la condizione di terminazione. Ed è proprio così, perché non riusciamo a calcolare il fattoriale di 0. Infatti questo corrisponde a

```
iftern(True,1,0*f(-1)
```

e vediamo che dobbiamo calcolare

```
iftern(False,1,-1*f(-2)
```

ecc. Vediamo quindi che gli operatori logici della matematica, che sono simmetrici, non si prestano a un utilizzo procedurale. Per questa ragione nei linguaggi di programmazione gli operatori logici sono definiti nel modo non simmetrico che abbiamo visto in precedenza; su questa interpretazione procedurale dell'asimmetria degli operatori logici si basa la *programmazione logica*.

A pagina 32 del corso di Algoritmi abbiamo incontrato la funzione `ifelse` di R che può essere considerata una generalizzazione vettoriale di `iftern` a cui è equivalente nel caso di vettori di lunghezza 1.

```
x=(1,0,1,0,1,0,1,0,1,0,1,0)
a=(7,None,8,None,9,None)
b=(None,1,None,2)
u=r.ifelse(x,a,b)
print u
# [7, 1, 8, 2, 9, 1, 7, 2, 8, 1, 9, 2]
```

Logica procedurale

Esaminiamo ancora la funzione `f` dell'esempio precedente. La funzione fallisce perché cerca di definire la condizione di terminazione all'interno dell'espressione

```
iftern(n==0,1,n*f(n-1))
```

che richiede il calcolo di `f(n-1)` anche quando il risultato non è usato perché la condizione iniziale `n==0` è soddisfatta. Possiamo invece ridefinire la funzione nel modo seguente:

```
def fatt (n):
  return iftern(n==0,1,n and n*fatt(n-1))
```

Se proviamo la funzione, vediamo che viene calcolata correttamente. Esaminiamo separatamente i casi `n==0` e `n>0` (naturalmente l'algoritmo ricorsivo non può essere usato per valori negativi di `n`):

`n==0`: Bisogna calcolare non soltanto il risultato 1, ma anche l'espressione `0 and 0*fatt(-1)`. Siccome però 0 in un contesto logico è falso, la definizione asimmetrica di `and` implica che il secondo termine `0*fatt(-1)` non viene più calcolato e quindi non provoca errori.

`n>0`: In questo caso bisogna di nuovo calcolare 1 (benché non sia il risultato) e l'espressione `n and n*fatt(n-1)`. Siccome `n` stavolta è vero, perché diverso da zero, l'espressione è uguale ad `n*fatt(n-1)`. Infatti la regola per il calcolo di `and` implica (applicata al caso particolare di due argomenti) che `True and X` è sempre uguale ad `X`, indipendentemente dal valore di verità che ha `X`.

Diamo alcuni altri esempi per l'utilizzo procedurale degli operatori logici.

```
def positivo (x): return x>0 or False

def segno (x):
  return x>0 and 1 or x<0 and -1 or 0

def bin (n,k): # Numeri binomiali.
  return k==0 and 1 or n<k and 0
    or (float(n)/k)*bin(n-1,k-1)

def prod (a,b): # a(a+1) ... b
  return iftern(a>b,1,a<=b and
    iftern(a==b,a,a<b and b*prod(a,b-1)))

for b in xrange(2,7): print prod(3,b),
# 1 3 12 60 360
```

Giustificare gli algoritmi per `bin` e `prod`.

Per ridefinire `iftern` mediante gli operatori logici non possiamo utilizzare direttamente

```
def iftern (A,x,y): # Non corretto!
  return A and x or y
```

perché allora `iftern(True,0,y)` sarebbe uguale ad `y`, ma piuttosto

```
def iftern (A,x,y):
  return (A and [x] or [y])[0]
```

Si vede che l'uso procedurale degli operatori logici, nonostante l'importanza teorica di queste costruzioni, non è sempre facilmente leggibile; in pratica in Python è spesso preferibile l'utilizzo di `if ... else`.

if ... elif ... else

Le istruzioni condizionali in Python vengono utilizzate con la sintassi

```
if A: alfa()

if A: alfa()
else: beta()

if A:
  if B: alfa()
  else: beta()
else: gamma()
```

Non dimenticare i doppi punti. Spesso si incontrano diramazioni della forma

```
if A: alfa()
else:
  if B: beta()
  else:
    if C: gamma()
    else: delta()
```

In questi casi i doppi punti e la necessità delle indentazioni rendono la composizione del programma difficoltosa; è prevista perciò in Python l'abbreviazione `elif` per un `else ... if` come nell'ultimo esempio che può essere riscritto in modo più semplice:

```
if A: alfa()
elif B: beta()
elif C: gamma()
else: delta()
```

Esempio:

```
def segno (x):
  if x>0: return 1
  elif x==0: return 0
  else: return -1
```

Purtroppo il Python non prevede la costruzione `switch ... case` del C. Con un po' di attenzione la si può comunque emulare con l'impiego di una serie di `elif` oppure, come nel seguente esempio, mediante l'impiego adeguato di un dizionario:

```
operazioni = {'Roma' : 'print "Lazio"',
  'Ferrara' : 'print "Romagna"',
  'Cremona' : 'x=5; print x*x'}
for x in ['Roma','Ferrara','Cremona']:
  exec(operazioni[x])
```

`exec(a)` esegue la stringa `a` come se fosse un'istruzione del programma.

Operatori aritmetici

Interi possono avere un numero arbitrario di cifre. Il quoziente intero di due interi a e b lo si ottiene con `a/b`, quindi ad esempio `28/11` è uguale a 6, e il resto di a modulo b è dato da `a%b`. Purtroppo gli operatori `/` e `%`, come peraltro in C, non funzionano correttamente (dal punto di vista matematico) per $b < 0$, ad esempio `40%(-6)` è -2, mentre in matematica si vorrebbe che il resto r modulo b soddisfi $0 \leq r < |b|$. Una soluzione di questo problema si trova alle pagine 17-18 del corso di Programmazione 2004/05. Python fornisce anche la funzione `divmod` che restituisce la coppia `(a/b,a%b)`. Per ottenere il quoziente reale di due numeri interi bisogna evidenziare uno degli operandi come reale, come ad esempio in `20.0/7` oppure `float(20)/7`. Si può anche calcolare il quoziente intero di numeri reali (con lo stesso difetto indicato prima) mediante l'operatore `//` che è stato definito in Python 2.2 con lo scopo di poter ridefinire in futuro (probabilmente dalla versione 3.0) l'operatore `/` come operatore di divisione reale. Le potenze x^a (ad esponenti reali) si ottengono con `x**a`:

```
print 10**math.log10(17)
# 17.0
```

filter

Questa utilissima funzione con la sintassi `filter(f,v)` estrae da un vettore `v` tutti gli elementi `x` per cui `f(x)` è vera e restituisce come risultato il vettore che consiste di tutti questi elementi. In R filtri sono realizzati mediante vettori di indici logici (pag. 40 del corso di Algoritmi).

Dopo aver definito

```
def pari (x): return x%2==0
```

possiamo estrarre tutti i numeri pari da un vettore:

```
v=xrange(1,21)
print filter(pari,v)
# [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Similmente da un elenco di parole possiamo estrarre le parole che hanno lunghezza ≥ 5:

```
v=['Roma','Ferrara','Bologna','Pisa']
print filter(lambda x: len(x)>4,v)
# ['Ferrara', 'Bologna']
```

L'espressione `lambda x: f(x)` corrisponde alla nostra notazione $\bigcirc_x f(x)$. Filtri si usano anche nella gestione di basi di dati; possiamo ad esempio trovare tutti gli impiegati di una ditta che guadagnano almeno 3000 euro al mese:

```
imp = [['Rossi',2000],['Verdi',3000],
  ['Gentili',1800],['Bianchi',3400],
  ['Tosi',1600],['Neri',2800]]
imp3000=filter(lambda x: x[1]>=3000,imp)
for x in imp3000: print x[0],
# Verdi Bianchi
```

Il crivello di Eratostene

Imitiamo il programma in R (pagina 40 del corso di Algoritmi):

```
def Eratostene (n):
  v=range(2,n+1); u=[]
  r=math.sqrt(n); p=2
  while p<=r:
    p=v[0]; u.append(p)
    v=filter(lambda x: x%p>0,v)
  return u+v

v=Eratostene(100); m=len(v)
for i in xrange(0,m):
  if i%8==0: print
  print v[i],

# 2 3 5 7 11 13 17 19 23 29
# 31 37 41 43 47 53 59 61 67 71
# 73 79 83 89 97
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 3

for

La sintassi di base del `for` ha, come sappiamo, la forma

```
for x in v: istruzioni
```

dove `v` è una sequenza o un iteratore.
Dal `for` si esce con `break` (o naturalmente, dall'interno di una funzione, con `return`), mentre `continue` interrompe (come il `next` di R) il passaggio corrente di un ciclo e porta all'immediata esecuzione del passaggio successivo, cosicché

```
for x in xrange(0,21):
  if x%2>0: continue
  print x,
# 0 2 4 6 8 10 12 14 16 18 20
```

stampa sullo schermo i numeri pari compresi tra 1 e 20.

Il `for` può essere usato anche nella forma

```
for x in v: istruzioni
else: istruzionefinale
```

Quando le istruzioni nel `for` stesso non contengono un `break`, questa sintassi è equivalente a

```
for x in v: istruzioni
```

Un `break` invece *salta* la parte `else` che quindi viene eseguita solo se tutti i passaggi previsti nel ciclo sono stati effettuati. Questa costruzione viene utilizzata quando il percorso di tutti i passaggi è considerato come una condizione di eccezione: Assumiamo ad esempio che cerchiamo in una sequenza un primo elemento con una determinata proprietà - una volta trovato, usciamo dal ciclo e continuiamo l'elaborazione con questo elemento; se invece un elemento con quella proprietà non si trova, abbiamo una situazione diversa che trattiamo nel `else`. Nei due esempi che seguono cerchiamo il primo elemento positivo di una tupla di numeri:

```
for x in (-1,0,0,5,2,-3,4):
  if x>0: print x; break
else: print 'Nessun elemento positivo.'
# 5
```

```
for x in (-1,0,0,-5,-2,-3,-4):
  if x>0: print x; break
else: print 'Nessun elemento positivo.'
# Nessun elemento positivo.
```

Cicli `for` possono essere annidati; con

```
for i in xrange(4):
  for j in xrange(5):
    print i+j,
  print
```

otteniamo

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

Se gli elementi della sequenza che viene percorsa dal `for` sono a loro volta sequenze tutte della stessa lunghezza, nel `for` ci possiamo riferire agli elementi di queste sequenze con nomi di variabili:

```
u=[[1,10],[2,10],[3,10],[4,20]]
for x,y in u: print x+y,
print
# 11 12 13 24

v=['Aa','Bb','Cc','Dd','Ee']
for x,y in v: print y+'.'+x,
print
# a.A b.B c.C d.D e.E

w=[[1,2,5],[2,3,6],[11,10,9]]
for x,y,z in w: print x*y+z,
# 7 12 119
```

Combinando il `for` con `zip` possiamo calcolare il prodotto scalare di due vettori:

```
def prodottoscalare (u,v):
  s=0
  for x,y in zip(u,v): s+=x*y
  return s

u=[1,3,4]
v=[6,2,5]

print prodottoscalare(u,v)
# 32
```

while

Il `while` controlla cicli più generali del `for` e gli è molto simile nella sintassi:

```
while A: istruzioni
```

oppure

```
while A: istruzioni
else: istruzionefinale
```

`break` e `continue` vengono utilizzati come nel `for`. Se è presente un `else`, l'istruzione finale viene eseguita se l'uscita dal ciclo è avvenuta perché la condizione `A` non era più soddisfatta, mentre viene saltata se si è usciti con un `break` o un `return`.

```
x=0; v=[]
while not x in v:
  v.append(x)
  x=(7*x+13)%17

for x in v: print x,
print
# 0 13 2 10 15 16 6 4 7 11 5 14 9 8 1 3

while 1:
  nome=raw_input('Come ti chiami? ')
  if nome=='': break
  print 'Ciao, %s!' %(nome)
```

In questo numero

zip

Questa funzione utilissima corrisponde essenzialmente alla formazione della trasposta di una matrice. Esempio:

```
a=[1,2,3]; b=[11,12,13]
c=[21,22,23]; d=[31,32,33]
for x in zip(a,b,c,d): print x
# Output:

(1, 11, 21, 31)
(2, 12, 22, 32)
(3, 13, 23, 33)
```

Il risultato di `zip` è sempre una lista i cui elementi sono tuple. Quando gli argomenti non sono tutti della stessa lunghezza, viene usata la lunghezza minima, come in

```
a=[1,2,3,4]; b=[11,12]
c=[21,22,23,24]
for x in zip(a,b,c): print x
# Output:

(1, 11, 21)
(2, 12, 22)
```

Il tipo dei dati non ha importanza:

```
a=[0,1,2,3]; b=['a',7,'geo',[9,10]]
c=[11,12,13,14]
d=['A','B','C',['E','F']]
for x in zip(a,b,c,d): print x
# Output:

(0, 'a', 11, 'A')
(1, 7, 12, 'B')
(2, 'geo', 13, 'C')
(3, [9, 10], 14, ['E', 'F'])
```

Uso di `zip` con `for x,y`:

```
nomi=('Verdi','Rossi','Bianchi')
stipendi=(2000,1800,2700)
for x,y in zip(nomi,stipendi):
  print x,y
# Output:

Verdi 2000
Rossi 1800
Bianchi 2700
```

enumerate

Un'altra funzione utile è `enumerate` che da un vettore `v` genera un iteratore che corrisponde alle coppie `(i,v[i])`. Se trasformiamo l'iteratore in una lista, vediamo che l'oggetto che si ottiene è essenzialmente equivalente a `zip(xrange(len(v)),v)`:

```
v=['A','D','E','N']

e=enumerate(v)
print e
# <enumerate object at 0x1f726c>

a=list(enumerate(v))
b=zip(xrange(len(v)),v)
print a==b
# True

# Infatti:
print a
# [(0,'A'), (1,'D'), (2,'E'), (3,'N')]

print b
# [(0,'A'), (1,'D'), (2,'E'), (3,'N')]
```

`enumerate` è però più efficiente di `zip` e viene tipicamente usato quando nel percorrere un vettore bisogna tener conto sia del valore degli elementi sia della posizione in cui si trovano nel vettore. Assumiamo ad esempio che vogliamo calcolare la somma degli indici di quegli elementi in un vettore numerico che sono maggiori di 10:

```
v=[8,13,0,5,17,8,6,24,6,15,3]

s=0
for i,x in enumerate(v):
  if x>10: s+=i
print s
# 21, perche' sono maggiori di 10
# gli elementi con gli indici
# 1,4,7,9 e 1+4+7+9=21.
```

Possiamo con questa tecnica anche definire una funzione che calcola il valore di una somma trigonometrica

$$\sum_{n=0}^{N} a_n \cos nx$$

con

```
def sommatrigonometrica (a,x):
  s=0
  for n,an in enumerate(a):
    s+=an*math.cos(n*x)
  return s

a=[2,3,0,4,7,1,3]

print sommatrigonometrica(a,0.2)
# 14.7458647279
```

Nello stesso modo potremmo anche calcolare il valore $f(\alpha)$ di un polinomio $\sum_{n=0}^{N} a_n x^n$, ma vedremo che lo schema di Horner fornisce un algoritmo più efficiente sia per i polinomi che per le somme trigonometriche. Nonostante ciò anche in questo contesto `enumerate` può risultare utile, ad esempio per calcolare somme della forma generale $\sum_{n=0}^{N} \varphi(n,x)$.

map semplice

f sia una funzione (o un'espressione `lambda`) di un singolo argomento, a una sequenza o un iteratore con gli elementi $a_0, \ldots, a_n$. Allora `map(f,a)` è la lista $[f(a_0), \ldots, f(a_n)]$. Esempi:

```
a='ABCDabcd'
print map(ord,a)
# [65, 66, 67, 68, 97, 98, 99, 100]
```

Infatti, per un carattere `x` si ottiene con `ord(x)` il suo codice ASCII.

```
u=map(lambda x: x**2, xrange(10))
print u
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

L'esempio che segue esprime in poche righe la potenza di Python:

```
def vocale (x):
  return x.lower() in 'aeiou'

print map(vocale,'Crema')
# [False, False, True, False, True]
```

Usiamo `map` per costruire il grafico di una funzione:

```
def grafico (f,a):
  return zip(a,map(f,a))

a=(0,1,2,3)
print grafico(lambda x: x**4,a)
# [(0, 0), (1, 1), (2, 16), (3, 81)]
```

La funzione `str` trasforma un oggetto in una stringa e, utilizzata insieme a `join`, permette ad esempio di trasformare una lista di numeri in una stringa i cui i numeri appaiono separati da spazi:

```
a=(88,20,17,4,58)
v=map(str,a)
print v
# ['88', '20', '17', '4', '58']

print ' '.join(v)
# 88 20 17 4 58
```

Se X è un insieme, la diagonale n-esima di X è il sottoinsieme di X^n che consiste delle tuple della forma $(x, \ldots, x)$:

```
def diagonale (a,n=2):
  return map(lambda x: (x,)*n,a)
# Non dimenticare la virgola.

a=(2,3,5,6)
print diagonale(a)
# [(2, 2), (3, 3), (5, 5), (6, 6)]
```

Alcune regolarità del codice genetico possono essere studiate meglio se le lettere G,A,C,T vengono sostituite con 0,1,2,3:

```
def dnanumerico (x):
  x=x.lower()
  sost={'g': 0, 'a': 1, 'c': 2, 't': 3}
  return sost[x]

g='TGAATGCTAC'
print map(dnanumerico,g)
# [3, 0, 1, 1, 3, 0, 2, 3, 1, 2]
```

Cesare codificava talvolta i suoi messaggi sostituendo ogni lettera con la lettera che si trova a tre posizioni dopo la lettera originale, calcolando le posizioni in maniera ciclica. Assumiamo di avere un alfabeto di 26 lettere (senza minuscole, spazi o interpunzioni):

```
def cesare (a):
  o=ord('A')
  def codifica(x):
    n=(ord(x)-o+3)%26
    return chr(o+n)
  v=map(codifica,a)
  return ''.join(v)

a='CRASCASTRAMOVEBO'
print cesare(a)
# FUDVFDVWUDPRYHER
```

I `map` e i `lambda` possono essere annidati. Per costruire una matrice identica possiamo usare (non è questo il modo più efficiente) la seguente costruzione:

```
a=map(lambda i:
  map (lambda j:
    i==j,xrange(4)), xrange(4))
for x in a: print x
```

ottenendo

```
[True, False, False, False]
[False, True, False, False]
[False, False, True, False]
[False, False, False, True]
```

Addizionando 0 a `True` o `False` otteniamo il valore 1 o 0 corrispondente. Modificando leggermente la funzione precedente, troviamo quindi la matrice identica numerica:

```
a=map(lambda i:
  map (lambda j:
    (i==j)+0,xrange(4)), xrange(4))
for x in a: print x
```

con output

```
[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
```

Usando la funzione `rapp2` che sarà definita a pagina 9 possiamo ottenere un ipercubo:

```
def ipercubo (m):
  def rapp(x): return rapp2(x,cifre=m)
  return map(rapp,xrange(2**m))

for x in ipercubo(4): print x # Output:

[0, 0, 0, 0]
[0, 0, 0, 1]
[0, 0, 1, 0]
[0, 0, 1, 1]
[0, 1, 0, 0]
[0, 1, 0, 1]
[0, 1, 1, 0]
[0, 1, 1, 1]
[1, 0, 0, 0]
[1, 0, 0, 1]
[1, 0, 1, 0]
[1, 0, 1, 1]
[1, 1, 0, 0]
[1, 1, 0, 1]
[1, 1, 1, 0]
[1, 1, 1, 1]
```

map multivariato

$a = (a_0, \ldots, a_n)$ e $b = (b_0, \ldots, b_n)$ siano due sequenze o iteratori della stessa lunghezza ed f una funzione di 2 argomenti. Allora `map(f,u,v)` è la lista

$$[f(a_0, b_0), \ldots, f(a_n, b_n)]$$

In modo simile sono definite le espressioni `map(f,u,v,w)` per funzioni di tre argomenti ecc. Esempi:

```
def somma (*a): # Come a pagina 2.
  s=0
  for x in a: s+=x
  return s

print map(somma,
  (0,1,5,4),(2,0,3,8),(6,2,2,7))
# [8, 3, 10, 19]
```

`None`, pur non essendo una funzione, nel `map` è considerata come la funzione identica:

```
print map(None,(1,2,3))
# [1, 2, 3]

print map(None,(11,12,13),(21,22,23))
# [(11, 21), (12, 22), (13, 23)]

v=map(None,
  (11,12,13),(21,22,23),(31,32,33))
for r in v: print r
```

L'ultimo output è

```
(11, 21, 31)
(12, 22, 32)
(13, 23, 33)
```

Vediamo che `map(None,...)` è equivalente all'utilizzo di `zip`. Un altro esempio:

```
a=(2,3,5,0,2); b=(1,3,8,0,4)

def f(x,y): return (x+y,x-y)

for x in map(f,a,b): print x,
# (3,1) (6,0) (13,-3) (0,0) (6,-2)
```

map implicito

Il Python prevede una costruzione che permette spesso di sostituire il `map` con un `for`. Se `f` è una funzione in una variabile,

```
[f(x) for x in a]
```

è equivalente a `map(f,a)`; similmente per una funzione di due variabili

```
[f(x,y) for x,y in ((a1,b1),(a2,b2),...)]
```

corrisponde a `map(f,a,b)`. Questa forma è spesso più intuitiva e talvolta più breve, soprattutto nel caso delle espressioni lambda sufficientemente semplici:

```
a=('Roma','Pisa','Milano','Trento')
print [x[0] for x in a]
# ['R', 'P', 'M', 'T']

a=[x*x+3 for x in xrange(8)]
print a
# [3, 4, 7, 12, 19, 28, 39, 52]

a=[x+y for x,y in ((3,2),(5,6),(1,9))]
print a
# [5, 11, 10]
```

Rappresentazione binaria

Ogni numero naturale n possiede una rappresentazione binaria, cioè una rappresentazione della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_1 2 + a_0$$

con coefficienti (o cifre binarie) $a_i \in \{0, 1\}$. Per $n = 0$ usiamo $k = 0$ ed $a_0 = 0$; per $n > 0$ chiediamo che $a_k \neq 0$. Con queste condizioni k e gli a_i sono univocamente determinati. Sia $r_2(n) = (a_k, \ldots, a_0)$ il vettore i cui elementi sono queste cifre. Dalla rappresentazione binaria si deduce la seguente relazione ricorsiva:

$$r_2(n) = \begin{cases} (n) & \text{se } n \leq 1 \\ (r_2(\frac{n}{2}), 0) & \text{se } n \text{ è pari} \\ (r_2(\frac{n-1}{2}), 1) & \text{se } n \text{ è dispari} \end{cases}$$

È molto facile tradurre questa idea in una funzione di Python. Siccome Python per operandi interi esegue una divisione intera, anche per n dispari possiamo scrivere `n/2`, in tal caso automaticamente uguale a `(n-1)/2`.

```
def rapp2pv (n): # Prima versione.
  if n<=1: v=[n]
  else:
    v=rapp2pv(n/2)
    if n%2==0: v.append(0)
    else: v.append(1)
  return v
```

La versione definitiva della funzione prevede un secondo parametro facoltativo `cifre`; quando questo è maggiore del numero di cifre necessarie per la rappresentazione binaria di n, i posti iniziali vuoti vengono riempiti con zeri.

