PROGRAMMAZIONE

Capitoli

I. PRELIMINARI

Obiettivi del corso

Impareremo in questo corso il Python, attualmente forse il miglior linguaggio di programmazione per un uso generale: 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; per la ricerca riuscita di meccanismi di leggibilità; per la grafica con Tkinter (un'interfaccia integrata che permette di utilizzare Tcl/Tk); per la semplicità dell'aggancio ad altri linguaggi; per le crescenti librerie matematiche e scientifiche.

Siti su Internet:

www.python.org. Sito principale di Python.
docs.python.org/py3k/modindex.html. Elenco dei moduli.
pypi.python.org. Elenco dei pacchetti.
docs.python.org/py3k. Documentazione di Python 3.
felix.unife.it. Mathematical BBS. Scegliere Studenti oppure Python.
www.scipy.org. Scipy (librerie matematiche di Python).

Installazione

Le indicazioni date di seguito valgono per Windows. Le operazioni di installazione sotto Linux o sul Mac sono le solite.

Creare sul proprio desktop una cartella *Installazioni*, in cui conservare tutti i file di installazione (anche di altri programmi). Non installare direttamente online.

Prelevare *Python 3* dal sito principale di Python, scegliendo il file appropriato tra quelli offerti e salvarlo nella cartella delle installazioni. Da lì lo si può installare.

Prelevare *Scite*. Per Windows bisogna scegliere la voce *full download*. Anche Scite va prima salvato nella nostra cartella delle installazioni. Una volta installato, lasciare un'icona di Scite sul desktop e configurare Scite seguendo le istruzioni nella prossima scheda.

Per il Mac invece di Scite bisogna usare *Fraise*. Anche questo editor può essere impostato in modo che con *ALT-CMD-B* venga eseguito Python.

Questi appunti si trovano, periodicamente aggiornati, su *felix.unife.it*: seguire la voce *Studenti*.

Impostazione di Scite

Scegliere *Open Global Options File* ed effettuare le seguenti modifiche (che si ottengono in parte eliminando il simbolo `#`), adattando i parametri al proprio schermo.

```
position.width=1000
position.height=740
output.horizontal.size=500
toolbar.visible=1
title.fullpath=1
indent.automatic=0
```

Scite adesso si apre con una finestra divisa in due parti; nella parte sinistra scriviamo il nostro programma, a destra apparirà l'output del programma. I file sorgente di Python dovranno avere l'estensione `.py`.

Esecuzione da Scite con F5

Dobbiamo ancora dire a Scite di eseguire il programma ogni volta che premiamo il tasto *F5*. A questo scopo scegliamo *Open python.properties* e modifichiamo la prima delle righe che iniziano con `command.go` nel modo seguente:

```
command.go.*.py=c:/python31/pythonw -u "$(FilePath)"
```

La forma esatta di questa istruzione cambia naturalmente a seconda della versione e locazione con cui abbiamo installato Python.

Varie

Conviene creare cartelle apposite per le lezioni subito al di sotto di *c:*. Ad esempio (l'indentazione significa che si tratta di una sottocartella):

```
c:\Lezioni
  Programmazione-1011
    01-Preliminari
    02-Primi-programmi
      0201.py
      0202.py
```

Non usare nomi che contengono spazi e connettere invece parti di un nome composto con un trattino!

I file di programma (che potranno essere eseguiti con Scite) devono portare il suffisso *.py*. Copiare gli esempi dalla lezione con lo stesso numero. Si possono poi aggiungere altri esempi, raccolti in file che portano nomi come *0202-mio.py* oppure *mio-0202.py*.

Nei nomi aggiungiamo degli zeri per ottenere che i file vengono elencati nell'ordine giusto (altrimenti ad esempio *0211.py* verrebbe prima di *022.py*).

Come studiare

- ◇ Rileggere attentamente più volte gli appunti.
- ◇ Quando ci sono esempi, eseguirli con Scite.
- ◇ Dopo aver studiato gli esempi del corso, provare ad inventare da soli nuovi esempi.
- ◇ Argomenti più teorici possono essere rielaborati in un quaderno.
- ◇ Quando ci sono domande e risposte, preparare un foglio su cui scrivere (in modo abbrevviato o abbozzato) le risposte, prima di guardarle sulla scheda.
- ◇ Se si ha tempo, ripetere più volte ogni ciclo di domande e risposte.
- ◇ Prima di iniziare a studiare per l'esame, creare a mano un indice dei capitoli e degli argomenti. Alla destra di ogni voce segnare con una crocetta che si è studiato quel punto. Studiare in questo modo tutti gli argomenti di un capitolo e fare poi un ripasso veloce del capitolo. Con lo stesso metodo ripetere tutto il corso in un secondo passaggio.