```
def rapp2 (n,cifre=0):
  if n<=1: v=[n]
  else:
    v=rapp2(n/2)
    if n%2==0: v.append(0)
    else: v.append(1)
  d=cifre-len(v)
  if d>0: v=[0]*d+v
  return v
```

Per provare la funzione usiamo la possibilità di costruire una stringa da una lista di numeri mediante la funzione `str`, come abbiamo visto in precedenza, ottenendo l'output

```
  0  0 0 0 0 0 0 0 0
  1  0 0 0 0 0 0 0 1
  2  0 0 0 0 0 0 1 0
  3  0 0 0 0 0 0 1 1
  4  0 0 0 0 0 1 0 0
  5  0 0 0 0 0 1 0 1
  6  0 0 0 0 0 1 1 0
  7  0 0 0 0 0 1 1 1
  8  0 0 0 0 1 0 0 0
  9  0 0 0 0 1 0 0 1
 10  0 0 0 0 1 0 1 0
 11  0 0 0 0 1 0 1 1
 12  0 0 0 0 1 1 0 0
 19  0 0 0 1 0 0 1 1
 48  0 0 1 1 0 0 0 0
 77  0 1 0 0 1 1 0 1
106  0 1 1 0 1 0 1 0
135  1 0 0 0 0 1 1 1
164  1 0 1 0 0 1 0 0
194  1 1 0 0 0 0 1 0
221  1 1 0 1 1 1 0 1
```

con

```
def strdalista (a,sep=' '):
  return sep.join(map(str,a))

for n in range(13)+ \
  [19,48,77,106,135,164,194,221]:
  print "%3d  %s" \
    %(n,strdalista(rapp2(n,cifre=8)))
```

Se usiamo invece

```
for n in xrange(256):
  print "%3d  %s" \
    %(n,strdalista(rapp2(n,cifre=8)))
```

otteniamo gli elementi dell'ipercubo 2^8 (cfr. pagina 10).

Numeri esadecimali

Nei linguaggi macchina e assembler molto spesso si usano i numeri esadecimali o, più correttamente, la rappresentazione esadecimale dei numeri naturali, cioè la loro rappresentazione in base 16.

Per $2789 = 10 \cdot 16^2 + 14 \cdot 16 + 5 \cdot 1$ potremmo ad esempio scrivere

`2789 = (10,14,5)`$_{16}$.

In questo senso `10`,`14` e `5` sono le cifre della rappresentazione esadecimale di `2789`. Per poter usare lettere singole per le cifre si indicano le cifre `10`, ..., `15` mancanti nel sistema decimale nel modo seguente:

```
10   A
11   B
12   C
13   D
14   E
15   F
```

In questo modo adesso possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che `(F)`$_{16}$ `= 15` assume nel sistema esadecimale lo stesso ruolo come il `9` nel sistema decimale. Quindi `(FF)`$_{16}$ `=` $255 = 16^2 - 1 = 2^8 - 1$. Un numero naturale n con $0 \leq n \leq 255$ si chiama un *byte*, un *bit* è invece uguale a 0 o a 1.

Esempi:

	0	$(0)_{16}$
	14	$(E)_{16}$
	15	$(F)_{16}$
	16	$(10)_{16}$
	28	$(1C)_{16}$
2^5	32	$(20)_{16}$
2^6	64	$(40)_{16}$
	65	$(41)_{16}$
	97	$(61)_{16}$
	127	$(7F)_{16}$
2^7	128	$(80)_{16}$
	203	$(CB)_{16}$
	244	$(F4)_{16}$
	255	$(FF)_{16}$
2^8	256	$(100)_{16}$
2^{10}	1024	$(400)_{16}$
2^{12}	4096	$(1000)_{16}$
	65535	$(FFFF)_{16}$
2^{16}	65536	$(10000)_{16}$

Si vede da questa tabella che i byte sono esattamente quei numeri per i quali sono sufficienti due cifre esadecimali.

L'ipercubo

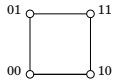
Sia $X = \{1, \ldots, n\}$ con $n \geq 0$.

Identificando $\mathcal{P}(X)$ con 2^n, geometricamente otteniamo un *ipercubo* che può essere visualizzato nel modo seguente.

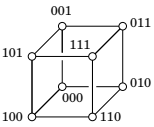
$n = 0$:

$n = 1$:

$n = 2$:



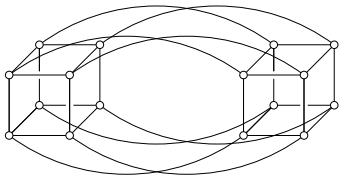
$n = 3$:



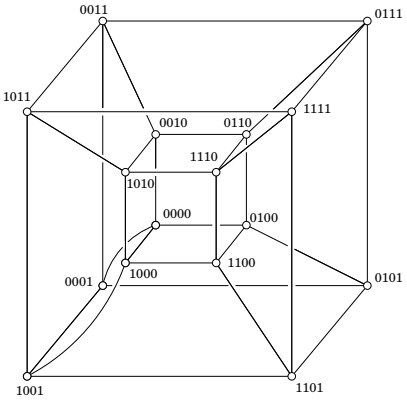
$n = 4$: L'ipercubo a 4 dimensioni si ottiene dal cubo 3-dimensionale attraverso la relazione

$$2^4 = 2^3 \times \{0, 1\}$$

Dobbiamo quindi creare due copie del cubo 3-dimensionale. Nella rappresentazione grafica inoltre sono adiacenti e quindi connessi con una linea quei vertici che si distinguono in una sola coordinata. Oltre ai legami all'interno dei due cubi dobbiamo perciò unire i punti $(x, y, z, 0)$ e $(x, y, z, 1)$ per ogni x, y, z.



La figura diventa molto più semplice, se si pone uno dei due cubi (quello con la quarta coordinata $= 0$) all'interno dell'altro (quello con la quarta coordinata $= 1$):



$n \geq 5$: Teoricamente anche qui si può usare la relazione

$$2^n = 2^{n-1} \times \{0, 1\}$$

ma la visualizzazione diventa difficoltosa.

Ogni vertice dell'ipercubo corrisponde a un elemento di 2^n che nell'interpretazione insiemistica rappresenta a sua volta un sottoinsieme di X (il punto 0101 ad esempio l'insieme $\{2, 4\}$ se $X = \{1, 2, 3, 4\}$).

La distanza di Hamming

La distanza di Hamming è definita come il numero delle coordinate in cui due elementi di 2^n differiscono e che in Python può essere calcolata con la seguente semplicissima funzione:

```
def hamming (a,b):
  return map(lambda x,y:
    x==y and 1 or 0, a,b).count(1)

a=(0,1,0,1,0,1,1,1,0,1)
b=(1,0,0,1,0,0,1,1,1,0)

print hamming(a,b)
# 5
```

Questa distanza ha le proprietà di una metrica ed è molto utilizzata nella teoria dei codici.

Lo schema di Horner

Sia dato un polinomio

$$f = a_0 x^n + a_1 x^{n-1} + \cdots + a_n \in A[x]$$

dove A è un qualsiasi anello commutativo.

Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$.

Sia ad esempio $f = 3x^4 + 5x^3 + 6x^2 + 8x + 7$. Poniamo

$$\begin{aligned}
b_0 &= 3 \\
b_1 &= b_0\alpha + 5 = 3\alpha + 5 \\
b_2 &= b_1\alpha + 6 = 3\alpha^2 + 5\alpha + 6 \\
b_3 &= b_2\alpha + 8 = 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\
b_4 &= b_3\alpha + 7 = 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 7
\end{aligned}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{aligned}
b_0 &= a_0 \\
b_1 &= b_0\alpha + a_1 \\
&\ldots \\
b_k &= b_{k-1}\alpha + a_k \\
&\ldots \\
b_n &= b_{n-1}\alpha + a_n
\end{aligned}$$

con $b_n = f(\alpha)$, come dimostriamo adesso. Consideriamo il polinomio

$$g := b_0 x^{n-1} + b_1 x^{n-2} + \cdots + b_{n-1}.$$

Allora, usando che $\alpha b_k = b_{k+1} - a_{k+1}$ per $k = 0, \ldots, n-1$, abbiamo

$$\begin{aligned}
\alpha g &= \alpha b_0 x^{n-1} + \alpha b_1 x^{n-2} + \cdots + \alpha b_{n-1} \\
&= (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \ldots \\
&\quad + (b_{n-1} - a_{n-1})x + b_n - a_n \\
&= (b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1}x + b_n) \\
&\quad - (a_1 x^{n-1} + a_2 x^{n-2} + \ldots a_{n-1}x + a_n) \\
&= x(g - b_0 x^{n-1}) + b_n - (f - a_0 x^n) \\
&= xg - b_0 x^n + b_n - f + a_0 x^n \\
&= xg + b_n - f
\end{aligned}$$

quindi

$$\boxed{f = (x - \alpha)g + b_n}$$

e ciò implica $f(\alpha) = b_n$.

$b_0, \ldots, b_{n-1}$ sono perciò i coefficienti del quoziente nella divisione con resto di f per $x - \alpha$, mentre b_n è il resto, uguale a $f(\alpha)$.

Questo algoritmo è detto *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece semplicemente l'operazione `x**n`.

Abbiamo ricordato l'algoritmo di Horner già a pagina 2; modifichiamo leggermente la funzione che vogliamo usare in Python:

```
def horner (a,x):
  b=0
  for ak in a: b=b*x+ak
  return b
```

Una frequente applicazione dello schema di Horner è il calcolo del valore corrispondente a una rappresentazione binaria o esadecimale.

Infatti otteniamo $(1, 0, 0, 1, 1, 0, 1, 1, 1)_2$ come

```
horner([1,0,0,1,1,0,1,1,1],2)
```

e $(A, F, 7, 3, 0, 5, E)_{16}$ come

```
horner([10,15,7,3,0,5,14],16):

x=horner([1,0,0,1,1,0,1,1,1],2)
print x # 311

y=horner([10,15,7,3,0,5,14],16)
print y # 183971934
```

Somme trigonometriche

Per calcolare somme trigonometriche della forma $\sum_{n=0}^{N} a_n \cos nx$ possiamo usare lo schema di Horner, ottenendo così un algoritmo notevolmente più efficiente del calcolo diretto visto a pagina 8. La rappresentazione

$$\cos x = \frac{e^{ix} + e^{-ix}}{2}$$

(Algoritmi, pag. 14) ci permette infatti di scrivere la somma nella forma

$$\frac{1}{2}\left(\sum_{n=0}^{N} a_n e^{inx} + \sum_{n=0}^{N} a_n e^{-inx}\right)$$

Ponendo $z_1 := e^{ix}$ e $z_2 := e^{-ix}$, la somma diventa quindi

$$\frac{1}{2}\left(\sum_{n=0}^{N} a_n z_1^n + \sum_{n=0}^{N} a_n z_2^n\right)$$

che può esser calcolata con lo schema di Horner. Per l'esponenziale complesso utilizziamo il modulo `cmath` di Python. A differenza da R i numeri complessi in Python sono scritti nella forma `a+bj`.

```
def sommacoseni (a,x):
  a=a[:]; a.reverse(); ix=x*1j
  z1=cmath.exp(ix); z2=cmath.exp(-ix)
  return (horner(a,z1)+horner(a,z2))/2

a=[2,3,0,4,7,1,3]

print sommacoseni(a,0.2)
# (14.7458647279+0j)
```

Lo sviluppo di Taylor

Sia di nuovo dato un polinomio

$$f = a_0x^n + \ldots + a_n$$

Fissato α, con lo schema di Horner otteniamo i valori $b_0, \ldots, b_n$ che sono tali che $b_n = f(\alpha)$ e

$$g := b_0x^{n-1} + b_1x^{n-2} + \ldots + b_{n-1}$$

è il quoziente nella divisione con resto di f per $x - \alpha$. Ponendo $f_0 := f$ ed $f_1 := g$ possiamo scrivere

$$f_0 = (x - \alpha)f_1 + b_n$$

Se f ha grado 1, allora f_1 è costante e la rappresentazione ottenuta è lo sviluppo di Taylor di f. Altrimenti possiamo applicare lo schema di Horner (con lo stesso α) ad f_1, ottenendo valori $c_0, \ldots, c_{n-1}$ tale che con $f_2 := c_0x^{n-2} + \ldots + c_{n-2}$ si abbia

$$f_1 = (x - \alpha)f_2 + c_{n-1}$$

cosicché

$$f_0 = (x - \alpha)^2 f_2 + (x - \alpha)c_{n-1} + b_n$$

Se f ha grado 2, questo è lo sviluppo di Taylor, altrimenti continuiamo, ottenendo valori $d_0, \ldots, d_{n-2}$ tali che con $f_3 := d_0x^{n-3} + \ldots + d_{n-3}$ si abbia

$$f_2 = (x - \alpha)f_3 + d_{n-2}$$

cosicché

$$f_0 = (x - \alpha)^3 f_3 + (x - \alpha)^2 d_{n-2} + (x - \alpha)c_{n-1} + b_n$$

Se f ha grado 3, questo è lo sviluppo di Taylor, altrimenti continuiamo nello stesso modo.

Per calcolare lo sviluppo di Taylor con una funzione in Python dobbiamo quindi prima definire una funzione `hornercompleto` che calcola non solo il valore b_n, ma tutti i b_k. Da essa otteniamo la funzione `taylor` che ripete il procedimento fino a quando il vettore dei coefficienti è vuoto:

```
def hornercompleto (a,x):
  b=0; v=[]
  for ak in a: b=b*x+ak; v.append(b)
  return v

def taylor (a,x):
  v=a # Corretto, ma cfr. terza colonna.
  w=[]
  while v:
    v=hornercompleto(v,x)
    w.append(v.pop())
  return w

a=(3,5,6,8,7)
print taylor(a,2)
# [135, 188, 108, 29, 3]
```

Lo sviluppo di Taylor di

$$f = 3x^4 + 5x^3 + 6x^2 + 8x + 7$$

per $x = 2$ è quindi

$$f = 135 + 188(x - 2) + 108(x - 2)^2 + 29(x - 2)^3 + 3(x - 2)^4$$

Esercizio: Calcolare le derivate $f^{(k)}(2)$.

Argomenti opzionali

Gli argomenti di una funzione in Python nella chiamata della funzione possono essere indicati con i nomi utilizzati nella definizione, permettendo in tal modo di modificare l'ordine in cui gli argomenti appaiono:

```
def f(x,y): return x+2*y

print f(y=2,x=7)
# 11
```

Quando argomenti con nome nella chiamata vengono utilizzati insieme ad argomenti senza nomi, questi ultimi vengono identificati per la loro posizione; ciò implica che argomenti senza nomi devono sempre precedere eventuali argomenti con nome:

```
def f(x,y,z): return x+2*y+3*z

print f(2,z=4,y=1)
# 16
```

Un argomento a cui già nella definizione viene assegnato un valore, è opzionale e può essere tralasciato nelle chiamate della funzione:

```
def f(x,y,z=4): return x+2*y+3*z

print f(2,1)
# 16

print f(2,1,5)
# 19

print f(2,z=5,y=1)
# 19

def tel (nome,numero,prefisso='0532'):
  return [nome,prefisso+'-'+numero]

print tel('Rossi','974002')
# ['Rossi', '0532-974002']
```

Abbiamo già utilizzato questa possibilità nelle definizioni delle funzioni `diagonale` (pagina 8) e `rapp2` (pagina 9).

Bisogna qui stare attenti al fatto che il valore predefinito viene assegnato solo in fase di definizione; quando questo valore è mutabile, ad esempio una lista, in caso di più chiamate della stessa funzione viene utilizzato ogni volta il valore fino a quel punto raggiunto:

```
def f(x,v=[]): v.append(x); print v

f(7)
# [7]

f(8)
# [7, 8]
```

Ciò è in accordo con il comportamento descritto nella terza colonna su questa pagina. Invece

```
def f(x,y=3): print x+y; y+=1

f(1)
# 4

f(1)
# 4 - perche' y non e' mutabile.
```

Una trappola pericolosa

Abbiamo già visto a pagina 4 che i nomi del Python devono essere considerati come nomi di puntatori nel C e che quindi un'assegnazione `b=a` dove `a` è un nome (e non una costante) implica che `b` ed `a` sono nomi diversi per lo stesso oggetto. Ciò vale (come per i puntatori del C!) anche all'interno di funzioni, per cui una funzione può effettivamente modificare il valore dei suoi argomenti (più precisamente il valore degli oggetti a cui i nomi degli argomenti corrispondono). Mentre in C, se il programmatore ha davanti agli occhi la memoria su cui sta lavorando, ciò è piuttosto evidente e comprensibile, la sintattica semplice del Python induce facilmente a dimenticare questa circostanza.

```
def f(x,a):
  b=a; b.append(x); return b

a=[1,2]

f(4,a)
print a
# [1, 2, 4]

f(5,a)
print a
# [1, 2, 4, 5]
```

Nella funzione `taylor` su questa pagina abbiamo utilizzato semplicemente un'assegnamento `v=a`. Ciò è corretto, ma soltanto perché successivamente `v` viene riassegnato al valore di `hornercompleto` prima che entrino in azione i `v.pop()` del ciclo che modificano solo il nuovo `v` e non `a`. Comunque bisogna stare molto attenti in questi casi e soltanto la particolare forma dell'algoritmo rende corretta l'assegnamento semplice; altrimenti avremmo dovuto usare `b=a[:]` o addirittura `b=copy.deepcopy(a)`.

L'istruzione def

Per il suo formato particolare si sarebbe indotti a credere che la definizione di una funzione mediante un'istruzione

```
def f (x): ...
```

assomigli nel significato alla definizione di una funzione in C e che quindi le funzioni vengano create prima dell'esecuzione del programma, cosicché in particolare una tale definizione non possa essere ripetuta con lo stesso nome. Non è così invece: Ogni `def` è eseguito dall'interprete nel punto in cui si trova nel programma ed è piuttosto equivalente a un'istruzione `f = function (x) ...` di R. Non soltanto i `def` possono essere annidati, come abbiamo visto a pagina 2, ma lo stesso nome può essere riutilizzato in un altro `def`:

```
def f(x): print x+3
f(7)
# 10

def f(x,y): print x+y
f(3,9)
# 12
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 4

Espressioni lambda

L'espressione `lambda x: f(x)` corrisponde all'espressione `function (x) f(x)` di R; con più variabili diventa `lambda x,y: f(x,y)`. A differenza da R però in Python `f(x)` deve essere un'espressione unica che soprattutto non può contenere istruzioni di assegnamento o di controllo; manca anche il `return`. Ciononostante il valore di un'espressione lambda è una funzione che può essere usata come valore di un'altra funzione:

```
def u(x): return x**2

def v(x): return 4*x+1

def comp (f,g):
  return lambda x: f(g(x))

print comp(u,v)(5)
# 441
```

Qui abbiamo ridefinito la funzione `comp` vista a pagina 2. Possiamo anche assegnare un nome alla funzione definita da un'espressione lambda:

```
f=lambda x,y: x+y-1

print f(6,2)
# y
```

Come abbiamo visto a pagina 8, i `lambda` possono essere annidati. Un'espressione lambda senza variabile è una funzione costante:

```
def costante (x): return lambda : x

f=costante(10); print f()
# 10
```

Espressioni lambda vengono spesso usate come argomenti di altre funzioni, ad esempio possiamo creare un filtro che restituisce i numeri pari di una successione di interi senza definire un'apposita funzione `pari` (cfr. pagina 6):

```
print filter(lambda x: x%2==0,
  xrange(15))
# [0, 2, 4, 6, 8, 10, 12, 14]
```

Per uno spazio vettoriale V possiamo definire un'iniezione canonica

$$j := \bigcirc_{\alpha} \alpha(v) : V \longrightarrow V''$$

di V nel suo doppio duale; se V è di dimensione finita, j è un isomorfismo. Possiamo imitare j in Python con

```
def incan (v):
  def veval (a): return a(v)
  return veval
```

oppure più brevemente con

```
def incan (v):
  return lambda a: a(v)
```

Il λ-calcolo

Siccome in matematica bisogna distinguere tra la funzione f e i suoi valori $f(x)$, nel corso di Algoritmi abbiamo introdotto la notazione $\bigcirc_x f(x)$ per la funzione che manda x in $f(x)$. Ad esempio $\bigcirc_x \sin(x^2+1)$ è la funzione che manda x in $\sin(x^2+1)$. È chiaro che ad esempio $\bigcirc_x x^2 = \bigcirc_y y^2$ (per la stessa ragione per cui $\sum_{i=0}^{n} a_i = \sum_{j=0}^{n} a_j$), mentre $\bigcirc_x xy \neq \bigcirc_y yy$ (così come $\sum_i a_{ij} \neq \sum_j a_{jj}$) e, come non ha senso l'espressione $\sum_i \sum_i a_{ii}$, così non ha senso $\bigcirc_x \bigcirc_x x$. Siccome in logica si scrive $\lambda x.f(x)$ invece di $\bigcirc_x f(x)$, la teoria molto complicata di questo operatore si chiama λ-calcolo. Su esso si basano il Lisp e molti concetti dell'intelligenza artificiale.

H. Barendregt: The lambda calculus. Elsevier 1990.

Scrivere su più righe

Siccome Python usa l'indentazione per strutturare il testo sorgente, per poter scrivere un'istruzione su più righe, dobbiamo indicarlo all'interprete. Ciò avviene ponendo un \ alla fine di ogni riga che si vuole continuare su quella successiva. Il simbolo \ non è necessario, quando si distribuiscono su più righe parametri separati da virgole.

```
# Qui \ e' necessario.
print x+3*x*x+math.log(4*x+\
  math.sqrt(1+x*x))

# Non e' richiesto un \.
print f(math.log(x),math.log(y),
  x+y+z)
```

Abbiamo già visto che testi su più righe possono essere racchiusi tra apici o virgolette triplici:

```
a='''Taciti, soli, sanza compagnia
n'andavam, l'un dinanzi e l'altro dopo,
come i frati minor vanno per via.
...
La' giu' trovammo una gente dipinta,
che giva intorno assai con lenti passi
```

Come si vede, un apice singolo può apparire all'interno di apici triplici, bisogna però evitare un `''''` finale, perché allora i primi tre apici verrebbero letti come fine della stringa lunga.

In questo numero

Aiuto con help

Python fornisce numerosi strumenti per esplorare il linguaggio stesso. Le più importanti sono `help` e `dir`. Quest'ultimo comando permette di esaminare tutti i componenti di un oggetto predefinito o definito dall'utente, come vedremo più avanti. `help` può essere usato sia come comando normale all'interno di un programma: dopo aver importato il modulo `math` con `help(math)` otteniamo una stringa che contiene le informazioni sul modulo, mentre dopo `x=7` con `help(x)` otteniamo le informazioni su `x`. Per uno studio approfondito si può invece, tramite il comando `help()` dal terminale di Python, entrare nella modalità di aiuto; a questo punto basta battere il nome di un oggetto o di un argomento (ad esempio `math`, `topics`) per vedere le rispettive informazioni. Per `math` otteniamo ad esempio una breve descrizione degli oggetti compresi nel modulo `math`. Dall'aiuto interattivo si esce con `Ctrl D`.

```
Help on module math:

NAME
  math

FILE
  /usr/local/lib/python2.4/lib-dynload/math.so

MODULE DOCS
  http://www.python.org/doc/current/lib/module-math.html

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.

  asin(...)
      asin(x)

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

atan(...)
       atan(x)

       Return the arc tangent (measured in radians) of x.
...
```

Per uscire da Python dall'interno di un programma si usa `sys.exit(0)`. Questa funzione viene spesso utilizzata in programmi interattivi.

Il massimo comune divisore

Tutti i numeri $a, b, c, d, \ldots$ considerati sono interi, cioè elementi di $\mathbb{Z}$. Usiamo l'abbreviazione $\mathbb{Z}d := \{nd \mid n \in \mathbb{Z}\}$.

Diciamo che a è un *multiplo* di d se $a \in \mathbb{Z}d$, cioè se esiste $n \in \mathbb{Z}$ tale che $a = nd$. In questo caso diciamo anche che d *divide* a o che d è un *divisore* di a e scriviamo $d|a$.

Definizione 1. Per $(a, b) \neq (0, 0)$ il *massimo comune divisore* di a e b, denotato con $\mathrm{mcd}(a, b)$, è il più grande $d \in \mathbb{N}$ che è un comune divisore di a e b, cioè tale che $d|a$ e $d|b$. Poniamo invece $\mathrm{mcd}(0, 0) := 0$. In questo modo $\mathrm{mcd}(a, b)$ è definito per ogni coppia (a, b) di numeri interi.

Perché esiste $\mathrm{mcd}(a, b)$? Per $(a, b) = (0, 0)$ è uguale a 0 per definizione. Assumiamo che $(a, b) \neq (0, 0)$. Adesso $1|a$ e $1|b$ e se $d|a$ e $d|b$ e ad esempio $a \neq 0$, allora $d \leq |a|$, per cui vediamo che esiste solo un numero finito (al massimo $|a|$) di divisori comuni ≥ 1, tra cui uno ed uno solo deve essere il più grande. $\mathrm{mcd}(a, b)$ è quindi univocamente determinato e uguale a 0 se e solo se $a = b = 0$. Si noti che $d|a \iff -d|a$, per cui possiamo senza perdita di informazioni assumere che $d \in \mathbb{N}$.

L'algoritmo euclideo

Questo algoritmo familiare a tutti e apparentemente a livello solo scolastico, è uno dei più importanti della matematica ed ha numerose applicazioni: in problemi pratici (ad esempio nella grafica al calcolatore), in molti campi avanzati della matematica (teoria dei numeri e analisi complessa), nell'informatica teorica. L'algoritmo euclideo si basa sulla seguente osservazione (*lemma di Euclide*):

Lemma 2. *Siano a, b, c, q, d numeri interi e $a = qb + c$. Allora*

$$(d|a \text{ e } d|b) \iff (d|b \text{ e } d|c)$$

Quindi i comuni divisori di a e b sono esattamente i comuni divisori di b e c. In particolare le due coppie di numeri devono avere lo stesso massimo comune divisore: $\mathrm{mcd}(a, b) = \mathrm{mcd}(b, c)$.

Dimostrazione. Se $d|a$ e $d|b$, cioè $dx = a$ e $dy = b$ per qualche x, y, allora $c = a - qb = dx - qdy = d(x - qy)$ e vediamo che $d|c$.
E viceversa.

Calcoliamo $d := \mathrm{mcd}(7464, 3580)$:

$$\begin{aligned}
7464 &= 2{\cdot}3580 + 304 &&\Longrightarrow d = \mathrm{mcd}(3580, 304)\\
3580 &= 11{\cdot}304 + 236 &&\Longrightarrow d = \mathrm{mcd}(304, 236)\\
304 &= 1{\cdot}236 + 68 &&\Longrightarrow d = \mathrm{mcd}(236, 68)\\
236 &= 3{\cdot}68 + 32 &&\Longrightarrow d = \mathrm{mcd}(68, 32)\\
68 &= 2{\cdot}32 + 4 &&\Longrightarrow d = \mathrm{mcd}(32, 4)\\
32 &= 8{\cdot}4 + 0 &&\Longrightarrow d = \mathrm{mcd}(4, 0) = 4
\end{aligned}$$

Si vede che il massimo comune divisore è l'ultimo resto diverso da 0 nell'algoritmo euclideo. L'algoritmo in Python è molto semplice (dobbiamo però prima convertire i numeri negativi in positivi):

```
def mcd (a,b):
  if a<0: a=-a
  if b<0: b=-b
  while b: a,b=b,a%b
  return a
```

Altrettanto semplice è la versione ricorsiva:

```
def mcd (a,b):
  if a<0: a=-a
  if b<0: b=-b
  if b==0: return a
  return mcd(b,a%b)
```

dove usiamo la relazione

$$\mathrm{mcd}(a, b) = \begin{cases} a & \text{se } b = 0 \\ \mathrm{mcd}(b, a\%b) & \text{se } b > 0 \end{cases}$$

Sia $d = \mathrm{mcd}(a, b)$. Si può dimostrare che esistono sempre $x, y \in \mathbb{Z}$ tali che $d = ax + by$, seguendo ad esempio il seguente ragionamento ricorsivo. Se abbiamo

$$\begin{aligned} a &= \alpha b + c \\ d &= bx' + cy' \end{aligned}$$

allora

$$d = bx' + (a - \alpha b)y' = ay' + b(x' - \alpha y')$$

per cui $d = ax + by$ con $x = y'$ ed $y = x' - \alpha y'$. L'algoritmo euclideo esteso restituisce la tupla (d, x, y):

```
def mcde (a,b):
  if a<0: a=-a
  if b<0: b=-b
  if b==0: return a,1,0
  d,x,y=mcde(b,a%b); alfa=a/b
  return d,y,x-alfa*y
```

I numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza $F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$. Quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Un programma iterativo in Python per calcolare l'n-esimo numero di Fibonacci:

```
def fib (n):
  if n<=1: return 1
  a=b=1
  for k in xrange(n-1): a,b=a+b,a
  return a
```

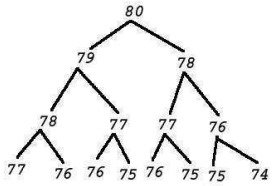
Possiamo visualizzare i numeri di Fibonacci da F_0 a F_{20} e da F_{50} a F_{60} con le seguenti istruzioni:

```
for n in xrange(10): print fib(n),
# 1 1 2 3 5 8 13 21 34 55

print
for n in xrange(50,61): print fib(n)
# Output:

20365011074
32951280099
53316291173
86267571272
139583862445
225851433717
365435296162
591286729879
956722026041
1548008755920
2504730781961
```

La definizione stessa dei numeri di Fibonacci è di natura ricorsiva, e quindi sembra naturale usare invece una funzione ricorsiva:

```
def fibr (n):
  if n<=1: return 1
  return fibr(n-1)+fibr(n-2)
```

Se però adesso per la visualizzazione sostituiamo `fib` con `fibr`, ci accorgiamo che il programma si blocca nella seconda serie, cioè che anche un computer a 3 GHz non sembra in grado di calcolare F_{50}. Infatti qui incontriamo il fenomeno di una ricorsione con *sovrapposizione dei rami*, cioè una ricorsione in cui le operazioni chiamate ricorsivamente vengono eseguite molte volte.



Questo fenomeno è frequente nella ricorsione doppia e diventa chiaro se osserviamo l'illustrazione che mostra lo schema secondo il quale avviene il calcolo di F_{80}. Si vede che F_{78} viene calcolato due volte, F_{77} tre volte, F_{76} cinque volte, ecc. Si ha l'impressione che riappaia la successione di Fibonacci ed è proprio così, quindi un numero esorbitante di ripetizioni impedisce di completare la ricorsione. È infatti noto che

$$\left| F_n - \tfrac{1}{\sqrt{5}}\left(\tfrac{1+\sqrt{5}}{2}\right)^{n+1} \right| < \frac{1}{2}$$

per ogni $n \in \mathbb{N}$ e da ciò segue che questo algoritmo è di complessità esponenziale.

Il sistema di primo ordine

In analisi si impara che un'equazione differenziale di secondo ordine può essere ricondotta a un sistema di due equazioni di primo ordine. In modo simile possiamo trasformare la ricorrenza di Fibonacci in una ricorrenza di primo ordine in due dimensioni. Ponendo $x_n := F_n, y_n := F_{n-1}$ otteniamo il sistema

$$\begin{aligned} x_{n+1} &= x_n + y_n \\ y_{n+1} &= x_n \end{aligned}$$

per $n \geq 0$ con le condizioni iniziali $x_0 = 1, y_0 = 0$ (si vede subito che ponendo $F_{-1} = 0$ la relazione $F_{n+1} = F_n + F_{n-1}$ vale per $n \geq 1$). Per applicare questo algoritmo dobbiamo però in ogni passo generare due valori, avremmo quindi bisogno di una funzione che restituisce due valori numerici. Ciò in Python, dove una funzione può restituire come risultato una lista, è molto facile:

```
def fibs (n):
  if n==0: return [1,0]
  (a,b)=fibs(n-1); return [a+b,a]
```

Per la visualizzazione usiamo le istruzioni

```
for n in xrange(50,61):
  print fibs(n)[0]
```

ottenendo in modo fulmineo il risultato.

Rappresentazione b-adica

Sia $b \in \mathbb{N}+2$. Allora ogni numero naturale n possiede una rappresentazione b-adica, cioè una rappresentazione della forma

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0$$

con coefficienti $a_i \in \{0, 1, \ldots, b-1\}$. Per $b = 2$ otteniamo la rappresentazione binaria, per $b = 16$ la rappresentazione esadecimale, entrambe viste a pagina 9. Come allora per $n = 0$ usiamo $k = 0$ ed $a_0 = 0$; per $n > 0$ chiediamo che $a_k \neq 0$. Con queste condizioni k e gli a_i sono univocamente determinati. Per calcolare questa rappresentazione (in forma di una lista di numeri $[a_k, \ldots, a_0]$) osserviamo che per $n < b$ la rappresentazione b-adica coincide con $[n]$, mentre per $n \geq b$ la otteniamo eseguendo la divisione intera $q = n/b$ e aggiungendo il resto $n\%b$ alla rappresentazione b-adica di q. Ad esempio si ottiene la rappresentazione 10-adica di $n = 7234$ aggiungendo il resto 4 di n modulo 10 alla rappresentazione 10-adica $[7, 2, 3]$ di 723. Possiamo facilmente tradurre questo algoritmo in una funzione in Python, il cui secondo argomento è la base b:

```
def rapp (n,base):
  if n<base: return [n]
  q,r=n/base,n%base
  return rapp(q,base)+[r]
```

Naturalmente questa funzione per $b = 2$ potrebbe sostituire la `rapp2pv` definita a pagina 9. Definiamo inoltre una funzione che per $b \leq 36$ trasforma la lista numerica ottenuta in una stringa, imitando l'idea utilizzata per i numeri esadecimali, sostituendo cioè le cifre da 10 a 35 con le lettere `a`,...,`z`:

```
def rappcomestringa (n,base):
  v=rapp(n,base)
  def codifica (x):
    if x<10: return str(x)
    return chr(x-10+ord('a'))
  return ''.join(map(codifica,v))
```

Esempi:

```
for x in (8,12,24,60,80,255,256):
  print rappcomestringa(x,16),
# 8 c 18 3c 50 ff 100

print
print rappcomestringa(974002,36)
# kvjm
```

È un comodo metodo per ricordarsi il numero telefonico del Dipartimento di Matematica o un qualsiasi altro numero telefonico che non inizia con 0.

Conversione di numeri

La funzione `int` può essere usata in vari modi. Se x è un numero, `int(x)` è l'intero che si ottiene da x per arrotondamento verso lo zero (quindi -13.2 diventa -13). Se a è una stringa, bisogna indicare un numero intero b compreso tra 2 e 36 e allora `int(a,base=b)` è l'intero rappresentato in base b dalle cifre che compongono la stringa a, interpretate nel modo usato anche da noi per `rappcomestringa`. Se la stringa inizia con `-`, viene calcolato il corrispondente intero negativo. In questa seconda forma `int(a,base=b)` è probabilmente equivalente a `string.atoi(a,base=b)`. Esempi:

```
for x in (3.5,4,6.9,-3.7): print int(x),
# 3 4 6 -3

print
print int('-88345',base=10)
# -88345

print int('kvjm',base=36)
# 974002

print string.atoi('kvjm',base=36)
# 974002
```

`float` converte un numero o una stringa adatta a un numero a virgola mobile:

```
for x in (3,8.88,'6.25','-40'):
  print float(x),
# 3.0 8.88 6.25 -40.0

x=100; y=17
print x/y, float(x)/y
# 5 5.88235294118
```

`complex(real=a,imag=b)` restituisce il numero complesso $a + bi$. I parametri `real` e `imag` sono preimpostati a zero. Invece di numeri `a` e `b` si può anche utilizzare come unico argomento una stringa che rappresenta il numero complesso nel formato previsto da Python:

```
print complex(7)
# (7+0j)

print complex(imag=8)
# 8j

print complex(imag=8,real=1)
# (1+8j)

print complex('7+4j')
# (7+4j)
```

Le funzioni `bool` e `str` sono state introdotte alle pagine 5 e 8.

Nella funzione `cesare` a pagina 8 e in `rappcomestringa` abbiamo utilizzato gli operatori di conversione `ord` (codice ASCII di un carattere) e `chr` (carattere che corrisponde a un codice ASCII).

`hex` e `oct` forniscono le rappresentazioni esadecimali e ottali di un numero intero nel formato talvolta utilizzato da Python:

```
for x in (15,92,187): print hex(x),
# 0xf 0x5c 0xbb
print
for x in (9,13,32): print oct(x),
# 011 015 040
```

Funzioni matematiche di base

Per calcolare il massimo di una sequenza `a` si può usare `max(a)`. Alternativamente questa funzione può essere usata anche con argomenti multipli, separati da virgole, di cui calcola il massimo. Nello stesso modo si usa `min` per il minimo.

```
a=[3,5,0,2,12,7,33,6,2]
print max(a)
# 33

print max(3,5,0,2,12,7,33,6,2)
# 33

print max('alfa','beta',17)
# beta

print min(3,8,[4,7],'otto')
# 3

print max([[3,6,2],[1,5,10]])
# [3,6,2]

print min([2,0,4],[1,7,15,18])
# [1,7,15,18]
```

`abs(x)` restituisce il valore assoluto del numero reale o complesso `x`:

```
for x in (7,-3,5+1j): print abs(x),
# 7 3 5.09901951359
```

Con `round(x)` si ottiene un arrotondamento del numero `x` al più vicino intero. Per $x \in \frac{1}{2}\mathbb{Z}$ tra $x - \frac{1}{2}$ e $x + \frac{1}{2}$ viene scelto l'intero più lontano dallo zero:

```
for x in (1.4, 1.5, 1.6):
  print round(x),
# 1.0 2.0 2.0

print
for x in (-1.4, -1.5, -1.6):
  print round(x),
# -1.0 -2.0 -2.0
```

`round` può essere anche usato con un secondo argomento che indica il numero di cifre decimali a cui si vuole arrotondare:

```
print round(1.68454,2)
# 1.68

print round(1.685,2)
# 1.69

print round(-1.68454,2)
# -1.68

print round(-1.685,2)
# -1.69
```

La potenza x^a può essere calcolata come `x**a` oppure con `pow(x,a)`. Nell'aritmetica intera si calcolano talvolta potenze in $\mathbb{Z}/m$: `pow(x,n,m)` è equivalente a `pow(x,n)%m`, ma più veloce.

```
for n in xrange(2,5):
  print pow(5,n,17),
# 8 6 13
```

Con `divmod(a,m)` si ottiene la tupla `(a/m,a%m)`, anche in questo caso per numeri negativi calcolata come in C, cioè diversamente da quanto si farebbe in matematica (cfr. pagina 6):

```
print divmod(30,11)
# (2, 8)
for x in (30/11,30%11): print x,
# 2 8

print
print divmod(30,-11)
# (-3, -3)
for x in (30/(-11),30%(-11)): print x,
# -3 -3
```

Potremmo usare questa funzione per abbreviare leggermente la funzione `rapp`:

```
def rapp (n,base):
  if n<base: return [n]
  q,r=divmod(n,base)
  return rapp(q,base)+[r]
```

Il modulo math

Questo modulo contiene le seguenti funzioni e costanti, corrispondenti per la maggior parte ad analoghe funzioni del C.

Funzioni trigonometriche: `math.cos`, `math.sin` e `math.tan`.

Funzioni trigonometriche inverse: `math.acos`, `math.asin`, `math.atan` e `math.atan2`.

`math.atan2(y,x)` calcola l'angolo nella rappresentazione polare di un punto $(x, y) \neq (0, 0)$ nel piano. Attenti all'ordine degli argomenti!

```
print math.degrees(math.atan2(1,0))
# 90

print math.degrees(math.atan2(1,-1))
# 135.0

print math.degrees(math.atan2(-1,-1))
# -135.0
```

Funzioni per la conversione di angoli a gradi: `math.degrees(alfa)` esprime l'angolo `alfa` in gradi, `math.radians(gradi)` calcola il valore in radianti di un angolo espresso in gradi.

```
for x in (90,135,180,270,360):
  print math.radians(x)
# Output:

1.57079632679
2.35619449019
3.14159265359
4.71238898038
6.28318530718

for x in (1.6,2.4,3.1,4.7,6.3):
  print math.degrees(x)
# Output:
91.6732472209
137.509870831
177.616916491
269.290163711
360.963410932
```

Funzioni iperboliche: `math.cosh`, `math.sinh` e `math.tanh`.

Ricordiamo le definizioni di queste importanti funzioni (Algoritmi, 28):

$$\cosh x := \frac{e^x + e^{-x}}{2}$$

$$\sinh x := \frac{e^x - e^{-x}}{2}$$

$$\tanh x := \frac{\sinh x}{\cosh x} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Esponenziale e logaritmi: `math.exp`, `math.pow`, `math.log`, `math.log10`, `math.ldexp`, `math.sqrt` e `math.hypot`.

Il logaritmo naturale di x lo si ottiene con `math.log(x)`, il logaritmo in base b con `math.log(x,b)`, il logaritmo in base 10 anche con `math.log10(x)`. `ldexp(a,n)` è uguale ad $a2^n$ e difficilmente ne avremo bisogno. Abbiamo già visto che `math.sqrt` calcola la radice di un numero reale > 0. Con `math.hypot(x,y)` otteniamo $\sqrt{x^2 + y^2}$. Invece di `math.pow` possiamo usare `pow`.

```
for x in (0,1,2,math.log(7)):
  print math.exp(x),
# 1.0 2.71828182846 7.38905609893 7.0

print
print math.ldexp(1,6)
# 64.0
print math.ldexp(2.5,6)
# 160.0

for b in (2,math.e,7,10):
  print round(math.log(10,b),4),
# 3.3219 2.3026 1.1833 1.0

print math.log(1024,2)
# 10.0
```

Alcune funzioni aritmetiche: `math.floor`, `math.ceil`, `math.fmod`.

`math.floor(x)` restituisce la parte intera del numero reale x, cioè il più vicino intero alla sinistra di x, `math.ceil` il più vicino intero alla destra di x. Entrambe le funzioni restituiscono valori reali e funzionano correttamente anche per argomenti negativi:

```
for x in (-1.3,-4,0.5,2,2.7):
  print math.floor(x), math.ceil(x)
# Output:

-2.0 -1.0
-4.0 -4.0
 0.0  1.0
 2.0  2.0
 2.0  3.0
```

Le funzioni `math.modf`, `math.fmod` e `math.frexp` non funzionano correttamente.

`math.fabs(x)` è il valore assoluto di x, ma possiamo usare la funzione `abs`.

Le costanti `math.e` e `math.pi` rappresentano i numeri e e π:

```
print math.e
# 2.71828182846

print math.pi
# 3.14159265359
```

In entrambi i casi l'ultima cifra è arrotondata in alto.

Il modulo cmath

Questo modulo fornisce (con il prefisso `cmath.`) le funzioni `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`, `exp`, `log`, `log10` e `sqrt` per argomenti complessi:

```
for k in xrange(1,8):
  print cmath.sin(k*math.pi*1j)
# Output:

11.5487393573j
267.744894041j
6195.82386361j
143375.656567j
3317811.99967j
76776467.6977j
1776660640.42j
```

Si vede chiaramente che sulla retta $\mathbb{R}i$ il seno cresce in modo esponenziale.

apply

f sia una funzione di n argomenti, dove n può essere anche variabile, e v una sequenza di lunghezza n. Allora `apply(f,v)` è uguale ad $f(v_1, \ldots, v_n)$. Esempi:

```
def somma (*a):
  s=0
  for x in a: s+=x
  return s

v=[1,2,4,9,2,8]
s=apply(somma,v)
print s # 26
```

---

reduce

f sia una funzione di due argomenti. Come in algebra scriviamo $a \cdot b := f(a, b)$. Per una sequenza $v = (a_1, \ldots, a_n)$ di lunghezza $n \geq 1$ l'espressione $\mathrm{reduce}(f, v)$ è definita come il prodotto (in genere non associativo) da sinistra verso destra degli elementi di v:

$$\begin{aligned}
\mathrm{reduce}(f, [a_1]) &= a_1 \\
\mathrm{reduce}(f, [a_1, a_2]) &= a_1 \cdot a_2 \\
\mathrm{reduce}(f, [a_1, a_2, a_3]) &= (a_1 \cdot a_2) \cdot a_3 \\
&\ldots \\
\mathrm{reduce}(f, [a_1, \ldots, a_{n+1}]) &= (a_1 \cdots a_n) \cdot a_{n+1}
\end{aligned}$$

`reduce` può essere anche usata con un argomento iniziale a_0; in questo caso il valore è definito semplicemente mediante

$$\mathrm{reduce}(f, [a_1, \ldots, a_n], a_0) := \mathrm{reduce}(f, [a_0, a_1, \ldots, a_n])$$

Nella genetica algebrica (che esprime le leggi di Mendel) si usa la composizione non associativa $a \cdot b := \dfrac{a+b}{2}$:

```
def f(x,y): return (x+y)/2.0

print reduce(f,[3]),
print reduce(f,[3,4]),
print reduce(f,[3,4,5]),
print reduce(f,[3,4,5,8])
# 3 3.5 4.25 6.125
```

Se, fissato x, nello schema di Horner definiamo $b \cdot a := bx + a$, possiamo riprogrammare l'algoritmo mediante `reduce`:

```
def hornerr (a,x):
  def f(u,v): return x*u+v
  return reduce(f,a,0)

a=[7,2,3,5,4]
print hornerr(a,10)
# 72354
```

Se per numeri reali $a, b > 0$ definiamo $a \cdot b := \dfrac{1}{a} + b$, otteniamo le frazioni continue, con gli argomenti invertiti nell'ordine:

```
def frazcont (v):
  def f(x,y): return y+1.0/x
  w=v[:]; w.reverse()
  return reduce(f,w)

print frazcont([2,3,1,5])
# 2.26086956522 = 52/23.0
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 5

Le torri di Hanoi

Abbiamo tre liste `a`, `b` e `c`. All'inizio `b` e `c` sono vuote, mentre `a` contiene i numeri $0, 1, \ldots, N-1$ in ordine strettamente ascendente. Vogliamo trasferire tutti i numeri da `a` in `b`, seguendo queste regole:

(1) Si possono utilizzare tutte e tre le liste nelle operazioni.
(2) In ogni passo si può trasferire solo l'elemento più in alto (cioè l'elemento più piccolo) di una lista a un altra, anteponendola agli elementi di questa seconda lista.
(3) In ogni passo, gli elementi di ciascuna delle tre liste devono rimanere in ordine naturale.

Si tratta di un gioco inventato dal matematico francese Edouard Lucas (1842-1891), noto anche per i suoi studi sui numeri di Fibonacci e per un test di primalità in uso ancora oggi. Nell'interpretazione originale `a`, `b` e `c` sono aste con una base. All'inizio `b` e `c` sono vuote, mentre su `a` sono infilati dei dischi perforati in ordine crescente dall'alto verso il basso, in modo che il disco più piccolo si trovi in cima. Bisogna trasferire tutti i disci sul paletto `b`, usando tutte e tre le aste, ma trasferendo un disco alla volta e in modo che un disco più grande non si trovi mai su uno più piccolo.

L'algoritmo più semplice è ricorsivo:

(1) Poniamo $n = N$.
(2) Trasferiamo i primi $n-1$ numeri da a all'inizio di c.
(3) Poniamo il primo numero di a all'inizio di b.
(4) Trasferiamo i primi $n-1$ numeri di c all'inizio di b.

In Python definiamo la seguente funzione, che contiene anche un comando di stampa affinché il contenuto delle tre liste venga visualizzato dopo ogni passaggio:

```
def hanoi (a,b,c,n=None):
  print a,b,c
  if n==None: n=len(a)
  if n==1: b.insert(0,spezza(a))
  else:
    hanoi(a,c,b,n-1); hanoi(a,b,c,1)
    hanoi(c,b,a,n-1)
```

Abbiamo usato la funzione `spezza` definita in precedenza.

```
a=range(0,4); b=[]; c=[]
hanoi(a,b,c)
print a,b,c
# Output:

[0, 1, 2, 3] [] []
[0, 1, 2, 3] [] []
[0, 1, 2, 3] [] []
[0, 1, 2, 3] [] []
[1, 2, 3] [] [0]
[0] [1] [2, 3]
[2, 3] [] [0, 1]
[0, 1] [2] [3]
[0, 1] [3] [2]
[1] [2] [0, 3]
[0, 3] [1, 2] []
[3] [] [0, 1, 2]
[0, 1, 2] [3] []
[0, 1, 2] [] [3]
[0, 1, 2] [3] []
[1, 2] [] [0, 3]
[0, 3] [1] [2]
[2] [3] [0, 1]
[0, 1] [2, 3] []
[0, 1] [] [2, 3]
[1] [2, 3] [0]
[0] [1, 2, 3] []
[] [0, 1, 2, 3] []
```

Il gioco delle torri di Hanoi è legato ad alcuni aspetti della teoria dei numeri, discussi nelle pagine Web indicate.

mathworld.wolfram.com/TowerofHanoi.html. Questa pagina contiene animazioni e molti dettagli interessanti anche per il matematico.

it.wikipedia.org/wiki/Torre_di_Hanoi. Wikipedia in italiano.

www.kernelthread.com/hanoi/. Hanoimania - programmi per le torri di Hanoi in 100 linguaggi.

Il codice genetico

Dizionari possono essere molto utili in alcuni compiti della bioinformatica. Definiamo un dizionario che descrive il codice genetico:

```
cogen1 = {'Gly' : 'ggg gga ggc ggt',
  'Ala' : 'gcg gca gcc gct',
  'Val' : 'gtg gta gtc gtt',
  'Leu' : 'ctg cta ctc ctt ttg tta',
  'Ile' : 'ata atc att',
  'Ser' : 'tcg tca tcc tct agc agt',
  'Thr' : 'acg aca acc act',
  'Cys' : 'tgc tgt', 'Met' : 'atg',
  'Asp' : 'gac gat', 'Glu' : 'gag gaa',
  'Asn' : 'aac aat', 'Gln' : 'cag caa',
  'Phe' : 'ttc ttt', 'Tyr' : 'tac tat',
  'Lys' : 'aag aaa', 'His' : 'cac cat',
  'Trp' : 'tgg',
  'Arg' : 'cgg cga cgc cgt agg aga',
  'Pro' : 'ccg cca ccc cct',
  'STOP' : 'tga tag taa'}

cogen = {}
for amina in cogen1.keys():
  v=cogen1[amina].split()
  for x in v: cogen[x]=amina

dna='ttagattgcttggagtcatacttagataca'
n=len(dna)

for i in xrange(0,n,3):
  tripla=dna[i:i+3]
  print cogen[tripla],

# Output scritto su due righe:

Leu Asp Cys Leu Glu
Ser Tyr Leu Asp Thr
```

In questo esempio abbiamo prima creato una tabella `cogen1` in cui ad ogni aminoacido è assegnata una stringa che contiene le triple di nucleotidi che corrispondono a questo aminoacido. Spezzando queste stringhe mediante `split` riusciamo poi a creare una tabella `cogen` in cui per ogni tripla è indicato l'aminoacido definito dalla tripla. Infine abbiamo tradotto un pezzo di DNA nella sequenza di aminoacidi a cui esso corrisponde secondo il codice genetico.

In questo numero

Dizionari

Abbiamo incontrato esempi di dizionari nelle pagine 2, 6 e in questa. Come voci (o chiavi) si possono usare oggetti non mutabili (ad esempio numeri, stringhe oppure tuple i cui elementi sono anch'essi non mutabili). Un dizionario, una volta definito, può essere successivamente modificato:

```
stipendi = {('Rossi','Trento') : 4000,
('Rossi','Roma') : 8000,
('Gardini','Pisa') : 3400}

stipendi[('Neri','Roma')]=4500

voci=list(stipendi.keys())
voci.sort()
for x in voci:
  print '%-s %7s %d' \
    %(x[0],x[1],stipendi[x])

# Output:

Gardini Pisa    3400
Neri    Roma    4500
Rossi   Roma    8000
Rossi   Trento  4000
```

Si osservi il modo in cui le voci sono state ordinate alfabeticamente, prima rispetto alla prima colonna, poi, in caso di omonimia, rispetto alla seconda colonna.

Se `d` è un dizionario, `d.keys()` è una lista che contiene le chiavi di `d`. L'ordine in cui queste chiavi appaiono nella lista non è prevedibile per cui spesso, ad esempio nell'output, essa viene ordinata come abbiamo fatto in alcuni degli esempi precedenti.

`d.has_key(x)` indica (mediante un valore di verità) se `d` possiede una voce `x`. Con `del d[x]` la voce `x` viene cancellata dal dizionario. Questa istruzione provoca un errore, se la voce `x` non esiste.

`diz.items()` è la lista delle coppie di cui consiste il dizionario.