R

Tra i linguaggi che si presterebbero altrettanto bene per un corso di programmazione segnaliamo qui R, equivalente dal punto di vista teorico a Python e di facile installazione.

R è un linguaggio di programmazione ad altissimo livello orientato soprattutto all'uso in statistica. In verità lo sbilanciamento verso la statistica non deriva dalla natura del linguaggio, ma dalla disponibilità di grandi raccolte di funzioni statistiche e dagli interessi dei ricercatori che lo hanno inventato e lo mantengono.

R è particolarmente popolare nella statistica medica, ma viene anche usato nella statistica economica o sociale, in geografia, nella matematica finanziaria.

L'alto livello del linguaggio permette di creare facilmente librerie di funzioni per nuove applicazioni. Sono inoltre ricchissime le capacità grafiche.

---

www.r-project.org. Sito principale di R.
felix.unife.it/++/je-v-fond0405. Corso di Informatica 2004/05.
felix.unife.it/++/je-v-asd0405. Corso di Algoritmi 2004/05.

II. PRIMI PROGRAMMI

Commenti iniziano con

Una crocetta (**#**), quando non si trova all'interno di una stringa, indica che il resto della riga (compresa la crocetta) è un *commento* e non viene considerato dall'interprete.

Fa naturalmente eccezione la sigla **#!** sotto Unix.

In questo capitolo presenteremo in modo esemplificativo alcuni semplici programmi in Python, anticipando concetti (regole di sintassi, strutture di dati, istruzioni di controllo ecc.) che verranno trattati più dettagliatamente nel seguito del corso.

Indentazione

- ◇ A differenza da molti linguaggi di programmazione il Python utilizza l'*indentazione* come elemento organizzatore.
- ◇ Più istruzioni possono trovarsi sulla stessa riga, se sono separate da un punto e virgola. Alla fine della riga il punto e virgola è facoltativo.
- ◇ Il preambolo delle istruzioni di controllo (`if`, `for`, `while`) termina con un doppio punto. Se l'istruzione da eseguire trova posto su una riga, può essere direttamente scritta alla destra del doppio punto. Altrimenti bisogna passare alla riga successiva e indentare tutto il blocco governato dall'istruzione di controllo. Usare sempre un'indentazione di 2 caratteri.
- ◇ Se una riga termina con \, il Python tratta la riga successiva come se il suo contenuto fosse scritto nella riga del \.

Fahrenheit e Celsius

Scriviamo due semplici funzioni per la conversione tra Fahrenheit e Celsius. Se f è la temperatura in gradi Fahrenheit e c la stessa temperatura espressa in gradi Celsius, allora valgono le relazioni

$c = \frac{f-32}{1.8}$ e $f = 1.8c + 32$.

```
def celsiusdafahrenheit (f): return (f-32)/1.8
def fahrenheitdacelsius (c): return 1.8*c+32

print ('Fahrenheit -> Celsius\n')

for f in (86,95,104):
  print (f,'F =',celsiusdafahrenheit(f),'C')

print ('--------------------------------')
print ('Celsius -> Fahrenheit\n')

for c in (20,30,35,40):
  print (c,'C =',fahrenheitdacelsius(c),'F')
```

con output

```
Fahrenheit -> Celsius

86 F = 30.0 C
95 F = 35.0 C
104 F = 40.0 C
--------------------------------
Celsius -> Fahrenheit

20 C = 68.0 F
30 C = 86.0 F
35 C = 95.0 F
40 C = 104.0 F
```

range

La funzione `range` restituisce una *sequenza virtuale* di interi equidistanti (i matematici dicono una *progressione aritmetica*), cioè una sequenza della forma

$$a, a+d, a+2d, \ldots, a+kd$$

con $a, d, k \in \mathbb{N}$ tale che per $d > 0$ si arrivi all'ultimo numero minore del parametro b, per $d < 0$ all'ultimo numero maggiore di b. Il valore $d = 0$ non è ammesso.