```
diz = {'a' : 1, 'b' : 2, 'c' : 3,
  'd' : 4, 'x' : 5, 'y' : 6, 'z' : 7}

print diz.items()
# Output che riscriviamo su
# piu' righe:

[('a', 1), ('c', 3), ('b', 2),
('d', 4), ('y', 6), ('x', 5),
('z', 7)]
```

dict

La funzione `dict` può essere usata, con alcune varianti di sintassi, per generare dizionari da sequenze già esistenti:

```
d = dict(); print d
# {} - dizionario vuoto

a=[('u',1), ('v',2)]
d=dict(a); print d
# {'u': 1, 'v': 2}
```

La funzione permette anche di creare un dizionario senza dover scrivere gli apici per le voci, se queste sono tutte stringhe:

```
cogen1=dict(Gly='ggg gga ggc ggt',
  Ala='gcg gca gcc gct',...)
```

Utilissimo è

```
voci=['Rossi','Verdi','Bianchi']
stipendi=[1800,1500,2100]

print dict(zip(voci,stipendi))
# {'Bianchi': 2100, 'Rossi': 1800,
# 'Verdi': 1500}
```

Traduzione di nomenclature

Un problema apparentemente banale, ma molto importante in alcuni campi della ricerca scientifica è la traduzione di nomenclature. Assumiamo che un gruppo di scienziati abbia acquistato un microchip di DNA che permette di studiare l'espressione di 1000 geni che sono identificati secondo una nomenclatura definita dalla ditta produttrice e supponiamo che il gruppo di lavoro abbia studiato (per semplicità) gli stessi geni, ma sotto altro nome. Allora è necessario creare un dizionario che converte i nomi dei geni da una nomenclatura all'altra. Un problema simile si pone spesso quando si effettua una ricerca di una sostanza nota sotto vari nomi in Internet.

In HTML i simboli speciali possono essere individuati con un nome tradizionale oppure con un codice numerico. Per una conversione automatica si potrebbe utilizzare un dizionario:

```
spec = dict(agrave=224, ntilde=241,
  zeta=950, infin=8734)
```

Da questa tabella possiamo poi derivare i simboli veramente usati, ad esempio `&agrave;` e `&#224;` ecc. Quando una tabella data da un dizionario è biiettiva (nel senso che ad ogni voce corrisponde esattamente un valore e viceversa), la tabella inversa (che intuitivamente corrisponde semplicemente allo scambio delle due colonne) la si può ottenere nel modo seguente:

```
inv = {}
for x,y in tab.items():
  inv[y]=x
```

clear

Con `clear(diz)` vengono cancellate tutte le voci in un dizionario `diz` che quindi dopo l'esecuzione del comando risulta uguale al dizionario vuoto `{}`.

Fusione di dizionari

Con `tab.update(tab1)` il dizionario `tab` viene fuso con il dizionario `tab1`. Ciò significa più precisamente che alle voci di `tab` vengono aggiunte, con i rispettivi valori, le voci di `tab1`; voci già presenti in `tab` vengono sovrascritte.

```
tab=dict(a=1, b=2)
tab1=dict(b=300, c=4, d=5)

tab.update(tab1)
print tab
# {'a': 1, 'c': 4, 'b': 300, 'd': 5}
```

Argomenti associativi

Se definiamo una funzione

```
def f (**tab):
  print tab
```

la possiamo utilizzare nel modo seguente:

```
f(u=4,v=6,n=13)
# {'n': 13, 'u': 4, 'v': 6}
```

Gli argomenti associativi individuati dal doppio asterisco possono essere preceduti da argomenti fissi o opzionali, evitando interferenze. Come si vede, nella tabella che viene generata i nomi delle variabili vengono trasformati in stringhe che corrispondono alle voci della tabella.

Menu interattivi

Possiamo usare dizionari i cui valori sono funzioni per creare piccoli menu interattivi sul terminale.

```
def menu (scelte, testo=''):
  while 1:
    risposta=raw_input(testo)
    if risposta=='fine': break
    if scelte.has_key(risposta):
      scelte[risposta]()

def curva (): print 'Disegno una curva.'

def media (): print 'Calcolo la media.'

def file (): print 'Carico un file.'

scelte=dict(c=curva, m=media, f=file)

menu(scelte,'scelta: ')
```

Per aiutare l'utente si potrebbe visualizzare sul terminale anche l'elenco delle scelte possibili.

Con `d.get(x)` si ottiene `d[x]`; questo metodo può essere usato con un secondo argomento con cui si può impostare il valore che viene restituito quando la chiave indicata nel primo non esiste. Possiamo effettuare le seguenti modifiche nel nostro programma:

```
def menu (scelte, testo=''):
  while 1:
    risposta=raw_input(testo)
    if risposta=='fine': break
    scelte.get(risposta,passa)()

def passa (): pass
```

Concatenazione di più liste

Usiamo il metodo `extend` definito per le liste per creare una funzione che concatena un numero arbitrario di liste:

```
def concatl (*v):
  a=[]
  for x in v: a.extend(x)
  return a

a=[0,1,2,3,4,5]
b=[6,7,8]; c=[9,10,11,12]
print concatl(a,b,c)
# [0,1,2,3,4,5,6,7,8,9,10,11,12]
```

del

Il comando `del` cancella un *nome* che quindi non può più essere usato per riferirsi a un oggetto. Se lo stesso oggetto è conosciuto all'interprete sotto più nomi, gli altri nomi rimangono validi:

```
a=7; b=a; del a; print b
# 7
print a
# Errore: il nome a non e' definito

c=7; d=c; del d; print c
# 7
print d
# Errore: il nome a non e' definito
```

Si possono anche cancellare più nomi con un solo comando `del`:

```
a=88; b=a; c=a; del a,c
print b
# 88
```

Il comando `del` può essere anche utilizzato per eliminare un elemento da una lista:

```
a=[0,1,2,3,4,5,6,7]
del a[1]; print a
# [0, 2, 3, 4, 5, 6, 7]

b=[0,1,2,3,4,5,6,7]
del b[2:5]; print b
# [0, 1, 5, 6, 7]
```

Ciò ci permette di definire una funzione analoga al `pop`, la quale toglie il primo elemento da una lista e lo restituisce come risultato (`pop` fa la stesso cosa con l'ultimo elemento di una lista):

```
def spezza (a):
  x=a[0]; del a[0]; return x

a=[0,1,2,3,4,5,6]
x=spezza(a)
print x, a
# 0 [1, 2, 3, 4, 5, 6]
```

`del` può essere anche usato per dizionari:

```
voti = {'Rossi' : 28, 'Verdi' : 27,
  'Bianchi' : 25}
del voti['Verdi']
print voti
# {'Bianchi': 25, 'Rossi': 28}
```

Non confondere `del` con il metodo `remove` delle liste:

```
a=[0,1,2,3,4,5]
a.remove(2); print a[2]
# 3
```

Minilisp

Il Lisp è un linguaggio molto vasto che si basa però su un'idea semplice, benché estremamente efficace: Un programma in Lisp è una lista i cui elementi sono dati atomari (cioè dati che non sono liste) oppure altre liste. Se il primo elemento di una lista è una funzione, viene calcolato il valore di questa funzione sugli altri elementi della stessa lista che sono considerati suoi elementi. Possiamo saggiare la potenza delle operazioni in Python fin qui discusse, definendo una funzione `esegui` che, in modo ancora primitivo, corrisponde a questa idea. Si noti come in questo esempio si rivela utile la funzione `apply`.

```
import types

def esegui (a):
  if type(a)!=types.ListType: return a
  f=a[0]
  if type(f)!=types.FunctionType:
    # f non e' una funzione.
    return map(esegui,a)
  return apply(f,map(esegui,a[1:]))

def somma (*a):
  s=0
  for x in a: s+=x
  return s

def prodotto (*a):
  p=1
  for x in a: p*=x
  return p

programma = [somma,5,[prodotto,2,3,7],8]
print esegui(programma)
# 55
```

G. Steele: Common Lisp - the language. Digital Press 1990.
W. Stark: Lisp, lore, and logic. Springer 1990.
P. Graham: ANSI Common Lisp. Prentice-Hall 1996.

Impostare il limite di ricorsione

Se definiamo il fattoriale con

```
def fatt (n):
  if n==0: return 1
  else: return n*fatt(n-1)
```

riusciamo a calcolare con `fatt(998)` il fattoriale $998!$, mentre il programma termina con un errore se proviamo `fatt(999)`. Abbiamo infatti superato il limite di ricorsione, più precisamente dello stack utilizzato per l'esecuzione delle funzioni, inizialmente impostato a 1000. Si può ridefinire questo limite con la funzione `sys.setrecursionlimit`:

```
sys.setrecursionlimit(2000)

print fatt(1998)
# Funziona.
```

Per sapere il limite di ricorsione attuale si può usare la funzione `sys.getrecursionlimit`:

```
print sys.getrecursionlimit()
# 1000 (se non reimpostato)
```

Linearizzare una lista annidata

Per linearizzare una lista annidata possiamo usare la seguente funzione:

```
def lineare (a):
  v=[]
  for x in a:
    if isinstance(x,(list,tuple)):
      v.extend(lineare(x))
    else: v.append(x)
  return v
```

Studieremo la funzione `isinstance` più in dettaglio nell'ambito della programmazione orientata agli oggetti; qui evidentemente viene utilizzata per verificare se un oggetto è una lista o una tupla. Esempio:

```
a=[(1,3),5,[6,[8,1,2],5],8]

print lineare(a)
# [1, 3, 5, 6, 8, 1, 2, 5, 8]
```

Sottodizionari

Per estrarre da un dizionario le informazioni che corrispondono a un sottoinsieme di voci possiamo usare la seguente funzione:

```
def sottodizionario (diz,chiavi):
  return dict([(x,diz.get(x))
    for x in chiavi])
```

Esempio:

```
diz=dict(a=7,b=3,c=4,d=5,e=11)
sd=sottodizionario(diz,['a','e'])
print sd
# {'a': 7, 'e': 11}
```

Il più piccolo divisore

Per trovare il più piccolo divisore $d > 1$ di un numero intero $n \neq \pm 1, 0$, possiamo usare il seguente algoritmo, in cui (come nel crivello di Eratostene) usiamo che se n non è primo, necessariamente si deve avere $d^2 \leq \sqrt(n)$ e quindi anche $d^2 \leq [\sqrt(n)]$.

```
def divmin (n):
  r=int(math.sqrt(n))
  for d in xrange(2,r+1):
    if n%d==0: return d
  else: return n

print divmin(323)
# 17
print divmin(31)
# 31

for x in xrange(321,342,2):
  print divmin(x),
# 3 17 5 3 7 331 3 5 337 3 11
# Si presta per un algoritmo
# di crivello.
```

Il più piccolo divisore di 0 è 2, perché 0 è divisibile per ogni numero intero. Il nostro algoritmo però non funziona in questo caso, perché `n%0` provoca un errore. Potremmo anche inserire all'inizio della funzione le righe

```
  if n==0: return 2
  if n==1 or n==-1: return None
```

Note varie

`pass` è l'istruzione che non effettua nessuna operazione. L'abbiamo usata a pagina 17.

Se `d` è un dizionario, `d.values()` è una lista che contiene i valori delle singole voci; questa funzione non è particolarmente utile, perché in genere non si sa a quale voce un determinato valore è assegnato.

```
diz = {'a' : 1, 'b' : 2, 'c' : 3,
  'd' : 4, 'x' : 5, 'y' : 6, 'z' : 7}

print diz.values()
# [1, 3, 2, 4, 6, 5, 7]
```

Per ottenere i numeri di Fibonacci da F_0 ad F_n come lista, possiamo usare la funzione

```
def fib (n):
  v=[0,1]
  for i in xrange(n-1):
    v.append(v[-2]+v[-1])
  return v
```

Si noti che non si tratta di una ricorsione doppia (infatti la funzione stessa non è ricorsiva) e l'esecuzione è velocissima.

Sotto Linux (ma non con Enthought sotto Windows) per ogni file α*.py* Python crea un file semicompilato (*bytecode*) α*.pyc*. Ciò rende più veloce il caricamento dei moduli (ma non l'esecuzione), ha però lo svantaggio di affollare la nostra cartella. Il modo più semplice per eliminare questo problema è di inserire, alla fine del programma stesso un'istruzione che cancella i file *.pyc* creati:

```
import os

os.system('rm *.pyc')
```

Abbiamo introdotto il comando `os.system` a pagina 3.

„The Python world has definitely increased its growth rate over the last three years. The primary network newsgroup for the language has seen traffic grow to the point where only some sort of filtering allows the average reader to keep up. The number of questions from new users continues to increase, and the range of application areas with Python solutions is growing daily. Just the same, the programming community at large still manages to resist Python's charms, for reasons often unstated but doubtless including inertia and ignorance ...

Python ... is becoming increasingly popular with the web community. There are many reasons for this, not least of which are the amazing range of library code that comes with the system or can be obtained from other sources; the high levels of productivity that can be maintained over large projects; the wide range of platforms ...

Python is an excellent first language, but it also appeals to experienced programmers. Its clean syntax, coupled with a lack of static typing, make it ideal for the pragmatists among us who are primarily interested in programming languages as tools for getting jobs done," (Holden, xvii, 3, 21)

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 6

Output formattato

Abbiamo già usato varie volte (alle pagine 2, 9, 16) la possibilità di definire stringhe formattate mediante il simbolo %, secondo una modalità molto simile a quella utilizzata nelle istruzioni `printf`, `sprintf` e `fprintf` del C. Consideriamo un altro esempio:

```
m=124; x=87345.99
a='%3d %-12.4f' %(m,x)
print a
124 87345.9900
```

La prima parte dell'espressione che corrisponde ad `a` è sempre una stringa. Questa può contenere delle *sigle di formato* che iniziano con % e indicano la posizione e il formato per la visualizzazione degli argomenti aggiuntivi. Nell'esempio `%3d` *tiene il posto* per il valore della variabile `m` che (da un'istruzione `print`) verrà visualizzata come intero su uno spazio di tre caratteri, mentre `%-12.4f` indica una variabile di tipo `double` che è visualizzata su uno spazio di 12 caratteri, arrotondata a 4 cifre dopo il punto decimale, e allineata a sinistra a causa del - (altrimenti l'allineamente avviene a destra). Quando i valori superano lo spazio indicato dalla sigla di formato, viene visualizzato lo stesso il numero completo, rendendo però imperfetta l'intabulazione che avevamo in mente. Esempio:

```
a='u=%6.2f\nv=%6.2f' %(u,v)
print a
# Output:

u=12345.67
v= 20.00
```

Nonostante avessimo utilizzato la stessa sigla di formato `%6.2f` per `u` e `v`, prevedendo lo spazio di 6 caratteri per ciascuna di esse, l'allineamento in `v` non è più corretto, perché la `u` impiega più dei 6 caratteri previsti.

I formati più usati sono:

%c	carattere
%d	intero
%f	`double`
%s	stringa (utilizzando `str`)
%x	rappr. esadecimale minusc.
%X	rappr. esadecimale maiusc.
%o	rappr. ottale

Per specificare un segno di % nella sigla di formato si usa %%.

All'interno della specificazione di formato si possono usare - per l'allineamento a sinistra e `0` per indicare che al posto di uno spazio venga usato `0` come carattere di riempimento negli allineamenti a destra. Quest'ultima opzione viene usata spesso per rappresentare numeri in formato esadecimale byte per byte. Vogliamo ad esempio scrivere la tripla di numeri $(58, 11, 6)$ in forma esadecimale e byte per byte (cioè riservando due caratteri per ciascuno dei tre numeri). Le rappresentazioni esadecimali sono `3a`, `b` e `6`, quindi con

```
a=58; b=11; c=6
print '%2x%2x%2x' %(a,b,c)
# 3a b 6
```

otteniamo `3a b 6` come output; per ottenere il formato desiderato `3a0b06` (richiesto ad esempio dalle specifiche dei colori in HTML) dobbiamo usare invece

```
print '%02x%02x%02x' %(a,b,c)
# 3a0b06
```

Come visto, nelle sigle di formato di `printf` può essere specificato il numero dei caratteri da usare; con `*` questo numero diventa anch'esso variabile e viene indicato negli argomenti aggiuntivi. Assumiamo che vogliamo stampare tre stringhe variabili su righe separate; per allinearle, calcoliamo il massimo `m` delle loro lunghezze, usando le funzioni `len` e `max` del Python, e incorporiamo questa informazione nella sigla di formato:

```
a='Ferrara'; b='Roma'; c='Rovigo'
m=max(len(a),len(b),len(c))
print '|%*s|\n|%*s|\n|%*s|' \
  %(m,a,m,b,m,c)
```

Si ottiene l'output desiderato:

```
|Ferrara|
|   Roma|
| Rovigo|
```

L'elemento che segue il simbolo % esterno è una tupla; invece di indicare i singoli elementi possiamo anche usare una tupla definita in precedenza:

```
u=(7,3,5)

print "cosicche' a=%d, b=%d, c=%d" %u
# cosicche' a=7, b=3, c=5
```

Tramite stringhe formattate si possono anche costruire primitive tabelle. Con

```
stati=dict(Afghanistan=[652000,21000],
  Giordania=[89000,6300],
  Mongolia=[1587000,2580],
  Turchia=[780000,65000])

formato='%-11s | %14s | %7s'
print formato %('stato',
  'superficie/kmq','ab/1000')
print '-'*38
voci=stati.keys(); voci.sort()
for x in voci:
  stato=stati[x]
  print formato %(x,stato[0],stato[1])
```

otteniamo

```
stato       | superficie/kmq | ab/1000
--------------------------------------
Afghanistan |         652000 |   21000
Giordania   |          89000 |    6300
Mongolia    |        1587000 |    2580
Turchia     |         780000 |   65000
```

In questo numero

Invertire una stringa

Per poter applicare funzioni definite per le liste a una parola, si può trasformare la parola in una lista con `list`. Vogliamo ad esempio invertire una parola:

```
a='terra'
b=list(a); b.reverse()
print a
# terra
print b
# ['a', 'r', 'r', 'e', 't']
```

`a` non è cambiata (e non può cambiare, perché stringhe sono immutabili), perché `list` crea una copia (non profonda) di `a`. Ma `b` adesso è una lista; come ricaviamo una parola? Per fare ciò si può usare il metodo `join` previsto per le liste, che abbiamo già incontrato alle pagine 8, 9 e 14:

```
b=''.join(b); print b
# arret
```

Possiamo così definire una nostra funzione `invertistringa`:

```
def invertistringa (a):
  b=list(a); b.reverse()
  return ''.join(b)

a='acquario'
b=invertistringa(a)
print a
# acquario
print b
# oirauqca
```

Insiemi di lettere

Il modulo `string` contiene alcuni insiemi di lettere in forma di stringhe:

`.lowercase`	lettere minuscole
`.uppercase`	lettere maiuscole
`.letters`	minuscole e maiuscole
`.digits`	0123456789
`.hexdigits`	0123456789abcdefABCDEF
`.punctuation`	!"#$%&'()*+,-./:;<=>?@ [\]^_`{|}~
`.whitespace`	ASCII 9-13 e 32

Queste costanti devono essere usate con il prefisso `string`.

Unione di stringhe

Stringhe possono essere unite, come tutte le sequenze, mediante l'operatore + oppure, come abbiamo già visto alle pagine 8, 9, 14 e 19, utilizzando il metodo `join` del separatore che vogliamo usare. `join` è molto più veloce di +. La sintassi è

```
sep.join(v)
```

dove `sep` è il separatore, `v` una sequenza di stringhe che vogliamo unire utilizzando il separatore indicato. Un altro esempio:

```
print '--'.join(['alfa','beta','rho'])
# alfa--beta--rho
```

`join` viene usato piuttosto frequentemente, in genere con i separatori `' '` e `'\n'`. Le parentesi quadre necessarie quando gli elementi del vettore `v` delle stringhe da unire vengono dati esplicitamente possono ridurre la leggibilità (soprattutto quando il `join` nella stessa espressione viene usato più volte); perciò definiamo alcune funzioni ausiliarie con nomi brevi per questi casi:

```
def U (*a): return ''.join(a)

def Unr (*a): return '\n'.join(a)

def Usp (*a): return ' '.join(a)

def Uv (a): return ''.join(a)

def Uvnr (a): return '\n'.join(a)

def Uvsp (a): return ' '.join(a)
```

Esempi:

```
a=U('Africa','Asia','Europa')
print a
# AfricaAsiaEuropa

a=Usp('Africa','Asia','Europa')
print a
# Africa Asia Europa

a=Unr('Africa','Asia','Europa')
print a # Output:

Africa
Asia
Europa
---------------------------------------
a=Unr(Usp('Africa','Asia','Europa'),
    Usp('America','Australia'),
    Usp('Artide','Antartide'))
print a
# Output

Africa Asia Europa
America Australia
Artide Antartide
---------------------------------------
p=['Africa','Asia','Europa']
q=['America','Australia']
r=['Artide','Antartide']
a=Unr(Uvsp(p),Uvsp(q),Uvsp(r))
print a
# Output = precedente.
---------------------------------------
print Uvsp(string.digits)
# 0 1 2 3 4 5 6 7 8 9

url='www.unife.it'; fe='Ferrara'
print U('<a href="',url,'">',fe,'</a>')
# <a href="www.unife.it">Ferrara</a>
```

upper e lower

Per trasformare tutte le lettere di una stringa in maiuscole risp. minuscole si usano i metodi `upper` e `lower` (visto a pagina 8):

```
a='Padre Pio'; b='USA'
print a.upper(), b.lower()
# PADRE PIO usa
```

Il metodo `swapcase` trasforma minuscole in maiuscole e viceversa:

```
print 'Padre Pio'.swapcase()
# pADRE pIO
```

Il metodo `capitalize` trasforma la lettera iniziale di una stringa in maiuscola, le altre in minuscole, anche quando sono precedute da uno spazio. Se si desidera invece che la lettera iniziale della stringa e le lettere precedute da uno spazio vengano trasformate in maiuscole, bisogna usare il metodo `title` oppure la funzione `string.capwords`; quest'ultima sostituisce anche ogni spazio multiplo con uno spazio semplice e si comporta diversamente da `title` quando incontra lettere precedute da un'interpunzione:

```
print 'alfa beta rho'.capitalize()
# Alfa beta rho

print string.capwords('alfa beta rho')
# Alfa Beta Rho
print 'alfa beta rho'.title()
# Alfa Beta Rho

print string.capwords('alfa    beta')
# Alfa Beta
print 'alfa    beta'.title()
# Alfa    Beta

print string.capwords('ab ,u , u 2a xy')
# # Ab ,u , U 2a Xy
print 'ab ,u , u 2a xy'.title()
# Ab ,U , U 2A Xy
```

Verifica del tipo di carattere

I seguenti metodi restituiscono `False`, se `a` è una stringa vuota. Supponiamo quindi che `a` sia una stringa con almeno un carattere.

`a.isalpha()`	Vero, se ogni carattere di `a` è una lettera, fa cioè parte di `string.letters`.
`a.isdigit()`	Vero, se ogni carattere di `a` è una delle cifre 0,...,9.
`a.isalnum()`	Vero, se ogni carattere di `a` è una lettera o una delle cifre 0,...,9.
`a.islower()`	Vero, se tutte le lettere tra i caratteri di `a` sono minuscole.
`a.isupper()`	Vero, se tutte le lettere tra i caratteri di `a` sono maiuscole.
`a.isspace()`	Vero, se tutti i caratteri di `a` appartengono a `string.whitespace`.

```
print '3iax--ff'.islower()
# True
print 'NORD SUD'.isupper()
# True
```

Ricerca in stringhe

Per la ricerca in stringhe sono disponibili i seguenti metodi. Contrariamente a quanto affermato a pagina 4, `index` e `count` possono essere usati anche per stringhe. `index` non è indicato nel seguente elenco perché, a differenza da `find`, provoca un errore quando la sottostringa cercata non fa parte della stringa. In seguito supponiamo che `a` ed `x` siano stringhe.

`a.find(x)`	Indice della prima posizione in cui `x` appare in `a`; restituisce -1 se `x` non fa parte di `a`.
`a.find(x,i)`	Come `find`, ma con ricerca in `a[i:]`. L'indice trovato si riferisce ad `a`.
`a.find(x,i,j)`	Come `find`, ma con ricerca in `a[i:j]`. L'indice trovato si riferisce ad `a`.
`a.rfind(x)`	Come `find`, ma la ricerca inizia a destra, con l'indice comunque sempre contato dall'inizio della stringa.
`a.rfind(x,i)`	Come per `find`.
`a.rfind(x,i,j)`	Come per `find`.
`a.count(x)`	Indica quante volte `x` appare in `a`.
`a.count(x,i)`	Indica quante volte `x` appare in `a[i:]`.
`a.count(x,i,j)`	Indica quante volte `x` appare in `a[i:j]`.
`a.startswith(x)`	Vero se `a` inizia con `x`.
`a.startswith(x,i)`	Vero, se `a[i:]` inizia con `x`.
`a.startswith(x,i,j)`	Vero, se `a[i:j]` inizia con `x`.
`a.endswith(x)`	Vero, se `a` termina con `x`.
`a.endswith(x,i)`	Vero, se `a[i:]` termina con `x`.
`a.endswith(x,i,j)`	Vero, se `a[i:j]` termina con `x`

Esempi:

```
a='01201012345014501020'

print a.find('0120')
# 0
print a.find('01',2)
# 3

print a.rfind('010')
# 15

print a.count('01')
# 5

print a.startswith('012')
# True
print a.startswith('120')
# False
print a.startswith('120',1)
# True
print a.startswith('120',1,2)
# False
```

Eliminazione di spazi

Uno dei compiti più frequenti dell'elaborazione di testi elementare è l'eliminazione (talvolta anche l'aggiunta) di spazi (o eventualmente di altri caratteri) iniziali o finali da una parola. Elenchiamo i metodi per le stringhe previste a questo scopo in Python. `a` ed `s` siano stringhe, `n` un numero naturale; `s` viene utilizzata come contenitore dei caratteri da eliminare. Tutti i metodi restituiscono una nuova stringa senza modificare `a` (infatti stringhe non sono mutabili).

`a.strip()`	Si ottiene da `a`, eliminando spazi bianchi (caratteri appartenenti a `string.whitespace`) iniziali e finali.
`a.strip(s)`	Si ottiene da `a`, eliminando i caratteri iniziali e finali che appartengono ad `s`.
`a.lstrip()`	Come `strip`, eliminando però solo caratteri iniziali.
`a.lstrip(s)`	Come `strip`, eliminando solo caratteri iniziali.
`a.rstrip()`	Come `strip`, eliminando solo caratteri finali.
`a.rstrip(s)`	Come `strip`, eliminando solo caratteri finali.
`a.ljust(n)`	Se la lunghezza di `a` è minore di `n`, vengono aggiunti `n-len(a)` spazi alla *fine* di `a`.
`a.rjust(n)`	Se la lunghezza di `a` è minore di `n`, vengono aggiunti `n-len(a)` spazi all'*inizio* di `a`.

```
a='  libro  '
print '[%s]' %(a.strip())
# [libro]

print '[%s]' %(a.lstrip())
# [libro  ]

print '[%s]' %(a.rstrip())
# [  libro]

b='aei terra iia'
print '[%s]' %(b.strip('eia'))
# [ terra ]

c='gente'
print '[%s]' %(c.rjust(7))
# [  gente]
```

split

Abbiamo incontrato questo metodo (usato spessissimo), con cui una stringa può essere decomposta in una lista di parole, a pagina 16. `a` ed `s` siano stringhe, `n` un numero naturale > 0.

`a.split()`	Lista di stringhe che si ottiene spezzando `a`, utilizzando come stringhe separatrici le stringhe consistenti di spazi bianchi.
`a.split(s)`	Lista di stringhe che si ottiene spezzando `a`, usando `s` come separatrice.
`a.split(s,n)`	Al massimo `n` separazioni.

Con il terzo argomento opzionale `n` si può prescrivere il numero massimo di separazioni, ottenendo quindi al massimo `n+1` parti. Questa opzione si usa ad esempio con `maxsplit=1`, se si vuole separare una parola solo nel primo spazio che essa contiene.

```
a='Roma,   Como, Pisa'

print a.split()
# ['Roma,', 'Como,', 'Pisa']

print a.split(',')
# ['Roma', '   Como', ' Pisa']

print map(lambda x: x.strip(),
  a.split(','))
# ['Roma', 'Como', 'Pisa']

b='0532 Comune di Ferrara'
print b.split(' ',1)
# ['0532', 'Comune di Ferrara']
```

Sostituzioni in stringhe

Se `a`, `u` e `v` sono stringhe, `a.replace(u,v)` è la stringa che si ottiene da `a` sostituendo tutte le apparizioni (non sovrapposte) di `u` con `v`.

```
a='andare, creare, stare'
b=a.replace('are','ava')
print b
# andava, creava, stava

a='ararat'
a=a.replace('ara','ava')
print a
# avarat
```

Per sostituire in `a` simultaneamente e nell'ordine tutti i caratteri di una stringa `p` con i corrispondenti caratteri di una stringa `q`, bisogna (se non si usano espressioni regolari, che verranno trattate più avanti) procedere in due passi. In un primo momento si crea con `string.maketrans` una tabella di traduzione, successivamente con il metodo `translate` si ottiene la stringa desiderata. Benché macchinosa, l'esecuzione di questa operazione è molto veloce.

```
a='CRASCASTRAMOVEBO'
cesare=string.maketrans('ABCEMORSTV',
  'DEFHPRUVWY')
print a.translate(cesare)
# FUDVFDVWUDPRYHER

b='alfa, beta, gamma'
tra=string.maketrans(', ','!-')
print b.translate(tra)
# alfa!-beta!-gamma

c='ax tay uvzz xy'
tra=string.maketrans('xyz','XYZ')
print c.translate(tra)
# aX taY uvZZ XY
```

Simulare printf

La seguente ricetta, proposta da Tobias Klausmann e Andrea Cavalcanti nel cookbook, permette di simulare la funzione `printf` del C, mancante in Python.

```
def printf (format,*valori):
  sys.stdout.write(format %valori)
```

Essa talvolta risulta più leggibile del normale uso di stringhe formattate esposto a pagina 19. Mentre `print` passa o una nuova riga per ogni argomento che non sia seguito da una virgola, quando usiamo questa funzione la nuova riga va indicata, come in C, con il carattere `'\n'`; per questo invece di `print` abbiamo usato `sys.stdout.write`.

```
u=7; v=7.23; comune='Pisa'
printf('u=%d, v=%d\n',u,v)
printf('comune di %s\n',comune)
# Output:

u=7, v=7
comune di Pisa
```

A. Martelli/A. Ravenscroft/D. Ascher (ed.): Python cookbook. O'Reilly 2005.

Un semplice automa

A ed X siano insiemi finiti e $\varphi : X \times A \longrightarrow X$ un'applicazione. Allora la tripla (X, A, φ) si chiama un'*automa finito*. Questa è una delle possibili definizioni, di cui esistono anche varianti più generali.

Definiamo adesso l'insieme A^* delle *parole* sull'alfabeto A ponendo

$$A^* := \bigcup_{n=0}^{\infty} A^n$$

e denotiamo con ε la parola vuota, l'unico elemento di A^0. A^* è un semigruppo (altamente non commutativo) con la concatenazione naturale delle parole in cui ε funge da elemento neutro, per cui A^* è anche un monoide. Per ogni $a \in A$ abbiamo un'applicazione $\varphi_a := \bigcirc_x \varphi(x, a) : X \longrightarrow X$

e ciò ci permette di definire per ogni parola non vuota $p = a_1 \dots a_n$ un'applicazione $\varphi_p : X \longrightarrow X$ ponendo

$$\varphi_p(x) := \varphi_{a_n} \circ \cdots \circ \varphi_{a_1}$$

mentre $\varphi_\varepsilon(x) := x$. In questo modo otteniamo un sistema dinamico

$$X \times A^* \longrightarrow X$$

Realizzazione in Python: Siano $A = \{a, b\}$, $X = \{0, 1, 2, 3, 4\}$, e φ data dalla tabella

	a	b
0	1	4
1	3	4
2	2	3
3	0	1
4	1	0

che corrisponde al dizionario annidato

```
phi=dict(a={0:1,1:3,2:2,3:0,4:1},
  b={0:4,1:4,2:3,3:1,4:0})
```

I calcoli sono effettuati nel modo seguente:

```
def valautoma (phi):
  def val (p,x):
    for a in p: x=phi[a][x]
    return x
  return val

for x in xrange(5):
  print valautoma(phi)('baa',x),
# 3 3 1 0 3
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 7

Lettura e scrittura di files

Per leggere un file utilizziamo la funzione

```
def leggifile (nome):
  f=file(nome,'r'); testo=f.read()
  f.close(); return testo
```

che ci restituisce il contenuto del file in un'unica stringa. Se da questa vogliamo ottenere una lista che contiene le singole righe, possiamo procedere come nel seguente esempio:

```
a=leggifile('diffusion.f')
righe=a.split('\n')
```

In questo caso *diffusion.f* è il nome di un file nella stessa cartella in cui si trova il programma; altrimenti sotto Linux potremmo ad esempio usare il nome `/home/rita/fortran/diffusion.f` oppure, sotto Windows, `c:/rita/fortran/diffusion.f`.

Per poter scrivere un testo su un file, definiamo la funzione

```
def scrivifile (nome,testo):
  f=file(nome,'w')
  f.write(testo); f.close()
```

Se il testo consiste di più parti, possiamo unire queste parti mediante `join` oppure effettuare più operazioni di scrittura tra `file` e `close` come nella funzione `scrivifilev` che permette di scrivere su un file gli elementi di un vettore `v` di testi, separati da una stringa separatrice inizialmente impostata a `'\n'`:

```
def scrivifilev (nome,testi,sep='\n'):
  f=file(nome,'w')
  for x in testi: f.write(x+sep)
  f.close()
```

Queste operazioni sono molto semplici e quindi in un piccolo programma da una pagina spesso non si usano nemmeno funzioni apposite, scrivendo invece direttamente le istruzioni necessarie:

```
f=file('prova','w')
f.write('Maria Tebaldi\n')
f.write('Roberto Magari\n')
f.write('Erica Rossi\n')
f.close()
```

Quando un file viene aperto con la modalità di scrittura usando la sigla `'w'` come secondo argomento di `file`, il contenuto del file viene cancellato. Se vogliamo invece aggiungere un testo alla fine del file, senza cancellare il contenuto esistente, dobbiamo usare la sigla `'\'{a}`:

```
def aggiungifile (nome,testo):
  f=file(nome,'a')
  f.write(testo); f.close()
```

Anche qui potremmo definire una funzione `aggiungifilev` per la scrittura di un vettore di testi.

Comandi per files e cartelle

`os.getcwd()`	Restituisce il nome completo della cartella di lavoro.
`os.chdir(cartella)`	Cambia la cartella di lavoro.
`os.listdir(cartella)`	Lista dei nomi (corti) dei files contenuti in una cartella.
`os.path.abspath(nome)`	Restituisce il nome completo del file (o della cartella) `nome`.
`os.path.split(stringa)`	Restituisce una tupla con due elementi, di cui il secondo è il nome corto, il primo la cartella corrispondente a un file il cui nome è la stringa data. Si tratta di una semplice separazione della stringa; non viene controllato, se essa è veramente associata a un file.
`os.path.basename(stringa)`	Seconda parte di `os.path.split(stringa)`.
`os.path.dirname(stringa)`	Prima parte di `os.path.split(stringa)`.
`os.path.exists(nome)`	Vero, se `nome` è il nome di un file o di una cartella esistente.
`os.path.isdir(nome)`	Vero, se `nome` è il nome di una cartella.
`os.path.isfile(nome)`	Vero, se `nome` è il nome di un file.
`os.path.getsize(nome)`	Grandezza in bytes del file `nome`; provoca un errore se il file non esiste.

Esempi per l'utilizzo di `os.path.split`:

```
print os.path.split('/alfa/beta/gamma')
# ('/alfa/beta', 'gamma')

print os.path.split('/alfa/beta/gamma/')
# ('/alfa/beta/gamma', '')

print os.path.split('/')
# ('/', '')
print os.path.split('//')
# ('//', '') - benche' '//' non sia correttamente formato.

print os.path.split('/stelle')
# ('/', 'stelle')

print os.path.split('stelle/sirio')
# ('stelle', 'sirio')
```

In questo numero

sys.argv

`sys.argv` è un vettore di stringhe che contiene il nome del programma e gli eventuali parametri con i quali il programma è stato chiamato dal terminale (sia sotto Linux che sotto Windows). Assumiamo che (sotto Windows) il file *prova.py* contenga le seguenti righe:

```
v=sys.argv
print v[0]

a=leggifile(v[1])
print a
```

Se adesso dal terminale diamo il comando `prova.py lettera`, verrà visualizzato prima il nome del programma, cioè *prova.py*, e poi il contenuto del file *lettera*. Se il programma contiene invece le istruzioni

```
v=sys.argv
x=int(v[1]); y=int(v[2])
print x+y
```

dopo `prova.py 7 9` dal terminale verrà visualizzato `16`.

Con il seguente programma possiamo effettuare dal terminale la codifica di un testo secondo cesare (pagina 8):

```
parola=sys.argv[1]

def cesare (a):
  # Come a pagina 8
  ...

print cesare(parola)
```

Se dal terminale battiamo

```
prova.py FORTISSIMISUNTBELGAE
```

otteniamo come risposta

```
IRUWLVVLPLVXQWEHOJDH
```

`sys.argv` viene spesso utilizzato in combinazione con `execfile` (pagina 25) oppure, e questa è una tecnica molto potente, per attivare l'elaborazione di un file (ad esempio un file di testo su cui si sta lavorando) mediante un altro programma, ad esempio Latex.

Moduli

Un modulo è un file che contiene istruzioni di Python che possono essere utilizzate da altri moduli o programmi. Il nome del file deve terminare in *.py* (almeno nel caso dei moduli da noi creati), ad esempio *matematica.py*. Il file può trovarsi nella stessa cartella che contiene il programma, oppure in una delle cartelle in cui l'interprete cerca i programmi. La lista con i nomi di queste cartelle è `sys.path`. Per aggiungere una nuova cartella si può usare il metodo `sys.path.append`. Quando il programmatore può accedere al computer in veste di amministratore di sistema (come *root* sotto Linux), esiste però una soluzione molto più comoda per rendere visibili i moduli all'interprete di Python. È infatti sufficiente inserire un file (o anche più files) con l'estensione `.pth` nella apposita cartella di Python, che sotto Linux è tipicamente

```
/usr/local/lib/python2.4/site-packages
```

e sotto Windows (nella nostra versione di Python)

```
c:/python23
```

In essa creiamo il file `cammini.pth` che contiene sotto Linux ad esempio la riga

```
/home/rita/python/moduli
```

e sotto Windows

```
c:/docs/rita/documenti/python/moduli
```

dove abbiamo usato *docs* come abbreviazione per *documents and settings*. Nello stesso modo si possono anche indicare più cartelle per i moduli. Ogni volta quando l'interprete del Python entra in azione, tiene conto di queste direttive per la collocazione dei moduli che vogliamo utilizzare. Più precisamente l'interprete effettua la ricerca dei moduli nel seguente ordine:

(1) Cartella in cui si trova il programma.
(2) Cartelle elencate nella variabile `PYTHONPATH`, se impostata.
(3) Cartelle della libreria standard di Python.
(4) Cartelle indicate nei files *.pth*, se presenti.

Il percorso di ricerca completo può essere controllato esaminando la lista `sys.path`.

Dopo aver importato il modulo `matematica` con l'istruzione

```
import matematica
```

tralasciando il suffisso *.py*, un oggetto `x` di `matematica` al di fuori del modulo stesso è identificato con `matematica.x`. Possiamo anche importare tutti i nomi (in verità solo i nomi pubblici) di `matematica` con

```
from matematica import *
```

oppure anche solo alcuni degli oggetti del modulo, come abbiamo fatto con `rpy`:

```
from rpy import r
```

Ai fini della trasparenza dei programmi i moduli utilizzati (tranne il programma principale) dovrebbero soltanto contenere definizioni di funzioni ed eventualmente impostazioni iniziali di variabili globali. Il modulo può contenere un qualsiasi insieme di istruzioni che vengono eseguite all'atto dell'importazione che però in questo modo sono difficilmente visibili al programmatore. Si consiglia quindi di usare i moduli solo come raccolte di elementi passivi che verranno attivati dal programma principale.

Anche per le variabili importate mediante un `from` da un modulo bisogna distinguere tra oggetti mutabili e immutabili. Assumiamo che il modulo `aus` contenga le istruzioni

```
x=1; a=[1,2,3]
```

e il programma principale le seguenti righe:

```
from aus import x,a
x=10; a[1]=77

import aus
print aus.x, aus.a
# 1 [1, 77, 3]
```

Vediamo che la variabile `aus.x` non è stata modificata, mentre l'istruzione `a[1]=77` ha avuto effetto sulla lista `aus.a`. Semplificando leggermente, la ragione di questo comportamente è che l'istruzione

```
from aus import x,a
```

è essenzialmente equivalente alle istruzioni

```
x=aus.x
a=aus.a
```

e quindi ciò che abbiamo osservato è in accordo con quanto detto a pagina 4.

Pacchetti

Nelle istruzioni `import` non è possibile utilizzare direttamente cammini composti (ad esempio `grafica/cerchi`). Esiste comunque un meccanismo che permette di raccogliere moduli in sottocartelle, anche annidate. Assumiamo che (per semplicità nella stessa cartelli in cui si trova il programma principale) abbiamo creato una cartella *grafica* e in essa un file *cerchi.py*. Allora possiamo utilizzare il contenuto di questo file mediante l'istruzione

```
import grafica.cerchi
```

sotto la condizione però che la cartella *grafica* contenga un file *__init__.py* che può essere anche vuoto; se contiene delle istruzioni, queste vengono eseguite durante l'importazione, ma non è una cattiva idea lasciarlo vuoto. La cartella grafica può anche contenere una sottocartella *astro*; se questa contiene a sua volta un file *__init__.py* e un file *stelle.py*, possiamo importare quest'ultimo con

```
import grafica.astro.stelle
```

Ogni volta che usiamo un oggetto di questo modulo dovremmo riscrivere il lungo nome di questo modulo; per evitare ciò è possibile un'istruzione

```
import grafica.astro.stelle as stelle
```

che ci permette di scrivere semplicemente `stelle`. L'istruzione `import ... as ...` può essere anche usata per nomi complicati non contenuti in un pacchetto.

L'attributo __name__

A differenza dal C, in Python non esiste una chiara distinzione tra programma principale e files ausiliari. Infatti ogni file che contiene istruzioni di Python valide può essere utilizzato come programma principale e, se il suo nome termina con *.py*, anche essere importato da un altro file. Come già osservato, è preferibile separare in modo trasparente i moduli dal programma principale; per capire meglio il meccanismo facciamo però vedere che, tra due files *A.py* e *B.py* ciascuno possa fungere da programma principale utilizzando il secondo come modulo.

Assumiamo che *A.py* contenga le righe

```
if __name__=='__main__':
  import B
  print B.g(3)

def f(x): return x*3-1
```

e *B.py* le righe

```
if __name__=='__main__':
  import A
  print A.f(5)

def g(x): return 2*x+6
```

Se adesso dal terminale eseguiamo il comando `python A.py`, verrà visualizzato `12` (il valore di `B.g(3)`), se battiamo `python B.py`, l'output sarà `14`, il valore di `A.f(5)`.

Osserviamo in primo luogo che tralasciando l'`if`, l'esecuzione si fermerebbe con un errore perché i comandi di importazione reciproca creano una specie di corto circuito.

Possiamo invece, come nell'esempio, utilizzare il fatto che ad ogni modulo è associato un attributo `__name__` che, quando il modulo viene usato come programma principale, diventa `__main__`, mentre altrimenti coincide con il nome del modulo, cioè il nome del file senza l'estensione *.py*.

Alcune costanti in sys

Il modulo `sys` contiene alcune costanti che possono essere talvolta utili, tra cui:

`sys.executable`	Cammino completo in cui si trova l'interprete di Python.
`sys.path`	Lista con i nomi delle cartelle in cui l'interprete cerca i moduli.
`sys.platform`	Ad esempio `linux2` oppure `win32`.

Sotto Windows avremo ad esempio

```
print sys.executable
# c:\Python23\pythonw.exe
```

Variabili globali

Variabili *alla sinistra di assegnazioni* all'interno di una funzione e non riferite esplicitamente a un modulo sono *locali*, se non vengono definite globali con `global`. Non è invece necessario dichiarare `global` variabili esterne che vengono solo lette, senza che gli venga assegnato un nuovo valore.

```
x=7

def f(): x=2
f(); print x
# 7

def g(): global x; x=2

g(); print x
# 2
```

Nell'esempio

```
u=[8]

def f(): u[0]=5

f(); print u
# [5]
```

non è necessario dichiarare `u` come variabile globale, perché viene usata solo in lettura. Infatti non viene cambiata la `u`, ma solo un valore in un indirizzo a cui `u` punta.

Quindi si ha ad esempio anche

```
u=[8]

def aumenta (u): u[0]=u[0]+1

aumenta(u); print u
# [9]
```

Invece di dichiarare una variabile come globale, la si può anche riferire esplicitamente a un modulo. Assumiamo prima che la variabile `u` appartenga a un modulo esterno, ad esempio `mat`, in cui abbia inizialmente il valore 7. Allora possiamo procedere come nel seguente esempio:

```
import mat

x=7

def f ():
  mat.x+=1

f(); print x
# 8
```

Se la variabile appartiene invece allo stesso modulo come la funzione che la dovrebbe modificare, possiamo usare `sys.modules` per identificare il modulo:

```
x=7

def f ():
  sys.modules[__name__].x+=1

f(); print x
# 8
```

A questo scopo possiamo anche importare il modulo stesso in cui ci troviamo e usare la tecnica del penultimo esempio.

Si dovrebbe cercare di utilizzare variabili globali solo quando ciò si rivela veramente necessario e, in tal caso, di non definirle nel file che contiene il programma principale, ma in moduli appositi.

---

globals() e locals()

Con `globals()` si ottiene una tabella (in forma di dizionario) di tutti i nomi globali. Il programma può modificare questa tabella:

```
globals()['x']=33
print x
# 33
```

Come si vede, per ogni oggetto bisogna usare il suo nome come stringa; le istruzioni

```
x=33
```

e

```
globals()['x']=33
```

sono in pratica equivalenti. `locals()` è la lista dei nomi locali. La differenza si vede ad esempio all'interno di una funzione. Assumiamo che il nostro file contenga le seguenti istruzioni:

```
x=7

def f (u):
  print globals()
  print '------------'
  print locals()
  x=3; a=4

f(0)
```

Allora l'output (che abbiamo disposto su più righe sostituendo informazioni non essenziali con ...) sarà

```
{'f': <function ...>,
'__builtins__': <module ...>,
'__file__': './alfa',
'x': 7, '__name__': '__main__',
'__doc__': None}
------------
{'u': 0}
```

Si osservi in particolare che le variabili locali `x` ed `a` non sono state stampate, perché al tempo della chiamata di `locals()` ancora non erano visibili. Esse appaiono invece nell'elenco delle variabili locali della funzione se il file contiene le righe

```
x=7

def f (u):
  x=3; a=4
  print locals()
  x=100

f(0)
# {'a': 4, 'x': 3, 'u': 0}
```

`locals()` è stato stampato prima che venisse effettuato l'assegnamento `x=100`. Si noti che gli argomenti della funzione (in questo caso `u`) sono considerati variabili locali della funzione.