La funzione può essere usata con 1, 2 o 3 parametri nel modo seguente:

`range(n)`	$0, 1, 2, \ldots, n-1$
`range(a,b)`	$a, a+1, a+2, \ldots, b-1$ (per $a < b$)
`range(a,b,d)`	come sopra

quindi

`range(7)`	$0, 1, 2, 3, 4, 5, 6$
`range(2,7)`	$2, 3, 4, 5, 6$
`range(2,14,3)`	$2, 5, 8, 11$
`range(15,6,-2)`	$15, 13, 11, 9, 7$

Spiegheremo fra poco il significato del termine *sequenza virtuale*.

Esempi per range

`range` viene usato soprattutto nei cicli controllati da `for` oppure anche talvolta nelle liste implicite (ultimi tre esempi).

```
for i in range(6): print(i,i*i,i*i*i)
print()
# 0 0 0
# 1 1 1
# 2 4 8
# 3 9 27
# 4 16 64
# 5 25 125
```

```
for i in range(10000000):
  if i<9999996: continue
  print(i)
print()
# 9999996
# 9999997
# 9999998
# 9999999
```

```
u=[x for x in range(8)]
print(u)
# [0, 1, 2, 3, 4, 5, 6, 7]

v=[x*x for x in range(8)]
print(v)
# [0, 1, 4, 9, 16, 25, 36, 49]

w=[x*x for x in range(8) if x%2]
print(w)
# [1, 9, 25, 49]
```

Quando una lista viene costruita come negli ultimi esempi, si parla di una *lista implicita*. Usando parentesi tonde si ottiene invece un *generatore* (un tipo particolare di sequenza virtuale):

```
t=(x*x for x in range(8) if x%2)
print(t)
# <generator object <genexpr> at 0x5374b8>
```

Sequenze virtuali

In versioni precedenti di Python `range` restituiva una *lista* di numeri interi, mentre `xrange` era equivalente al `range` della versione attuale. Nei cicli si usava `xrange` perché altrimenti, ad esempio in un ciclo di un milione di passaggi (come nell'esempio della scheda precedente), veniva prima creata una lista di un milione di numeri in memoria che consumava quindi molto spazio nella RAM.

`range(1000000)` invece è in pratica un pezzettino di programma che in ogni passaggio del ciclo genera il prossimo numero di cui abbiamo bisogno per percorrere il ciclo.

Talvolta però (proprio in programmi di carattere più matematio) si vorrebbe lavorare con la lista stessa degli elementi. In questo caso si trasforma il risultato di `range` in una liste mediante `list` (oppure in una tupla mediante `tuple` oppure in un insieme mediante `set`).

```
a=range(7); b=range(2,7); c=range(2,14,3); d=range(15,6,-2)

print(a)
# range(0, 7)

print()
for u in (a,b,c,d): print(list(u))
# [0, 1, 2, 3, 4, 5, 6]
# [2, 3, 4, 5, 6]
# [2, 5, 8, 11]
# [15, 13, 11, 9, 7]
```

Assegnazioni e confronti simultanei

Sono possibili assegnazioni e confronti simultanei.

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

a=b=c=4
for x in [a,b,c]: print (x,end=' ')
# 4 4 4

print()
print (*[a,b,c])
# 4 4 4
```

Il significato del parametro `end` e dell'asterisco verrà spiegato più avanti.

Anche gli scambi possono essere effettuati in modo simultaneo:

```
a=5; b=3; (a,b)=(b,a); print ([a,b]) # [3,5]
```

Piuttosto utile è la possibilità di usare l'asterisco anche per denotare una parte di una sequenza come nei seguenti esempi:

```
(a,b,*u,c)=[3,5,8,2,10,6,4]
print(u) # [8, 2, 10, 6]

(*u,a,b,c)=[1,2,3,4,5,6,7]
print(u) # [1, 2, 3, 4]
```

Funzioni

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

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

for x in range(10): print (f(x),end=' ')
# 1 3 5 7 9 11 13 15 17 19
print()
print(*[f(x) for x in range(10)])
# 1 3 5 7 9 11 13 15 17 19

for x in range(-5,5): print (g(x),end=' ')
# 5 4 3 2 1 0 1 2 3 4
print()
print(*[g(x) for x in range(-5,5)])
# 5 4 3 2 1 0 1 2 3 4
```

Una funzione di due variabili viene definita mediante

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

similmente una funzione di tre variabili mediante

```
def f (x,y,z): ...
```

e così via.