Variabili autonominative

Chiamiamo *autonominativa* una variabile il cui valore è una stringa che coincide con il nome della variabile. Potremmo definire una tale variabile ad esempio con

```
Maria='Maria'
```

ma ciò ci obbliga a scrivere due volte ognuno di questi nomi. Si può ottenere lo stesso effetto utilizzando la funzione che adesso definiamo:

```
def varauto (a):
  for x in a.split(): globals()[x]=x

varauto('Maria Vera Carla')
print Maria
# Maria
```

J. Orendorff: Comunicazione personale, febbraio 1998.

Funzioni per una pila

Nella programmazione avanzata accade abbastanza frequentemente che si voglia utilizzare una pila (in inglese *stack*) globale, che assumiamo sia definita in un modulo `pila` che potrebbe, tra altre, contenere le seguenti istruzioni. Si noti che non abbiamo bisogno nè di dichiarare la pila come globale nè di usare la tecnica dell'esplicito riferimento a un modulo.

```
pila=[]

# Aggiunge gli argomenti alla pila.
def poni (*v): pila.extend(v)

# Toglie gli ultimi k elementi
# dalla pila. Per k=1 restituisce
# l'ultimo elemento della pila,
# altrimenti la lista degli ultimi
# k elementi.
def togli (k=1):
  if k==1: return pila.pop()
  v=[]
  for i in xrange(k):
    v.append(pila.pop())
  v.reverse(); return v

# Togli gli ultimi due elementi dalla
# pila e aggiunge alla pila la
# loro somma.
def add ():
  a,b=togli(2); pila.append(a+b)
```

Nel programma principale potremmo adesso usare le istruzioni

```
import pila

pila.poni(4,3,5,2,10)
print pila.pila
# [4, 3, 5, 2, 10]

print pila.togli()
# 10
print pila.pila
# [4, 3, 5, 2]

pila.add()
print pila.pila
# [4, 3, 7]
```

eval

Se `E` è una stringa che contiene un'espressione valida di Python (ma non ad esempio un'assegnazione), `eval(E)` è il *valore* di questa espressione. Esempi:

```
u=4
print eval('u*u+7')
# 23

def f (x): return x+5

print eval('f(2)+17')
# 24

print eval('f(u+1)')
# 10

def sommaf (F,x,y):
  f=eval(F); return f(x)+f(y)

def cubo (x): return x*x*x

print sommaf('cubo',2,5)
# 133
```

Si noti che avremmo anche potuto definire

```
def sommaf (F,x,y):
  f=globals()[F]; return f(x)+f(y)
```

exec

Se la stringa `a` contiene istruzioni di Python valide, queste vengono eseguite con `exec(a)`. `exec`, a differenza da `eval`, non restituisce un risultato. Esempi:

```
a='x=8; y=6; print x+y'

exec(a)
# 14
```

`exec` è utilizzato naturalmente soprattutto per eseguire comandi che vengono costruiti durante l'esecuzione di un programma; usato con raziocinio permette metodi avanzati e, se si vuole, lo sviluppo di meccanismi di intelligenza artificiale.

Consideriamo le istruzioni

```
x=0; comando='x=17'

def f ():
  global x
  exec comando

f(); print x
# 0
```

Come mai - nonostante che `x` in `f` sia `global`? La ragione è questa: `global` è una (anzi l'unica) direttiva per il compilatore e riguarda solo l'esecuzione in cui viene chiamata. `exec` fa ripartire il compilatore e quindi bisogna ripetere il `global` nell'espressione a cui si applica `exec`. Dovremmo quindi scrivere:

```
x=0; comando='global x; x=17'

def f ():
  exec comando

f(); print x
# 17
```

execfile

`nomef` sia il nome di un file che contenga istruzioni di Python. Allora con `execfile(nomef)` queste istruzioni vengono eseguite. La differenza pratica principale con `import` è che i nomi in `nomef` valgono come se fossero nomi del file chiamante, mentre con `import` bisogna aggiungere il prefisso corrispondente al modulo definito dal file. Essenzialmente l'effetto di `execfile` è come se avessimo letto il contenuto del file in una stringa `a` ed effettuato il comando `exec(a)`.

Talvolta `execfile` viene utilizzato per leggere parametri di configurazione da un file, ad esempio un file il cui nome appare tra gli elementi di `sys.argv`.

---

sort

Sappiamo dalla pagina 4 che il metodo `sort` permette di ordinare una lista. Per una lista `a` la sintassi completa è

```
a.sort(cmp=None,key=None,reverse=False)
```

Se non reimpostiamo il parametro `cmp`, il Python utilizza una funzione di confronto naturale, essenzialmente l'ordine alfabetico (inteso in senso generale). `key` è una funzione che viene applicata ad ogni elemento della lista prima dell'ordinamento; lasciando `key=None`, gli elementi vengono ordinati così come sono. `reverse=True` implica un ordinamento in senso decrescente.

Come esempio dell'uso del parametro `key` assumiamo che vogliamo ordinare una lista di numeri tenendo conto dei valori assoluti:

```
a=[1,3,-5,0,-3,7,-10,3]
a.sort(key=abs)
print a
# [0, 1, 3, -3, 3, -5, 7, -10]
```

Similmente possiamo usare come elementi di confronto i resti modulo 100; ciò è equivalente naturalmente a considerare solo le ultime due cifre di un numero naturale:

```
a=[3454,819,99,4545,716,310]
a.sort(key=lambda x: x%100)
print a
# [310, 716, 819, 4545, 3454, 99]
```

Utilizzando `key=string.lower` possiamo ordinare una lista di stringhe alfabeticamente, prescindendo dalla distinzione tra minuscole e maiuscole.
Impostando `cmp=f` è anche possibile utilizzare una propria funzione di confronto `f`. Questa deve essere una funzione di due variabili `x,y` che restituisce -1 (o un numero negativo) se consideriamo `x` minore di `y`, 0 se i due numeri sono considerati uguali, 1 (o un numero positivo) se consideriamo `x` maggiore di `y`. La funzione dovrebbe soddisfare la condizione `f(x,x)==0` e indurre una relazione transitiva.

Assumiamo ad esempio che consideriamo una lista `a` minore di una lista `b` se `a` ha meno elementi di `b`:

```
def conflun (a,b):
  if len(a)<len(b): return -1
  if len(a)==len(b): return 0
  return 1

v=[[6,2,0],[9,1],[4,5,8,3],[3,1,7]]
v.sort(cmp=conflun)
print v
# [[9,1],[6,2,0],[3,1,7],[4,5,8,3]]
```

Naturalmente in questo caso lo stesso effetto l'avremmo potuto ottenere con

```
v.sort(key=len)
```

Infatti molto spesso nella pratica (e teoricamente sempre) è più conveniente impostare il parametro `key` piuttosto di impostare `cmp`.

La sintassi indicata è disponibile solo dalla versione 2.4 di Python. Nella 2.3 i parametri corrispondenti a `key` e `reverse` non esistono, la funzione di confronto può essere usata ma senza che venga indicato il nome del parametro; quindi in 2.3 dovremmo scrivere

```
v.sort(conflun)
```

Il modulo time

Questo modulo contiene alcune funzioni per tempo e data, di cui elenchiamo le più importanti.

`time.time()`	Tempo in secondi, con una precisione di due cifre decimali, percorso dal primo gennaio 1970 (PG70).
`time.localtime()`	Tupla temporale che corrisponde all'ora attuale.
`time.localtime(s)`	Tupla temporale che corrisponde ad s secondi dopo il PG70.
`time.ctime()`	Stringa derivata da `localtime()`.
`time.ctime(s)`	Stringa derivata da `localtime(s)`.
`time.strftime`	Output formattato di tempo e ora. Vedere i manuali, ad esempio il compendio di Alex Martelli, pagg. 247-248.
`time.sleep(s)`	Aspetta s secondi, ad esempio `time.sleep(0.25)`.

Esempi:

```
t=time.localtime()
print t
# (2006, 2, 8, 19, 48, 23, 2, 39, 0)
# anno, mese, giorno, ora, minuti, secondi,
# giorno della settimana (0 = lunedi'),
# giorno nell'anno, 1 se tempo estivo.

print time.strftime('%d %b %Y, %H:%M',t)
# 08 Feb 2006, 19:54

print time.ctime()
# Wed Feb  8 19:50:20 2006

print time.time() # Meglio time.clock().
# 1139424643.05
for i in range(1000000): pass
print time.time() # Meglio time.clock().
# 139424643.23
```

CORSO DI PROGRAMMAZIONE

Generatori

Generatori sono oggetti simili a iteratori, ma più generali, perché possono essere creati mediante apposite funzioni per cui un generatore può generare successioni anche piuttosto complicate.

Il modo più semplice per creare un generatore è, nella sintassi, molto simile alla maniera con cui a pagina 9 abbiamo creato una lista mediante un map implicito: dobbiamo soltanto sostituire le parentesi quadre con parentesi tonde:

```
a=[n*n for n in xrange(8)]
print a
# [0, 1, 4, 9, 16, 25, 36, 49]

a=(n*n for n in xrange(8))
print a
# <generator object at 0x596e4c>

for k in xrange(4): print a.next(),
# 0 1 4 9
```

Per creare un generatore possiamo anche definire una funzione in cui al posto di un `return` appare un'istruzione `yield`. Ogni volta che il generatore viene invocato, mediante il metodo `next` o nei passaggi di un ciclo, viene fornito il prossimo elemento della successione; l'esecuzione dell'algoritmo viene poi *fermata* fino alla prossima invocazione. Con alcuni esempi il meccanismo diventa più comprensibile. Definiamo prima ancora un generatore di numeri quadratici:

```
def quadrati ():
  n=0
  while 1:
    yield n*n; n+=1

q=quadrati()
for k in xrange(8): print q.next(),
# 0 1 4 9 16 25 36 49
```

In questo caso abbiamo generato addirittura *una successione infinita!* Infatti ogni chiamata `q.next()` fornirebbe un nuovo quadrato. Possiamo anche creare una successione infinita di fattoriali:

```
def genfatt ():
  f=1; i=2
  while 1: yield f; f*=i; i+=1

fattoriali=genfatt()

for k in xrange(8):
  print fattoriali.next(),
# 1 2 6 24 120 720 5040 40320
```

Spesso però si vorrebbe una successione finita, simile a un `xrange`, da usare in un ciclo `for`. Allora possiamo modificare l'esempio nel modo seguente:

```
def quadrati (n):
  for k in xrange(n): yield k*k

q=quadrati(8)
for x in q: print x,
# 0 1 4 9 16 25 36 49
```

In modo simile possiamo creare un generatore per i numeri di Fibonacci, imitando la funzione vista a pagina 13:

```
def generafib (n):
  a=0; b=1
  for k in xrange(n):
    a,b=a+b,a
    yield a

fib=generafib(9)
for x in fib: print x,
# 1 1 2 3 5 8 13 21 34
```

Se la successione contenuta in un generatore è finita, può essere trasformata in una lista:

```
fib=generafib(9)
print list(fib)
# [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Un `for` eseguito su un generatore lo svuota:

```
fib=generafib(9)
for x in fib: print x,
print list(fib)
# []

fib=generafib(9)
for k in xrange(5):
  print fib.next(),
# 1 1 2 3 5
print
print list(fib)
# [8, 13, 21, 34]
```

Successioni infinite possono talvolta sostituire variabili globali. Assumiamo ad esempio che in un programma interattivo ogni volta che l'utente lo richieda vorremmo che venga creato un nuovo oggetto, come un elemento grafico o una nuova scheda per un libro che viene aggiunta al catalogo di una biblioteca, con un unico numero che lo identifica. Invece di mantenere una variabile globale che ogni volta viene aumentata, possiamo definire un generatore all'incirca in questo modo:

```
def generanumeri ():
  n=0
  while 1: yield n; n+=1

numeri=generanumeri

def schemadiprogramma ():
  while 1:
    e=evento()
    if e.azione=='fine': break
    if e.azione=='inserimento':
      n=numeri.next()
      inserisci(e.libro,n)
```

Alcuni operatori per sequenze, ad esempio `max`, possono essere applicati anche a generatori. Dopo l'uso però il generatore risulta vuoto:

```
cos=math.cos
g=(cos(x) for x in (0.9,0.1,0.4,0.4))
print max(g)
# 0.995004165278
print list(g)
# []
```

In questo numero

readline

Per leggere un file di testo, normalmente conviene usare la funzione `leggifile` definita a pagina 22 con cui otteniamo un'unica stringa che corrisponde all'intero contenuto del file. Questa stringa poi la elaboreremo con le funzioni per le stringhe fornite dal Python. In questo modo possiamo anche suddividere il testo in righe.

Quando lavoriamo con raccolte di dati di grandi dimensioni, dobbiamo però caricare in memoria molti megabyte. Ciò può essere evitato con l'uso di un generatore.

Utilizziamo in primo luogo il metodo `readline` con cui un file può essere letto una riga alla volta. Nella funzione `leggirighe` che adesso definiamo per ogni riga viene prima controllato se essa è una stringa vuota; questo accade solo se il file non contiene altre righe. Poi però eliminiamo caratteri bianchi iniziali e finali della riga (potremmo in modo simile anche eliminare commenti) e controlliamo una seconda volta se la stringa è vuota e se ciò non accade, la forniamo al generatore con `yield`.

Nell'esempio viene poi calcolata la somma degli elementi del file, che assumiamo contenga in ogni riga un numero intero, senza che questi elementi vengano caricati in memoria:

```
def leggirighe (nome):
  f=file(nome,'r')
  while 1:
    a=f.readline()
    if not a: break
    a=a.strip()
    if a: yield a
  f.close()

righe=leggirighe('dati')
numeri=(int(x) for x in righe)
print numeri
# <generator object at 0xf7d6ec>

def somma (v):
  s=0
  for x in v: s+=x
  return s

print somma(numeri)
# 8183 (ad esempio)
```

Classi

Una *classe* è uno schema di struttura composta; gli *oggetti* della classe, le sue *istanze*, possiedono componenti che possono essere dati (*attributi*) e operazioni (*metodi*) eseguibili per questi oggetti. Un linguaggio di programmazione che, come il Python, fa fortemente uso di questo meccanismo è detto un linguaggio di programmazione orientata agli oggetti. In inglese si parla di *object oriented programming* (OOP). La programmazione orientata agli oggetti è particolarmente popolare nella grafica: ad esempio una finestra di testo può avere come attributi posizione, larghezza e altezza, colore dello sfondo e colore del testo, e come metodi varie operazioni di spostamento e di scrittura. Anche un oggetto geometrico, ad esempio un cerchio, può avere come attributi le coordinate del centro e il raggio, e come metodi operazioni di disegno o di spostamento.

In molti altri tipi di applicazioni la programmazione orientata agli oggetti si rivela utile, ad esempio nell'elaborazione di testi, nella definizione di tipi matematici (matrici o strutture algebriche), ecc. Bisogna però aggiungere che talvolta l'organizzazione per classi causa una certa frammentazione di un linguaggio, sia perché bisogna naturalmente ricordarsi le classi che sono state create, sia perché spesso classi simili eseguono operazioni pressoché equivalenti, eppure non sono più riconducibili a un'unica classe.

La programmazione orientata agli oggetti richiede quindi un notevole lavoro di organizzazione preliminare e molta riflessione e disciplina nella realizzazione delle classi nonché una documentazione che dovrebbe essere allo stesso tempo completa e di facile consultazione.

Come primo esempio definiamo una classe che rappresenta vettori in $\mathbb{R}^3$.

```
class vettore:
  def __init__(A,x,y,z):
    A.x=float(x); A.y=float(y)
    A.z=float(z)
  def __add__(A,B):
    return vettore(A.x+B.x,
      A.y+B.y,A.z+B.z)
  def __mul__(A,t):
    return vettore(A.x*t,
      A.y*t,A.z*t)
  def __rmul__(A,t): return A*t
  def coeff(A): return [A.x,A.y,A.z]
```

In ogni metodo della classe il primo argomento, da noi denotato con `A`, corrisponde all'oggetto della classe a cui il metodo viene applicato. Discutiamo in dettaglio i cinque metodi definiti per la classe `vettore`.

Quando una classe contiene un metodo `__init__`, questo viene automaticamente chiamato quando un oggetto della classe viene creato con un'istruzione che nel nostro caso avrà la forma `vettore(x,y,z)`. Non sono possibili solo assegnazioni della forma `v=vettore(x,y,z)` ma, come si vede nelle definizioni di `__add__` e `__mul__`, è anche possibile definire il valore restituito da una funzione come un nuovo oggetto della classe mediante un `return vettore(x,y,z)`. Il metodo `__init__` può contenere istruzioni qualsiasi, in genere però viene utilizzato per impostare i dati dell'oggetto creato.

Un metodo `__add__` deve essere definito per due argomenti e permette l'uso dell'operatore + invece di `__add__`. Esempio:

```
a=vettore(2,3,5); b=vettore(1,7,2)

c=a+b
print c.coeff()
# [3.0, 10.0, 7.0]

c=a.__add__(b)
print c.coeff()
# [3.0, 10.0, 7.0]
```

Le istruzioni `c=a+b` e `c=a.__add__(b)` sono quindi equivalenti; naturalmente la prima è più leggibile e più facile da scrivere. Si vede che il primo argomento del metodo diventa, nella chiamata, il prefisso a cui il metodo con i rimanenti argomenti viene attaccato. In altre parole, se una classe

```
class alfa:
  ...
  def f(A,x,y): ...
  ...
```

contiene un metodo `f`, questo viene usato come in

```
u=alfa(...)
u.f(x,y)
```

In modo simile il metodo `__mul__` della nostra classe `vettore` permette l'uso dell'operatore `*`. In questo caso il secondo argomento è uno scalare con cui possiamo moltiplicare il vettore:

```
a=vettore(2,3,5)
b=a*4
print b.coeff()
# [8.0, 12.0, 20.0]
```

In verità in matematica si preferisce porre lo scalare prima del vettore, mentre nella definizione dei metodi di una classe l'oggetto chiamante viene sempre per primo. Per ovviare a questa difficoltà Python prevede un metodo `__rmul` che permette di invertire, nella notazione operazionale, l'ordine degli argomenti. In modo simile sono definiti i metodi `__radd__`, `__rdiv__`, `__rmod__`, `__rsub__`, `__rpow__`. Metodi analoghi esistono per gli operatori logici, ad esempio `__rand__`.

Il metodo `coeff` della nostra classe non è speciale e restituisce semplicemente la lista dei coefficienti di un vettore. Si osservi anche qui che un vettore `v` diventa prefisso nella chiamata:

```
print v.coeff()
```

Gli operatori ridefiniti (o *sovraccaricati*) rispettano poi le stesse priorità come gli operatori matematici omonimi. In particolare dobbiamo utilizzare parentesi solo quando lo faremmo anche in un'analoga espressione matematica, guadagnando così ancora in leggibilità:

```
a=vettore(2,3,5); b=vettore(1,7,2)
c=vettore(0,1,8); d=vettore(4,0,-1)
e=3*(a+b)+c+d
print e.coeff()
# [13.0, 31.0, 28.0]
```

Componenti di una classe o di un singolo oggetto di una classe possono essere introdotti anche al di fuori della definizione della classe come negli esempi che seguono. Non è una buona idea però, perché in pratica equivale a una ridefinizione della classe e quindi perdiamo il controllo delle strutture che il programma utilizza.

```
class punto:
  def __add__(A,B):
    return punto(A.x+B.x,A.y+B.y)

def add (A,B): return 7

punto.__add__=add

def init (A,x,y): A.x=x; A.y=y

punto.__init__=init

u=punto(3,5);
print u+u
# 7

u.f=math.cos

print u.f(0.8)
# 0.696706709347
```

Come si vede, Python permette di ridisegnare a piacere la struttura di una classe e di aggiungere componenti a singoli oggetti.

Per la stessa ragione i metodi di una classe possono riferirsi a componenti degli oggetti della classe definiti esternamente; anche questa è una possibilità che il programmatore non dovrebbe utilizzare:

```
class cerchio:
  def perimetro (A):
    return 2*A.r*math.pi

c=cerchio()
c.r=4
print c.perimetro()
# 25.1327412287
```

Se la definizione di una classe contiene istruzioni indipendenti, queste vengono direttamente eseguite, come se si trovassero al di fuori della classe:

```
class unaclasse:
  print 7
# 7
```

Anche qui bisogna valutare se l'inserimento di istruzioni non renda meno trasparente il programma. Può essere comunque utile definire valori iniziali di alcuni componenti di una classe; così ad esempio possiamo definire una classe cerchio, i cui oggetti hanno raggio 1 quando non indicato diversamente:

```
class cerchio:
  r=1
  def perimetro (A):
    return 2*A.r*math.pi

c=cerchio()
print c.perimetro()
# 6.28318530718
```

Vediamo che, come in una funzione, variabili che si trovano alla sinistra di un'assegnazione in una classe (al di fuori di un metodo), vengono considerate componenti degli oggetti della classe.

Sovraccaricamento di operatori

Gli operatori unari e binari matematici e logici possono essere sovraccaricati mediante metodi predefiniti nella sintassi, ma liberi nella definizione degli effetti, come abbiamo visto a pagina 27 per gli operatori di addizione e moltiplicazione. Elenchiamo le più importanti di queste corrispondenze:

`__add__`	+ binario
`__sub__`	- binario
`__mul__`	*
`__div__`	/
`__floordiv__`	//
`__mod__`	%
`__pos__`	+ unario
`__neg__`	- unario
`__lshift__`	<<
`__rshift__`	>>
`__and__`	&
`__or__`	|
`__xor__`	^
`__invert__`	~
`__iadd__`	+=
`__isub__`	-=
`__imul__` ecc.	*= ecc.

Il significato di `__radd__` ecc. è stato spiegato a pagina 27.

Se ai metodi della classe `vettore` a pagina 27 aggiungiamo

```
def __iadd__(A,B):
  A=A+B; return A
```

possiamo usare le istruzioni

```
v=vettore(2,3,5); w=vettore(1,0,8)
v+=w
print v.coeff()
[3.0, 3.0, 13.0]
```

Come già osservato, mentre gli operatori così sovraccaricati devono seguire la stessa sintassi degli operatori originali, il significato è del tutto libero:

```
class lista:
  def __init__(A,*v):
    A.elementi=v
  def __lshift__(A,n):
    return A.elementi[n]
  def __pos__(A):
    s=0
    for x in A.elementi:
      s+=x
    return s

a=lista(3,5,9,6,4)
print a<<2, +a
# 9 27

class cerchio:
  def __add__(A,B): print '***'
  def __pos__(A): print 'Ciao.'

u=cerchio()
u+u
# ***
+u
# Ciao.
```

Come vediamo, non è nemmeno necessario che questi metodi restituiscano un oggetto della classe (infatti, nell'ultimo esempio entrambi i metodi visualizzano una stringa sullo schermo e restituiscono `None`).

Per la classe `vettore` potremmo così definire un prodotto scalare e sovraccaricare ad esempio l'operatore `&`:

```
def __and__(A,B):
  return A.x*B.x+A.y*B.y+A.z*B.z

a=vettore(3,2,5); b=vettore(6,1,2)
print a & b
# 30
```

Se avessimo voluto usare lo stesso algoritmo come a pagina 7, avremmo dovuto lavorare con le liste dei coefficienti:

```
def __and__(A,B):
  s=0
  for x,y in zip(A.coeff(),B.coeff()):
    s+=x*y
  return s
```

In questo caso il calcolo diretto impiegato nella prima soluzione sembra più efficiente.

__str__

Il metodo speciale `__str__` può essere usato per ridefinire l'istruzione `print` per gli oggetti di una classe. Esso dovrebbe restituire una stringa che viene poi visualizzata da `print`. Per la classe `vettore` a pagina 27 con

```
a=vettore(3,5,7)
print a
```

otteniamo

```
<__main__.vettore instance at 0x25d9cc>
```

Se però aggiungiamo `__str__` ai metodi della classe con

```
def __str__(A):
  return '%.2f %.2f %.2f' \
    %(A.x,A.y,A.z)
```

possiamo usare `print` per visualizzare i coefficienti del vettore:

```
a=vettore(3,5,7)
print a
# 3.00 5.00 7.00
```

Metodi impliciti

Siccome la natura degli argomenti di una funzione in Python non deve essere nota all'atto della definizione, possiamo definire funzioni che utilizzano componenti di una classe senza che questa appaia esplicitamente nella funzione. Chiamiamo tali funzioni *metodi impliciti*.

Possiamo ad esempio creare una funzione che calcola il prodotto scalare per due oggetti della nostra classe `vettore`, semplicemente utilizzando i loro coefficienti:

```
def scalare (a,b):
  return a.x*b.x+a.y*b.y+a.z*b.z

a=vettore(1,3,9); b=vettore(7,2,4)
print scalare(a,b)
# 49.0
```

La funzione `scalare` è definita al di fuori della classe e formalmente non esiste alcun legame tra la funzione e la classe!

__call__

Un significato speciale ha anche il metodo `__call__`. Quando definito per una classe, esso permette di usare gli oggetti della classe come funzioni. Aggiungiamo questo metodo alla classe `vettore`:

```
def __call__(A,x,y,z):
  A.x=float(x); A.y=float(y)
  A.z=float(z)
```

Adesso con `v(x,y,z)` possiamo ridefinire i coefficienti del vettore `v`:

```
a=vettore(3,5,7)
print a
# 3.00 5.00 7.00
a(8,2,1)
print a
# 8.00 2.00 1.00
```

Le stesse istruzioni usate in `__call__` appaiono anche in `__init__`. È preferibile usarle una volta sola, e quindi i due metodi per la classe `vettore` possono essere riscritti nel modo seguente:

```
def __init__(A,x,y,z): A(x,y,z)
def __call__(A,x,y,z):
  A.x=float(x); A.y=float(y)
  A.z=float(z)
```

Anche nel caso di `__call__` naturalmente si è completamente liberi nella scelta delle operazioni che il metodo deve effettuare. In una libreria grafica potremmo ad esempio usare sistematicamente i metodi `__call__` per l'esecuzione delle operazioni di disegno oppure anche solo per preparare la struttura geometrica di una figura, riservando ad esempio l'operatore + per effettuare il disegno, così come nel corso di Algoritmi abbiamo spesso definito una figura come una funzione (in R) che genera un insieme di punti che successivamente può essere disegnato con `lines`:

```
class cerchio:
  def __call__(A,parametri):
    ...
    A.punti=...
  def __pos__(A):
    disegna(A.punti)

a=cerchio()
a(x,y,r); +a
a(u,v,s); +a
```

Sintassi modulare

Se `f` è un metodo della classe `vettore` e se `a` è un oggetto della classe, le espressioni `a.f(x)` e `vettore.f(a,x)` sono equivalenti. Ciò mostra che in Python classi e moduli sono molto simili tra di loro: entrambi sono essenzialmente collezioni di funzioni.