Quando manca il `return` oppure il `return` viene usato senza un argomento, il valore restituito dalla funzione è uguale a `None`.

import math e sqrt

Le funzioni matematiche elementari (reali) fanno parte del modulo `math` che deve essere importato con l'istruzione `import math`.

Per calcolare la radice di un numero reale si usa `math.sqrt`.

```
import math

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

print (raggio(2,3))
# 3.60555127546

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

Funzioni come risultati di funzioni

In Python funzioni possono essere non solo argomenti, ma anche risultati di altre funzioni. Ciò significa che Python appartiene (come Lisp, R, Perl, Ruby, Macaulay 2) alla famiglia dei potenti *linguaggi funzionali* che in un certo senso rappresenta il massimo livello teoricamente raggiungibile da un linguaggio di programmazione.

I matematici denotano l'insieme di tutte le funzioni definite su un insieme X e con valori in un insieme Y con Y^X. In un linguaggio funzionale è possibile utilizzare gli elementi di un Y^X alla stessa stregua degli elementi di X o Y.

```
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)) # 46

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

Funzioni come argomenti di funzioni

In C, Java o Matlab funzioni possono essere solo argomenti, ma non risultati, di altre funzioni. Le costruzioni della scheda precedente non funzionano quindi in questi linguaggi, mentre è possibile imitare le seguenti istruzioni di Python.

```
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))
# 46

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

Stringhe

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

Un *carattere di nuova riga* può essere anche rappresentato da `'\n'`.

```
print ('Carlo era bravissimo.')
# Carlo era bravissimo.

print ("Carlo e' bravissimo.")
# Carlo e' bravissimo.

print ('Pero\' Giovanni e\' pigro.')
# Pero' Giovanni e' pigro.

print ('alfa\nbeta')
# alfa
# beta

print ('''Stringhe a piu' righe si
usano talvolta.''')
# Stringhe a piu' righe si
# usano talvolta.
```

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 (dal terzo in poi) vengano riuniti in una tupla `(5,7,10,8)` che nel corpo della funzione può essere usata come se questa tupla fosse `a`.

```
def verifica (x,y,*a): print (a)

verifica(2,4,5,7,10,8)
# (5, 7, 10, 8)

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

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

a=[1,2,3,10,5]
print (somma(*a))
# 21
```

Talvolta una funzione definita nel modo indicata sopra la si vorrebbe usare in una situazione in cui gli elementi sono già raccolti in una lista o tupla `a`. In tal caso è sufficiente preporre l'asterisco anche quando si invoca la funzione, come nell'ultimo esempio.

Dizionari

Dizionari (o *vettori associativi* o *tabelle di hash*) vengono definiti e usati nel modo seguente.

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

# campo     = ager
# casa      = domus
# nave      = navis
# villaggio = pagus
```

Le sigle per l'output formattato (nella quarta riga) verranno spiegate più avanti.

Nella terza riga `latino` viene usato come lista; tramite `sorted` il dizionario viene ordinato alfabeticamente secondo lo voci.

Dizionari possono essere definiti anche con un'altra sintassi tramite la funzione `dict`, come vedremo.

Uguaglianza e assegnamento

Bisogna distinguere l'operatore di assegnamento `=` dal simbolo `==` che viene usato per verificare l'uguaglianza.

Per la negazione dell'uguaglianza si usa `!=`.

```
# if a=3: print 'o.k.'
# ERRORE DI SINTASSI.

a=3 # assegnamento.
if a==3: print ('o.k.')
# o.k.

if a!=5: print ('o.k.')
# o.k.
```

Le 200 professioni più gradite negli USA

1. Mathematician
2. Actuary
3. Statistician
4. Biologist
5. Software Engineer
6. Computer Systems Analyst
7. Historian
8. Sociologist
9. Industrial Designer
10. Accountant
11. Economist
12. Philosopher
13. Physicist

...

18. Computer Programmer

...

mestieri.dima.unige.it. I mestieri dei matematici.
www.laureescientifiche.dm.unibo.it. Progetto lauree scientifiche.
www.professionematematico.dm.unibo.it. Professione matematico.

III. STRUTTURE DI DATI

Definizione di funzioni

Una funzione `f` viene definita mediante l'istruzione `def f (...):`, dove i puntini indicano l'elenco (possibilmente vuoto) dei parametri (argomenti) della funzione.

Se lo stesso nome viene usato per un'altra funzione, ciò invalida la prima definizione (come se `def` fosse un operatore di assegnamento).

Le istruzioni `def` possono essere annidate, come abbiamo già visto.

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

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

def f (x,y):
  def g (t): return t
  if x==0:
    def g (t): return t*t
  print (g(y))

f(1,4) # 4
f(0,4) # 16
```

Il λ-calcolo

In matematica bisogna distinguere tra la funzione f e i suoi valori $f(x)$. Introduciamo perciò la notazione $\bigcirc\limits_x f(x)$ per la funzione che manda x in $f(x)$. Così ad esempio $\bigcirc\limits_x \sin(x^2+1)$ è la funzione che manda x in $\sin(x^2+1)$. È chiaro che $\bigcirc\limits_x x^2 = \bigcirc\limits_y y^2$ (per la stessa ragione per cui $\sum\limits_{i=0}^{n} a_i = \sum\limits_{j=0}^{n} a_j$), mentre $\bigcirc\limits_x xy \neq \bigcirc\limits_y yy$ (così come $\sum\limits_i a_{ij} \neq \sum\limits_j a_{jj}$) e, come non ha senso l'espressione $\sum\limits_i \sum\limits_i a_{ii}$, così non ha senso $\bigcirc\limits_x \bigcirc\limits_x x$. Siccome in logica si scrive $\lambda x.f(x)$ invece di $\bigcirc\limits_x f(x)$, 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.

Liste

Liste sono successioni finite (sequenze) *modificabili* di elementi *non necessariamente dello stesso tipo*. Liste possono essere annidate. L'`i`-esimo elemento dei una lista `v` è `v[i]`, contando (come in C) cominciando da 0.

La lista che consiste solo di un singolo elemento `x` si denota con `[x]`; la lista vuota con `[]`.

La lunghezza di una lista `v` si ottiene con `len(v)`.

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

for i in range(5): print (v[i],end=' ')
print() # 1 2 5 8 7

for x in v: print (x,end=' ')
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), len([])) # 2 0
```

Alla pagina 2 abbiamo già visto che liste possono essere anche definite come liste implicite e come in questa costruzione (che in inglese è detta *list comprehension*) possono essere incluse condizioni aggiuntive mediante una clausola `if` finale:

```
u=[1,2,5,7,10,13]
v=[0,2,3,4,5,8,10]

a=[x for x in u if x in v]
print(a) # [2, 5, 10]

b=[x*x for x in u if x in v]
print(b) # [4, 25, 100]

c=[x for x in u if not x in v]
print(c) # [1, 7, 13]
```

Tuple

Tuple sono successioni finite *non modificabili* di elementi non necessariamente dello stesso tipo che sono elencati separati da virgole e possono essere facoltativamente (ed è consigliabile farlo) incluse tra parentesi *tonde*. Il fatto che le parentesi tonde siano facoltative spiega anche perché (un po' a sorpresa ed è una facile fonte d'errore) una tupla con un solo elemento deve essere scritta nella forma `(x,)` (oppure `x,` senza parentesi), perché `(x)` è la stessa cosa come `x`. Ciò non vale per le liste: `[x]` è una lista. Anche le tuple possono essere annidate. La tupla vuota è `()`.

La lunghezza di una tupla `u` si ottiene con `len(u)`; l'`i`-esimo elemento di `v` è `v[i]`.

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.

```
for x in 6,: print (x) # 6

# for x in 6: print (x)
# Causa un errore - 6 senza la virgola non e' iterabile.

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,[8,9]
print (w) # (1, 2, (3, 4, 5), 6, 7, [8, 9])

vuota=(); print (vuota) # ()

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

for i in range(0,4): print (x[i],end=' ')
print() # 3 5 8 9

for a in x: print (a,end=' ') # 3 5 8 9
```

Sequenze

Successioni finite in Python vengono dette *sequenze*; 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` sono 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`.
`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`.
`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)`	*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)`	*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`.

Funzioni specifiche per le liste

Per le liste sono disponibili alcune funzioni speciali che non possono essere utilizzate per tuple o per stringhe.