```
a=vettore(7,3,5); b=vettore(2,8,1)
print vettore.__add__(a,b)
# 9.00 11.00 6.00
```

Il costruttore __init__

Il metodo speciale `__init__` si chiama *costruttore* della classe. Una classe può possedere al massimo un costruttore; questo può anche mancare. Usando argomenti opzionali in `__init__` si possono facilmente definire operazioni di inizializzazione differenti per la stessa classe. Esempio:

```
class punto:
  def __init__ (A,x=0,y=0): A(x,y)
  def __call__ (A,x,y):
    A.x=x; A.y=y
  def __str__(A):
    return '%.2f %.2f' %(A.x,A.y)

p=punto()
print p
# 0.00 0.00

p(7,4); print p
# 7.00 4.00
```

Metodi vettoriali

Si possono definire alcuni metodi speciali che permettono di estendere agli oggetti di una classe gli operatori vettoriali previsti per liste. Nell'elenco `a` sia un oggetto di una tale classe.

`__getitem__`	Per calcolare `a[k]`.
`__setitem__`	Per impostare `a[k]`.
`__contains__`	Verifica `x in a`.
`__len__`	Permette `len(a)`.

Potremmo ad esempio aggiungere alla classe `vettore` i metodi

```
  def __getitem__(A,k):
    return A.coeff()[k]
  def __setitem__(A,k,val):
    if k==0: A.x=val
    elif k==1: A.y=val
    elif k==2: A.z=val
  def __contains__(A,x):
    return x in A.coeff()
  def __len__(A): return len(A.coeff())
```

che possono essere così usati:

```
a=vettore(7,3,5)
for x in a: print x,
# 7.0 3.0 5.0

print
a[2]=99; print a, len(a)
# 7.00 3.00 99.00 3
```

__cmp__

Se una classe possiede un metodo `__cmp__`, questo viene utilizzato nei confronti tra oggetti della classe. Una tale funzione di confronto deve essere definita come l'argomento opzionale `cmp` di `sort` (cfr. pagina 25).

```
class punto:
  def __init__(A,x,y): A.x,A.y=x,y
  def __cmp__(A,B):
    return cmp((A.x,A.y),(B.x,B.y))

p=punto(3,5); q=punto(7,1)
print p<=q
# True
```

Sottoclassi

Dopo

```
class animale: ...
```

si può definire una sottoclasse con

```
class leone (animale): ...
```

In principio la sottoclasse eredita tutti i componenti della classe superiore che non vengono ridefiniti nella sottoclasse. L'introduzione di sottoclassi può però rendere complicato l'assetto di un programma e l'apprendimento delle classi create e dei legami tra di loro equivale facilmente a dover apprendere un nuovo linguaggio di programmazione.

Il modulo operator

Questo modulo contiene delle funzioni che permettono di utilizzare gli operatori standard di Python ad esempio in istruzioni `map` e `reduce`, senza dover definire apposite funzioni o espressioni lambda:

```
def somma (*a):
  return reduce(operator.add,a,0)

def prod (*a):
  return reduce(operator.mul,a,1)

print somma(7,5,8,3)
# 23

print prod(7,5,8,3)
# 840

print map(operator.sub,[8,9,4],[6,1,2])
# [2, 8, 2]

print map(operator.neg,[3,5,8,-9,-3,0])
# [-3, -5, -8, 9, 3, 0]
```

Metodi di confronto

Anche gli operatori di confronto possono essere sovraccaricati:

`__eq__`	`==`
`__ne__`	`!=`
`__le__`	`<=`
`__lt__`	`<`
`__ge__`	`>=`
`__gt__`	`>`

La semantica è anche qui del tutto libera:

```
class numero:
  def __init__(A,x): A.x=x
  def __lt__(A,B): print 'Ciao.'

a=numero(6); b=numero(9)
a<b
# Ciao.
```

dir

Con `dir(alfa)` si ottengono i nomi definiti per ogni oggetto alfa che possiede un attributo `__dict__`.

Attributi standard

`F` sia un modulo, una classe o una funzione. Allora sono definiti i seguenti attributi:

`F.__name__`	La stringa `F`.
`F.__dict__`	Elenco degli attributi e dei loro valori per `F`, in forma di un dizionario.
`F.__dir__`	Elenco degli attributi, senza i loro valori.

type e isinstance

Ogni oggetto di Python possiede un tipo univoco che si ottiene tramite la funzione `type`. Con `isinstance` si può verificare se un oggetto appartiene ad una classe.

```
print type(77)
# <type 'int'>
print isinstance(77,int)
# True

print type([3,5,1])
# <type 'list'>
print isinstance([3,5,1],tuple)
# False

print type('alfa')
# <type 'str'>

def f(x): pass
print type(f)
# <type 'function'>

a=vettore(8,0,2)
print type(a)
# <type 'instance'>
print isinstance(a,vettore)
# True
```

Il modulo types

Importando il modulo `types` si può verificare, se un oggetto è di un tipo predefinito.

Esempi:

```
if type(a)==types.IntType: ...
if type(a)==types.FloatType: ...
if type(a)==types.ComplexType: ...
if type(a)==types.FunctionType: ...
if type(a)==types.LambdaType: ...
if type(a)==types.StringType: ...
if type(a)==types.ListType: ...
if type(a)==types.DictionaryType: ...
if type(a)==types.FileType: ...
if type(a)==types.ClassType: ...
```

Una lista dei tipi predefiniti la si ottiene con `dir(types)`.

A pagina 18 abbiamo utilizzato `type` nel Minilisp.

„Python is an object-oriented programming language. Unlike some other object-oriented languages, Python doesn't force you to use the object-oriented paradigm exclusively. Python also supports procedural programming with modules and functions, so you can select the most suitable programming paradigm for each part of your program.“ (Martelli, pag. 69)

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 9

L'insieme delle parti

Situazione 30.1. $X, Y, \ldots$ siano insiemi.
A partire dalla situazione 30.9 X ed Y saranno insiemi finiti.

Osservazione 30.2. *Si possono definire i numeri naturali induttivamente con:*

$$\begin{aligned} 0 &:= \emptyset \\ 1 &:= \{0\} = \{\emptyset\} \\ 2 &:= \{0,1\} = \{\emptyset, \{\emptyset\}\} \\ 3 &:= \{0,1,2\} \\ &\ldots \\ n+1 &:= \{0,1,\ldots,n\} \\ &\ldots \end{aligned}$$

Useremo spesso l'uguaglianza tra 2 *e* $\{0,1\}$. *Non è un'abbreviazione!*

Definizione 30.3. Denotiamo con Y^X l'insieme di tutte le funzioni $X \longrightarrow Y$.

Osservazione 30.4. *Se* X *ed* Y *sono finiti, allora* $|Y^X| = |Y|^{|X|}$.
In particolare $|2^X| = 2^{|X|}$.

Definizione 30.5. Per $f : X \longrightarrow Y$ ed $y \in Y$ sia

$$(f = y) := \{x \in X \mid f(x) = y\} = f^{-1}(y)$$

e similmente per $B \subset Y$

$$(f \in B) := f^{-1}(B)$$

Definizione 30.6. $\mathcal{P}(X) := \{A \mid A \subset X\}$ si chiama l'*insieme delle parti* di X.

Nota 30.7. Per ogni sottoinsieme $A \subset X$ possiamo definire una funzione $1_A \in 2^X$ ponendo

$$1_A(x) := \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

1_A si chiama la *funzione caratteristica* di A; più precisamente si dovrebbe scrivere $1_{A,X}$. Dalla definizione segue

$$A = (1_A = 1)$$

Sia viceversa data una funzione $f \in 2^X$; allora f è univocamente determinata dall'insieme $(f = 1)$, poiché

$$f(x) = \begin{cases} 1 & \text{se } x \in (f = 1) \\ 0 & \text{se } x \notin (f = 1) \end{cases}$$

essendo 0 e 1 gli unici possibili valori di f. Quindi $f = 1_{(f=1)}$

Da ciò si deduce facilmente una biiezione naturale tra $\mathcal{P}(X)$ e 2^X, in cui un sottoinsieme A corrisponde alla funzione 1_A e una funzione f all'insieme $(f = 1)$.

$\mathcal{P}(X)$ e 2^X verranno spesso identificati.

Nota 30.8. Questa unificazione della notazione, precisata nell'articolo a lato sul prodotto cartesiano, porta a un'inconsistenza non più risolubile. Infatti, se n è un numero naturale, 2^n è da un lato un prodotto cartesiano n-plo, dall'altro lato vorremmo in questo modo denotare anche l'n-esima potenza aritmetica di 2. Ma queste due interpretazioni non coincidono; ad esempio aritmeticamente

$$2^2 = 4 = \{0,1,2,3\}$$

mentre come prodotto cartesiano

$$2^2 = \{(0,0),(0,1),(1,0),(1,1)\}$$

e, come si vede nella nota 30.12, le coppie $(0,0), \ldots$ sono oggetti molto più complicati dei numeri $0, 1, 2, 3$.

Dobbiamo quindi derivare dal contesto la corretta interpretazione di espressioni della forma 2^n. Un esempio è già la nota 30.11.

Funzioni booleane

Situazione 30.9. X sia un insieme finito con n elementi, $Y, Z, \ldots$ altri insiemi finiti.

Definizione 30.10. Una *funzione booleana* (semplice e espressa sopra X) è una funzione $\alpha : 2^X \longrightarrow 2$.

α è quindi un elemento di 2^{2^X}. Identificando 2^{2^X} con $\mathcal{P}(\mathcal{P}(X))$, vediamo che α non è altro che un insieme di sottoinsiemi di X, cioè un insieme $\alpha \subset \mathcal{P}(X)$. Useremo spesso questa interpretazione.

È chiaro che per $|X| = |Z|$ la struttura di 2^{2^X} è isomorfa a quella di 2^{2^Z}, perciò è la cardinalità n che conta.

In particolare potremmo anche assumere che $X = n$; siccome gli elementi di 2^n sono n-ple di elementi di 2, si dice anche che α è una funzione booleana di n variabili.

Nella teoria dei sistemi dinamici booleani si considerano anche funzioni $2^X \longrightarrow 2^Y$; queste funzioni sono dette *funzioni booleane multiple*.

Nota 30.11. Quante funzioni booleane ci sono? Se l'insieme X possiede n elementi, allora $|2^{2^X}| = 2^{2^n}$, dove l'espressione a destra è da interpretare numericamente (cfr. nota 30.8), quindi il numero delle funzioni booleane cresce nel modo seguente con il crescere di n:

n	$\lvert 2^{2^X} \rvert$
0	$2^1 = 2$
1	$2^2 = 4$
2	$2^4 = 16$
3	$2^8 = 256$
4	$2^{16} = 65536$
5	$2^{32} = 4294676296$
6	2^{64}
7	2^{128}

Si vede che ogni volta che si aumenta la cardinalità di X di uno, il numero delle funzioni booleane espresse sopra X diventa il quadrato di quello precedente. Esistono quindi moltissime funzioni booleane, già per 5 variabili sono più di 4 miliardi, e per 6 variabili diventano 18 miliardi di miliardi.

In questo numero

Il prodotto cartesiano

Nota 30.12. Una funzione è spesso definita come tripla, i cui componenti sono il dominio, il codominio e il grafico della funzione. Però cos'è una tripla?

D'altra parte vorremmo poter considerare coppie e triple come funzioni, vorremmo cioè che $X \times Y$ sia l'insieme di tutte le funzioni $f : 2 \longrightarrow X \cup Y$ tali che $f(0) \in X$ e $f(1) \in Y$ e che $X \times Y \times Z$ sia l'insieme di tutte le funzioni $f : 3 \longrightarrow X \cup Y \cup Z$ tali che in più $f(2) \in Z$. In questo caso poi avremmo automaticamente uguaglianze $X \times X = X^2$ e $X \times X \times X = X^3$ che sono in accordo con la definizione 30.3.

Se utilizziamo però le triple per definire le funzioni, ci troviamo in un circolo vizioso.

Ne possiamo venir fuori utilizzando nella definizione di funzione un altro concetto di tripla che garantisce che una tripla sia univocamente determinata dai suoi componenti. Definiamo quindi per tre oggetti matematici a, b, c la *coppia ausiliaria* o *logica* $\langle a, b\rangle := \{a, \{a, b\}\}$ e la *tripla ausiliaria* o *logica* $\langle a, b, c\rangle := \langle\langle a, b\rangle, c\rangle$.

A questo punto possiamo riformulare la definizione di funzione, sostituendo il termine *tripla* con *tripla logica* e il termine *coppia* che appare nella definizione del grafico con *coppia logica* e definire i prodotti cartesiani come sopra. Elementi di $X \times Y$ saranno detti *coppie* e scritti nella forma (x, y) - ma in verità dovremmo addirittura scrivere $(x, y)_{X\times Y}$ - e similmente per le triple. La coppia (x, y) è quindi quella funzione $f : 2 \longrightarrow X \cup Y$ per cui $f(0) = x, f(1) = y$.

Formalmente questa coppia è un oggetto piuttosto complicato:

$$(x, y) = \langle 2, X \cup Y, \{\langle 0, x\rangle, \langle 1, y\rangle\}\rangle$$

Se $X = Y$, la coppia $(x, y) \in X^2$ è data da

$$(x, y) = \langle 2, X, \{\langle 0, x\rangle, \langle 1, y\rangle\}\rangle$$

cosicché ad esempio

$$2^2 = \{(0,0),(0,1),(1,0),(1,1)\}$$

con $(0, 0) = \langle 2, 2, \{\langle 0, 0\rangle, \langle 1, 0\rangle\}\rangle$

ecc. Vediamo che il concetto di coppia è molto profondo è difficile. E i problemi non sono finiti, come si vede dalla nota 30.8.

Vita è codifica

Abbiamo visto che esistono più di 18 miliardi di miliardi di funzioni booleane di 6 variabili; ma gli ingegneri studiano funzioni booleane con 30 o anche 100 variabili. Di queste ultime ne esistono $2^{2^{100}}$, un numero che la matematica ci permette di scrivere, mentre non siamo in nessun modo in grado di elencare o anche solo lontanamente immaginare tutte queste funzioni, perché, se pensiamo che $N := 2^{100}$ è probabilmente superiore al numero di oggetti nell'universo, di quelle funzioni ne esistono addirittura 2^N. Il mondo delle funzioni booleane è quindi di una incredibile ricchezza, è un mondo di strumenti intellettuali a cui possiamo attingere per descrivere praticamente tutti gli oggetti, fenomeni e leggi del mondo reale. Per questo la teoria delle funzioni booleane è un mondo di ricerca intensissimo, non solo da parte degli ingegneri che cercano di studiare e semplificare i circuiti digitali, ma anche ad esempio, in modo sempre crescente, nella medicina, dove si cerca di descrivere i meccanismi biochimici e patologici con modelli booleani.

D'altra parte la vita stessa ne è un esempio. Abbiamo l'impressione che un organismo è tanto più vitale quanto più è ramificato. Troviamo strutture ramificate negli alberi, nell'organizzazione del sistema vasocircolatorio, dei polmoni e del sistema nervoso. Persino quando in un oggetto inanimato o su un pianeta lontano scopriamo sistemi ramificati questo ci dà una sensazione di vita anche quando forse è solo una somiglianza, un ricordo o una struttura potenziale.

Quindi la ramificazione organizzata è una delle basi della vita. Anche un programma in un linguaggio di programmazione comincia ad acquistare vita solo quando ci sono ramificazioni, espresse da istruzioni `if ... else`. La vita sul nostro pianeta è organizzata da un codice, il codice genetico, e, in maniera più nascosta e molto più complessa, da sistemi booleani che descrivono le regole della vita o del particolare organismo che in un certo modo è definito da quelle regole. Senza questa codifica la vita non avrebbe quella forza interna, quella pertinacia e quella capacità di rigenerarsi e di ottimizzarsi che la fa apparire superiore al mondo puramente fisico, e nemmeno quella capacità interattiva che sta alla base ad esempio dei fenomeni sociali.

Quindi la vita è codifica.

Sistemi di insiemi

Abbiamo osservato nella definizione 30.10 che una funzione booleana è la stessa cosa come un insieme di sottoinsiemi di un insieme finito, un *sistema di insiemi*. Come tali appaiono nella combinatoria (dove spesso i sistemi di insiemi vengono chiamati *ipergrafi*) e in molti altri campi della matematica e spesso la sola traduzione di enunciati da un linguaggio all'altro porta a interpretazioni sorprendenti o utili. Ad esempio una topologia su un insieme X può essere definita indicando l'insieme α degli aperti; se X è finito, abbiamo una funzione booleana. Le funzioni booleane monotone (o piuttosto le funzioni antitone) possono essere identificate con i *complessi simpliciali*. Nell'algebra universale si studiano *sistemi di chiusi*, anch'essi non sono altro che sistemi di insiemi con particolari proprietà.

Spazi topologici finiti

Nota 31.1. In uno spazio topologico X due intorni di un punto x si intersecano sempre in un altro intorno di quel punto; per induzione segue immediatamente che l'intersezione di un numero finito di intorni di x è ancora un intorno di x. Se lo spazio stesso è finito, l'intersezione U_x di tutti gli intorni di x è un intorno di x, necessariamente il più piccolo - infatti gli intorni di x sono esattamenti quei sottoinsiemi dello spazio che contengono U_x. Usando una notazione che introdurremo fra breve ciò significa che l'insieme degli intorni di x coincide con U_x^+.

Sempre nell'ipotesi che X sia finito, introduciamo su X una relazione $\leq$ ponendo

$$x \leq y : \iff U_y \subset U_x$$

È chiaro che ciò accade se e solo se $y \in U_x$ e che questa relazione è riflessiva e transitiva e costituisce quindi un quasiordine su X; inoltre (essendo $U_x^+ = U_y^+$ se e solo se $U_x = U_y$) la relazione $\leq$ è un ordine parziale (cioè anche antisimmetrica) se e solo se lo spazio topologico X soddisfa l'assioma di separazione T_0 che chiede che due punti che hanno gli stessi intorni sono uguali.

Sia viceversa dato un quasiordine $\leq$ su un insieme finito X. Se definiamo $U_x := \{y \in X \mid y \geq x\}$ e come intorni di x prendiamo tutti i sottoinsiemi di X che contengono U_x, otteniamo uno spazio topologico. Si dimostra adesso, con un po' di pazienza, che le due costruzioni sono una l'inversa dell'altra e che gli omomorfismi di insiemi quasiordinati, cioè le funzioni monotone, sono esattamente le funzioni continue nell'interpretazione topologica.

Troviamo così che spazi topologici finiti e insiemi quasiordinati finiti sono, a tutti gli effetti, la stessa cosa. Gli insiemi quasiordinati finiti sono a loro volta uno dei concetti più fondamentali della combinatoria (comprendono ad esempio i *reticoli*, ma anche strutture molto più generali).

Nota 31.2. Sia X un insieme finito. Un insieme di insiemi $\alpha \subset \mathcal{P}(X)$, cioè una funzione booleana espressa su X, coincide, come si impara nei corsi di topologia, con l'insieme degli aperti di una topologia su X se e solo se gode delle seguenti proprietà:

(1) $\emptyset \in \alpha$.
(2) $X \in \alpha$.
(3) $U, V \in \alpha \implies U \cap V \in \alpha$.
(4) $U, V \in \alpha \implies U \cup V \in \alpha$.

Attenzione: L'assioma (4) può essere formulato in questo modo solo se X è finito; nel caso generale bisogna chiedere che un'unione arbitraria di aperti sia ancora un aperto.

Vediamo quindi che la teoria delle funzioni booleane contiene (come una piccola parte!) la teoria degli spazi topologici finiti che, come abbiamo appena visto, non è banale ma di grande importanza.

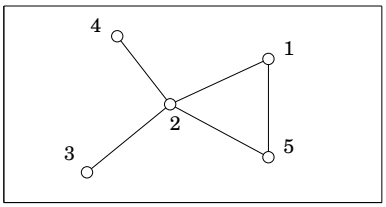
Se si rinuncia all'assioma (3) o all'assioma (4) si ottiene una teoria ancora più generale, quella dei *sistemi di aperti* o quella, equivalente (per dualità) e più spesso usata in molti contesti dell'algebra universale e della combinatoria, dei *sistemi di chiusi*. Queste teorie sono quindi, per insiemi finiti, anch'esse contenute nella teoria delle funzioni booleane. Oltre che nella matematica pura vengono da una quindicina di anni applicate ad esempio a problemi di classificazione in sociologia, biochimica e virologia.

Grafi

Un ipergrafo α tale che $|A| = 2$ per ogni $A \in \alpha$ si chiama un *grafo* (semplice). Ogni tale α dà luogo a un grafo nel senso comune come nell'esempio seguente:

$$X = \{1, 2, 3, 4, 5\}$$
$$\alpha = \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{2, 5\}\}$$

Adesso congiungiamo x ed y con una linea del grafo se e solo se $\{x, y\} \in \alpha$:

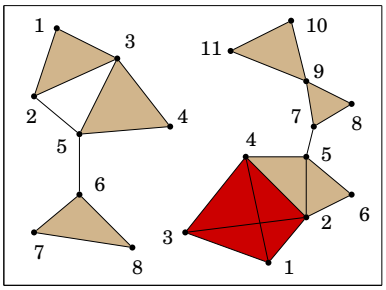


Quindi anche la teoria dei grafi fa parte della teoria delle funzioni booleane.

Complessi simpliciali

Uno dei concetti più importanti della *topologia combinatoria* o *geometrica* è il concetto di complesso simpliciale. Un *complesso simpliciale* è un sistema di insiemi α (cioè una funzione booleana) con la proprietà che ogni insieme contenuto in un elemento di α appartenga esso stesso ad α. L'importanza dei complessi simpliciali risiede nel fatto che mediante essi si possono descrivere le proprietà omotopiche di spazi topologici geometrici; per il loro studio è stata sviluppata una straordinariamente ricca teoria algebrica, detta *algebra omologica*; per questa ragione la teoria dei complessi simpliciali e le sue generalizzazioni fanno parte della *topologia algebrica*. In un certo senso quindi anche questo ramo molto sofisticato della matematica pura può essere inquadrato nella teoria delle funzioni booleane. Viceversa i risultati della topologia algebrica possono spesso essere applicati a problemi concreti o ingegneristici.

I complessi simpliciali possono essere rappresentati da figure in $\mathbb{R}^n$ come nei seguenti esempi:



Alla figura piana a sinistra possiamo associare il complesso simpliciale α che consiste degli insiemi $\{1, 2, 3\}$, $\{3, 4, 5\}$, $\{2, 5\}$, $\{5, 6\}$, $\{6, 7, 8\}$ e di tutti i loro sottoinsiemi, alla figura 3-dimensionale a destra, in cui il tetraedro con vertici 1,2,3,4 è pieno, il complesso simpliciale che consiste degli insiemi $\{1, 2, 3, 4\}$, $\{2, 4, 5\}$, $\{2, 5, 6\}$, $\{5, 7\}$, $\{7, 8, 9\}$, $\{9, 10, 11\}$ e di tutti i loro sottoinsiemi. Non è molto difficile dimostrare che ogni complesso simpliciale α può essere ottenuto da una figura come in questi esempi (detta talvolta *poliedro simpliciale*) in $\mathbb{R}^{2d+1}$, se $|A| \leq d + 1$ per ogni $A \in \alpha$.

Funzioni booleane monotone

Nota 32.1. Identifichiamo da ora in avanti sistematicamente le funzioni $\alpha : 2^X \longrightarrow 2$ con i sistemi di insiemi $\alpha \subset \mathcal{P}(X)$, usando la stessa lettera per una funzione booleana e il sistema di insiemi ad essa corrispondente. Quindi ad esempio

$$A \in \alpha \iff \alpha(A) = 1$$

Definizione 32.2. L'insieme $2 = \{0, 1\}$ è un insieme parzialmente ordinato con

$$\leq = \{(0, 0), (0, 1), (1, 1)\}$$

Ciò corrisponde, nell'identificazione tra 2 e $\mathcal{P}(1)$, all'inclusione insiemistica in $\mathcal{P}(1)$. Si noti che 2 (come insieme), con le operazioni di addizione e moltiplicazione modulo 2 (come numero!) è anche un campo, ma non è un campo ordinato, perché in un campo ordinato $a, b > 0$ deve implicare $a + b > 0$, mentre in 2 si ha $1 > 0$, ma $1 + 1 = 0$.

Più in generale l'inclusione insiemistica in $\mathcal{P}(X)$ corrisponde a un ordine parziale altrettanto naturale su 2^X dato da

$$f \leq g \iff f(x) \leq g(x) \text{ per ogni } x \in X$$

Se $A, B \subset X$, abbiamo allora

$$A \subset B \iff 1_A \leq 1_B$$

Se $X = n$, gli elementi di 2^X possono anche essere scritti come n-ple $(u_1, \ldots, u_n)$ e allora abbiamo $(u_1, \ldots, u_n) \leq (v_1, \ldots, v_n)$ se e solo se $u_i \leq v_i$ per ogni $i = 1, \ldots, n$.

Definizione 32.3. Una funzione booleana $\alpha : 2^X \longrightarrow 2$ si dice *monotona* se

$$A \subset B \Longrightarrow \alpha(A) \leq \alpha(B)$$

e *antitona* se

$$A \subset B \Longrightarrow \alpha(A) \geq \alpha(B)$$

È chiaro che α è monotona se e solo se

$$A \in \alpha \text{ e } A \subset B \Longrightarrow B \in \alpha$$

quindi se e solo se α contiene con ogni insieme anche tutti gli insiemi che contengono quell'insieme, e che α è antitona se e solo se

$$A \subset B \in \alpha \Longrightarrow A \in \alpha$$

cioè se solo se α è un complesso simpliciale!

Definizione 32.4. Per $\alpha \subset \mathcal{P}(X)$ sia

$$\neg\alpha := \mathcal{P}(X) \setminus \alpha$$

In altre parole,

$$A \in \neg\alpha \iff A \notin \alpha$$

oppure, nell'interpretazione funzionale,

$$\neg\alpha(A) = 1 \iff \alpha(A) = 0$$
$$\neg\alpha(A) = 0 \iff \alpha(A) = 1$$

È chiaro che $\neg\neg\alpha = \alpha$.

Osservazione 32.5. *Sia $\alpha \subset \mathcal{P}(X)$. Allora α è antitona (e quindi un complesso simpliciale) se e solo se $\neg\alpha$ è monotona.*

Dimostrazione. Sia $A \subset B$ e $A \in \neg\alpha$, cioè $A \notin \alpha$. Se si avesse $B \notin \neg\alpha$, si avrebbe $B \in \alpha$ e quindi $A \in \alpha$, perché α è antitona, una contraddizione. Nello stesso modo o per simmetria (sfruttando $\neg\neg\alpha = \alpha$) si ottiene l'altra direzione dell'implicazione.

Intervalli e cointervalli in $\mathcal{P}(X)$

Definizione 32.6. Per $\alpha, \beta \subset \mathcal{P}(X)$ scriviamo spesso $\alpha\beta$ invece di $\alpha \cap \beta$. Questa non è solo un'abbreviazione, ma corrisponde al fatto che nell'interpretazione funzionale all'intersezione corrisponde la moltiplicazione nell'anello 2^{2^X}.

Definizione 32.7. Per $P, R \subset X$ siano

$$P^+ := \{A \subset X \mid P \subset A\}$$
$$R^- := \{A \subset X \mid A \subset X \setminus R\}$$

Abbiamo usato la prima di queste notazioni già nella nota 31.1.

P^+ è l'insieme dei sottoinsiemi di X che contengono P, mentre

$$P^+R^- = \{A \subset X \mid P \subset A \subset X \setminus R\}$$

è l'intervallo $[P, X \setminus R]$ nell'insieme parzialmente ordinato $\mathcal{P}(X)$ determinato da P ed $X \setminus R$; cfr. nota 32.9.

Gli intervalli di $\mathcal{P}(X)$ in logica sono anche detti *termini booleani* o *monomi*.

Un intervallo che contiene esattamente un elemento è detto *semplice*.

Nota 32.8. Per $R \subset X$

$$\begin{aligned} R^- &= \{A \subset X \mid A \subset X \setminus R\} \\ &= \{A \subset X \mid R \subset X \setminus A\} \\ &= \{A \subset X \mid A \cap R = \emptyset\} \end{aligned}$$

R^- è quindi l'insieme dei sottoinsiemi di X che hanno intersezione vuota con R.
Vediamo anche che, per $A \subset X$,

$$A \in R^- \iff X \setminus A \in R^+$$

Nota 32.9. a, b, c, d siano elementi di un insieme parzialmente ordinato. Denotiamo con $[a, b]$ e $[c, d]$ gli intervalli determinati da questi elementi. Allora:

(1) $[a, b] = [c, d] \neq \emptyset \Longrightarrow a = c, b = d$.
(2) $|[a, b]| = 1 \iff a = b$.

Si noti che l'enunciato (1) vale solo per intervalli non vuoti; infatti affinche $[a, b] = \emptyset$, è sufficiente che $a \not\leq b$.

Dimostrazione. (1) In primo luogo abbiamo $a \leq b$ e $c \leq d$, altrimenti l'intervallo sarebbe vuoto.

L'uguaglianza implica che $a, b \in [c, d]$, e quindi abbiamo $c \leq a \leq b \leq d$; ma per simmetria anche $a \leq c \leq d \leq b$. Siccome abbiamo presupposto che $\leq$ sia un ordine parziale, da ciò seguono $a = c$ e $b = d$.

(2) Chiaro; bisogna però usare anche qui l'ipotesi che si tratti di un ordine parziale.

Corollario 32.10. *Siano $P, Q, R, S \subset X$ tali che $P^+R^- = Q^+S^- \neq \emptyset$.*

Allora $P = Q, R = S$.

Osservazione 32.11. *Sia $A \subset X$. Allora*

$$\{A\} = [A, A] = A^+(X \setminus A)^-$$

Corollario 32.12. *Per una funzione booleana $\sigma \subset \mathcal{P}(X)$ sono equivalenti:*

(1) σ è un intervallo semplice.
(2) σ è della forma $A^+(X \setminus A)^-$ per un sottoinsieme $A \subset X$.

In questo caso $\sigma = \{A\}$.

Definizione 32.13. Un *cointervallo* in un insieme parzialmente ordinato è il complemento di un intervallo.

Un cointervallo si dice *semplice*, se è complemento di un intervallo semplice; un cointervallo è quindi semplice se e solo se consiste di tutti gli elementi dell'insieme parzialmente ordinato tranne uno.

In logica i cointervalli di $\mathcal{P}(X)$ prendono il nome di *clausole booleane*.

Nota 32.14. Per $P, R \subset X$ abbiamo:

(1) $\neg P^+ = \{A \subset X \mid P \not\subset A\}$.
(2) $\neg R^- = \{A \subset X \mid A \cap R \neq \emptyset\}$.
(3) $\neg(P^+R^-) = \neg P^+ \cup \neg R^-$.

I cointervalli di $\mathcal{P}(X)$ coincidono quindi con gli insiemi della forma $\neg P^+ \cup \neg R^-$ con $P, R \subset X$.

Dimostrazione. (1) Per definizione di P^+.
(2) Nota 32.8.
(3) Chiaro.

Corollario 32.15. *Per una funzione booleana $\tau \subset \mathcal{P}(X)$ sono equivalenti:*

(1) τ è un cointervallo semplice.
(2) τ è della forma $\neg B^+ \cup \neg(X \setminus B)^-$ per un sottoinsieme $B \subset X$.

In questo caso $\tau = \neg\{B\}$.

Osservazione 32.16. *Siano P, Q, R, S sottoinsiemi di X. Allora*

$$P^+Q^+ = (P \cup Q)^+$$
$$R^-S^- = (R \cup S)^-$$

Dimostrazione. Chiaro.

Proposizione 32.17. *Per $x \in X$ si ha*

$$x^- = \neg x^+$$

Dimostrazione. Immediata.

Esercizio 32.18. Sia $P \subset X$. Allora:

$$P \in P^+$$
$$\emptyset \in P^-$$

Esercizio 32.19. Abbiamo

$$X^+ = \{X\}$$
$$X^- = \{\emptyset\}$$
$$\emptyset^+ = \mathcal{P}(X)$$
$$\emptyset^- = \mathcal{P}(X)$$

Esercizio 32.20. Sia $P \subset X$. Allora:

$$\neg P^+ = P^- \iff |P| = 1$$

Esercizio 32.21. Siano $P, R \subset X$. Allora:

$$P^+X^- = \begin{cases} \{\emptyset\} & \text{per } P = \emptyset \\ \emptyset & \text{altrimenti} \end{cases}$$

$$P^+\emptyset^- = P^+$$

$$X^+R^- = \begin{cases} \{X\} & \text{per } R = \emptyset \\ \emptyset & \text{altrimenti} \end{cases}$$

$$\emptyset^+R^- = R^-$$

Forma normale disgiuntiva

Osservazione 33.1. *Siano dati* $x_1, \ldots, x_p, y_1, \ldots, y_r \in X$. *Allora*

$$x_1^+ \cdots x_p^+ y_1^- \cdots y_r^-$$

è l'insieme di quei sottoinsiemi di X *che contengono tutti i punti* $x_1, \ldots, x_p$ *e nessuno dei punti* $y_1, \ldots, y_r$. *Inoltre*

$$\neg(x_1^+ \cdots x_p^+ y_1^- \cdots y_r^-) = x_1^- \cup \ldots \cup x_p^- \cup y_1^+ \cup \ldots \cup y_r^+$$

Dimostrazione. Il primo enunciato deriva direttamente dalla definizione 32.7, nel secondo usiamo la proposizione 32.17.

Osservazione 33.2. *Sia* $\alpha \subset \mathcal{P}(X)$. *Allora*

$$\alpha = \bigcup_{A \in \alpha} \{A\} = \bigcup_{A \in \alpha} A^+(X \setminus A)^-$$

Nota 33.3. Sia $A \subset X$. Possiamo scrivere A nella forma

$$A = \{a_1, \ldots, a_m\}$$

con gli a_j tutti distinti. Allora

$$X \setminus A = \{u_1, \ldots, u_{n-m}\}$$

con gli a_j e u_k tutti distinti perché $|X| = n$. Per l'osservazione 32.11 abbiamo allora

$$\{A\} = A^+(X \setminus A)^- = a_1^+ \cdots a_m^+ u_1^- \cdots u_{n-m}^-$$

In questa rappresentazione ogni elemento di X appare esattamente una volta: con esponente positivo, se appartiene ad A, con esponente negativo in caso contrario.

Nota 33.4. Riformulando l'osservazione 33.2 con l'aiuto della nota 33.3, vediamo che una funzione booleana $\alpha \subset \mathcal{P}(X)$ può essere rappresentata come unione (compresa l'unione vuota nel caso che $\alpha = \emptyset$) di termini della forma $a_1^+ \cdots a_m^+ u_1^- \cdots u_{n-m}^-$.

Questa rappresentazione viene chiamata la *forma normale disgiuntiva* (FND) di α. Essa consiste in pratica semplicemente di un elenco degli elementi A di α, in cui ogni volta si elencano sia gli elementi di A che quelli del complemento $X \setminus A$ e si dimostra facilmente che è unica a parte ovvie possibilità di cambi nell'ordine in cui quegli elementi vengono elencati.

Esempio 33.5. Siano $X = \{x_1, x_2, x_3\}$ un insieme con tre elementi ed $\alpha \subset \mathcal{P}(X)$ data dalla seguente tabella, in cui rappresentiamo in modo naturale l'insieme $\{x_2\}$ mediante il vettore riga 010 ecc.

x_1	x_2	x_3	α
0	0	0	1
0	0	1	0
0	1	0	1
1	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0
1	1	1	0

La forma normale disgiuntiva di α è

$$x_1^- x_2^- x_3^- \cup x_1^- x_2^+ x_3^- \cup x_1^- x_2^+ x_3^+ \cup x_1^+ x_2^- x_3^+$$

Forma normale congiuntiva

Nota 33.6. Nella forma normale disgiuntiva ogni funzione booleana è rappresentata come unione di intervalli semplici. Vogliamo adesso dimostrare che ogni funzione booleana può essere anche rappresentata come intersezione di cointervalli semplici: questa è la *forma normale congiuntiva*.

Anche stavolta la dimostrazione è molto facile e duale a quella precedente. Infatti le due forme normali sono in un certo senso *rappresentazioni tautologiche*: la forma normale disgiuntiva non è altro che un elenco degli insiemi che sono elementi della funzione booleana α data, la forma normale congiuntiva corrisponde a un elenco degli insiemi che non fanno parte di α.

Osservazione 33.7. *Sia* $\alpha \subset \mathcal{P}(X)$. *Allora*

$$\neg\alpha = \bigcup_{B \notin \alpha} \{B\}$$

e quindi

$$\alpha = \bigcap_{B \notin \alpha} \neg\{B\}$$

Nota 33.8. Sia $B \subset X$. Possiamo scrivere B nella forma

$$B = \{b_1, \ldots, b_m\}$$

con i b_j tutti distinti. Allora

$$X \setminus B = \{v_1, \ldots, v_{n-m}\}$$

con i b_j e v_k tutti distinti. Come nella nota 33.3 abbiamo allora

$$\{B\} = b_1^+ \cdots b_m^+ v_1^- \cdots v_{n-m}^-$$

cosicché, per l'osservazione 33.1,

$$\neg\{B\} = b_1^- \cup \ldots \cup b_m^- \cup v_1^+ \cup \ldots \cup v_{n-m}^+$$

Anche qui ogni elemento di X appare esattamente una volta e con esponente positivo se e solo se appartiene a B.

Nota 33.9. Riformulando l'osservazione 33.7 con l'aiuto della nota 33.8, vediamo che una funzione booleana $\alpha \subset \mathcal{P}(X)$ può essere rappresentata come intersezione (compresa l'intersezione vuota nel caso che α coincida con $\mathcal{P}(X)$) di termini della forma

$$b_1^- \cup \ldots \cup b_m^- \cup v_1^+ \cup \ldots \cup v_{n-m}^+$$

Questa rappresentazione viene chiamata la *forma normale congiuntiva* (FNC) di α. Essa non è altro che la negazione della FND di $\neg\alpha$, consiste quindi semplicemente di un elenco degli elementi di $\neg\alpha$ a cui poi si applica l'osservazione 33.1.

Esempio 33.10. Nell'esempio 33.5 la forma normale disgiuntiva di $\neg\alpha$ è

$$x_1^- x_2^- x_3^+ \cup x_1^+ x_2^- x_3^- \cup x_1^+ x_2^+ x_3^- \cup x_1^+ x_2^+ x_3^+$$

la forma normale congiuntiva di α è perciò

$$\alpha = (x_1^+ \cup x_2^+ \cup x_3^-)(x_1^- \cup x_2^+ \cup x_3^+)(x_1^- \cup x_2^- \cup x_3^+)(x_1^- \cup x_2^- \cup x_3^-)$$

Nota 33.11. Siano $\alpha \subset \mathcal{P}(X)$ ed $x \in X$. Le seguenti formule (che seguono dalla proposizione 32.17) vengono spesso usate nella semplificazone di funzioni booleane:

$$x^+ \cup x^- = \mathcal{P}(X)$$

$$x^+\alpha \cup x^-\alpha = \alpha$$

Le 16 funzioni booleane binarie

Nota 33.12. Quando $\alpha = \emptyset$, la FND di α è vuota. Nell'interpretazione funzionale $\emptyset$ corrisponde alla funzione costante 0; analogamente il caso $\alpha = \mathcal{P}(X)$ corrisponde alla funzione costante 1. Useremo queste e simili abbreviazioni nelle seguenti tabelle insieme alle formule insiemistiche.

Nota 33.13. Elenchiamo prima le 4 funzioni booleane unarie, in cui quindi $X = \{x\}$.

x	0 $\emptyset$	1 $\mathcal{P}(X)$	id x^+	NOT x^-
0	0	1	0	1
1	0	1	1	0

Vediamo che le prime due sono costanti (per ogni $n \geq 0$ esistono esattamente due funzioni costanti con i valori 0 e 1 che, come abbiamo visto, corrispondono a $\emptyset$ e $\mathcal{P}(X)$), la terza è l'identità, mentre l'unica funzione unaria non banale è la *negazione*.

Nota 33.14. Sono invece, come sappiamo, 16 le funzioni booleane binarie, in cui cioè $X = \{x, y\}$ con $x \neq y$.

x	y	0 $\emptyset$	1 $\mathcal{P}(X)$	p_x x^+	p_y y^+
0	0	0	1	0	0
0	1	0	1	0	1
1	0	0	1	1	0
1	1	0	1	1	1

Con p_x e p_y abbiamo denotato le proiezioni su x e y.

x	y	n_x x^-	n_y y^-	$=$ $x^-y^- \cup x^+y^+$	$\neq$ (XOR) $x^+y^- \cup x^-y^+$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	0

n_x e n_y sono le proiezioni negate.

x	y	$\leq$ $x^- \cup y+$	$<$ x^-y^+	$\geq$ $x^+ \cup y^-$	$>$ x^+y^-
0	0	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	1
1	1	1	0	1	0

Si osservi che nelle rappresentazioni insiemistiche stavolta abbiamo potuto usare semplificazioni che sono diverse dalle FND.

x	y	AND x^+y^+	OR $x^+ \cup y^+$	NAND $x^- \cup y^-$	NOR x^-y^-
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Abbiamo usato alcune abbreviazioni comuni della teoria dei circuiti: XOR (exclusive or), AND, OR, NAND, NOR.

Implicanti

Nota 34.1. Ogni intervallo σ di $\mathcal{P}(X)$ possiede un'unica rappresentazione della forma

$$\sigma = i_1^+ \dots i_p^+ j_1^- \dots j_r^-$$

con gli indici tutti distinti. Come si riconosce un tale intervallo nella rappresentazione grafica sull'ipercubo? Ma

$$i^+ = \{\text{punti dell'ipercubo con } i\text{-esima coordinata} = 1\}$$
$$i^- = \{\text{punti dell'ipercubo con } i\text{-esima coordinata} = 0\}$$

Ogni i^+ e ogni i^- corrisponde quindi a una faccia $(n-1)$-dimensionale di 2^n; chiedendo più di queste condizioni otteniamo tutte le facce di 2^n. Abbiamo perciò una biiezione naturale

$$\{\text{intervalli di } \mathcal{P}(X)\} \longleftrightarrow \{\text{facce dell'ipercubo } 2^n\}$$

Definizione 34.2. Sia $\alpha \subset \mathcal{P}(X)$ una funzione booleana. Un *implicante* di α è un intervallo di $\mathcal{P}(X)$ contenuto in α.

Definizione 34.3. Sia $\alpha \subset \mathcal{P}(X)$ una funzione booleana. Un implicante di α non contenuto in nessun altro implicante di α è detto un *implicante massimale* (o *primo*) di α.

Nota 34.4. Sia $\alpha \subset \mathcal{P}(X)$ una funzione booleana. Allora $A = \bigcup_{A \in \alpha} \{A\}$. Siccome ogni $\{A\} = [A, A]$ è un intervallo, vediamo che α è l'unione (vuota se $\alpha = \emptyset$) dei sui implicanti. Inoltre, essendo X finito, ogni implicante di α è contenuto in (almeno) un implicante massimale di α, quindi α è anche uguale all'unione dei suoi implicanti massimali. Denotando con $\mathrm{Max}(\alpha)$ l'insieme degli implicanti massimali di α, abbiamo perciò

$$\alpha = \bigcup_{\sigma \in \mathrm{Max}(\alpha)} \sigma$$

Se α è rappresentata sull'ipercubo, gli implicanti di α sono esattamente le facce dell'ipercubo contenute in α.

$\mathrm{Max}(\alpha)$ è l'insieme degli elementi massimali dell'insieme delle facce dell'ipercubo contenute in α.

Nota 34.5. Una delle tecniche di semplificazione di una FB α consiste nella rappresentazione di α come unione di implicanti massimali utilizzando complessivamente il minor numero possibile di condizioni (cfr. l'inizio della nota 34.1). Ciò è teoricamente sempre possibile ma molto difficile, e mentre esiste un algoritmo (di Quine/McCluskey) che permette di individuare tutti gli implicanti primi di una funzione booleana data dalla sua FND, è probabilmente intrattabile il problema di scegliere in modo efficiente quegli implicanti primi che sono necessari per la rappresentazione minimale che peraltro non è univocamente determinata.

Oltre a ciò, la rappresentazione di una funzione booleane tramite i suoi implicanti può essere eseguita in pratica solo per un numero non troppo grande di argomenti (altrimenti non siamo nemmeno in grado di compilare la tabella per la FND, da cui parte ad esempio l'algoritmo di Quine/McCluskey). Bisogna perciò spesso utilizzare altre tecniche (metodi grafici che utilizzano i *diagrammi binari di decisione* oppure i metodi algebrici potenti ma difficili che si basano sulla teoria dei *campi finiti*).

Contiamo gli intervalli di $\mathcal{P}(X)$

Nota 34.6. Consideriamo un intervallo $[P, Q]$ con $P \subset Q$ (altrimenti $[P, Q] = \emptyset$). Allora $[P, Q] \subset \mathcal{P}(Q)$, infatti

$$[P, Q] = \{A \in \mathcal{P}(Q) \mid P \subset A\}$$

Siccome ogni $A \in [P, Q]$ contiene l'insieme fissato P, ogni tale A è univocamente determinato dalla differenza $A \setminus P$, infatti

$$A = P \dot\cup (A \setminus P)$$

Abbiamo perciò una biiezione $[P, Q] \longleftrightarrow \mathcal{P}(Q \setminus P)$

in cui $A \in [P, Q]$ corrisponde ad $A \setminus P$ ed $Y \in \mathcal{P}(Q \setminus P)$ a $P \cup Y$.

Corollario 34.7. *Sia $P \subset Q$. Allora*

$$|[P, Q]| = 2^{|Q \setminus P|} = 2^{|Q| - |P|}$$

Dimostrazione. Ciò segue direttamente dalla nota 34.6.

Nota 34.8. Sia $P \subset Q$ con $P = \{x_1, \dots, x_p\}$, $X \setminus Q = \{y_1, \dots, y_r\}$. Assumiamo che gli x_j e y_k siano tutti distinti tra di loro. Sappiamo che

$$[P, Q] = P^+(X \setminus Q)^- = x_1^+ \cdots x_p^+ y_1^- \cdots y_r^-$$

Gli elementi di X che non appaiono in questa rappresentazione sono esattamente gli elementi $x_{p+1}, \dots, x_q$ (che assumiamo tutti distinti) di $Q \setminus P$. All'intervallo $[P, Q]$ corrisponde nella tabella una riga

$$\underbrace{1 \dots 1}_{x_1 \dots x_p} \ \underbrace{* \dots *}_{x_{p+1} \dots x_q} \ \underbrace{0 \dots 0}_{y_1 \dots y_r}$$

In ciascuna delle $q - p$ posizioni con * si può scegliere sia 0 che 1 con in tutto 2^{q-p} possibilità; ciò fornisce una seconda dimostrazione del corollario 34.7.

Nota 34.9.

(1) Sappiamo dal corollario 34.7 e dalla nota 34.8 che il numero degli elementi di un intervallo di 2^n è una potenza 2^d di 2; in questo caso diciamo che l'intervallo possiede la dimensione d. Come si vede dalla nota 34.8 un intervallo d-dimensionale si ottiene scegliendo $n - d$ indici e poi per ciascuno di questi $n - d$ indici j una delle funzioni j^+ o j^- (se assumiamo per semplicità che $X = n$). Ogni scelta determina univocamente un intervallo e viceversa.

(2) Esistono quindi $\binom{n}{d} 2^{n-d}$ intervalli d-dimensionali di 2^n.

(3) Il numero di tutti gli intervalli di 2^n è

$$\sum_{d=0}^{n} \binom{n}{d} 2^{n-d} = (1+2)^n = 3^n.$$

(4) I vertici sono gli intervalli 0-dimensionali; il loro numero è uguale a $\binom{n}{0} 2^n = 2^n$, correttamente.

(5) Gli spigoli sono gli intervalli 1-dimensionali; il loro numero è uguale a $\binom{n}{1} 2^{n-1} = n 2^{n-1}$.

(6) Il numero degli intervalli di dimensione $n - 1$ è uguale a $\binom{n}{n-1} 2 = 2n$.

Insiemi in Python

Le funzioni per insiemi sono contenute nel modulo `sets` di Python. In matematica si usano spesso insiemi annidati che in Python possono essere soltanto rappresentati mediante il tipo `ImmutableSet`. Illustriamo alcuni dei metodi più importanti previsti dal modulo `sets`. Per i dettagli consultare l'elenco dei moduli (*Global Module Index*) sul sito ufficiale.

`A & B`	$A \cap B$.
`A | B`	$A \cup B$.
`A - B`	$A \setminus B$.
`A<=B`	Vero, se $A \subset B$.
`A>=B`	Vero, se $A \supset B$.
`x in A, x not in A`	Vero, se $x \in A$ risp. $x \notin A$.
`len(A)`	Cardinalità di A.

```
def insiemev (x): return sets.ImmutableSet(x)

def insieme (*x): return insiemev(x)

A=insieme(3,5,8,9,4,6)
B=insieme(4,2,3,11)

C=A&B # Intersezione.
print C
# ImmutableSet([3, 4])

C=A|B # Unione.
print C
# ImmutableSet([2, 3, 4, 5, 6, 8, 9, 11])
print list(C)
# [2, 3, 4, 5, 6, 8, 9, 11]

D=insieme(A,B) # Insieme annidato.
print D
# ImmutableSet([ImmutableSet([3, 4, 5, 6, 8, 9]),
#        ImmutableSet([11, 2, 3, 4])])

print len(D)
# 2
```