`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.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.

```
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),v.count(77)) # 5 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]
```

zip

Questa utilissima funzione corrisponde nell'idea alla formazione della trasposta di una matrice. Spieghiamo prima il contenuto matematico:

Siano date n sequenze $a_{11}, \ldots, a_{1m}, \ldots, a_{n1}, \ldots, a_{nm}$. Con esse possiamo formare lo schema rettangolare

$$\begin{array}{ccc} a_{11} & \ldots & a_{1m} \\ \vdots & \vdots & \vdots \\ a_{n1} & \ldots & a_{nm} \end{array}$$

Da esso otteniamo lo schema *trasposto*:

$$\begin{array}{ccc} a_{11} & \ldots & a_{n1} \\ \vdots & \vdots & \vdots \\ a_{1m} & \ldots & a_{nm} \end{array}$$

Un tale schema rettangolare si chiama una *matrice*. In matematica le matrici vengono normalmente racchiuse tra parentesi tonde.

Un simile passaggio a uno schema trasposto è l'effetto di `zip`. Per quanto riguarda la natura del risultato bisogna però stare un po' attenti:

Infatti `zip` restituisce un *iteratore ad esaurimento*, cioè una sequenza che non solo è virtuale, ma una volta che è stata percorsa, si esaurisce. Per trasformarla in una lista risp. una tupla si possono usare le funzioni `list` e `tuple`.

Gli elementi della sequenza virtuale restituita da `zip` sono tuple.

```
a=[11,12,13,14]
b=[21,22,23,24]
c=[31,32,33,34]

u=zip(a,b,c)
print (u) # <zip object at 0x537418>
print (list(u))
# [(11, 21, 31), (12, 22, 32), (13, 23, 33), (14, 24, 34)]

for x in u: print (x)
# Nessun output, perche' u e' esaurito!

v=list(u)
print(v) # []

u=zip(a,b,c); v=list(u)
print (v)
# [(11, 21, 31), (12, 22, 32), (13, 23, 33), (14, 24, 34)]

for x in v: print (x)
# (11, 21, 31)
# (12, 22, 32)
# (13, 23, 33)
# (14, 24, 34)

u=zip(a,b)
print (u.__next__()) # (11, 21)
print (u.__next__()) # (12, 22)
```

Non è necessario che gli argomenti di `zip` siano tutti della stessa lunghezza; se ciò non accade, il risultato ha lunghezza uguale al minimo delle lunghezze degli argomenti.

Nomi ed assegnamento

A differenza ad esempio dal C, il Python non distingue tra il nome `a` di una variabile e l'indirizzo dell'oggetto a cui la variabile si riferisce. Ciò ha implicazioni a prima vista sorprendenti nelle assegnazioni `b=a` in cui `a` è un oggetto modificabile (ad esempio una lista o un dizionario), mentre nel caso che `a` non sia modificabile (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 indirizzo e se modifichiamo l'oggetto che si trova all'indirizzo `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[:]; b.sort(); print (a)
# [0, 1, 17, 5] - a non e' cambiata.

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

def azzeraprimo (a): a[0]=0

a=[5,6]; b=a; azzeraprimo(b)
print (a) # [0, 6]
```

Copie profonde

Se `a` è una lista e gli elementi di `a` sono immutabili, allora è sufficiente, come nella scheda precedente, creare una copia con `b=a[:]` oppure `b=list(a)` affinché cambiamenti in `b` non influenzino `a`.

Ciò non basta quando anche gli elementi di `a` sono mutabili, come 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` che dobbiamo importare con `import 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 dei valori, non degli oggetti che corrispondono ai nomi `a` e `b`.

In altre parole: `a is b` è vero se e solo se `a` e `b` si riferiscono allo stesso oggetto, cosicché cambiamenti in `a` sono automaticamente anche cambiamenti in `b`.

```
import copy

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

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

a=[[1,2],[3,4]]; b=[[1,2],[3,4]]
print (b==a) # True
print (b is a) # False
print (b[0] is a[0]) # False !
print (b[0]==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
```

Sezioni di sequenze

Nella scheda 3.5 abbiamo visto che con `a[i:j]` si ottiene la sottosequenza (o *sezione*) di `a` costituita dagli elementi `a[k]` con $i \leq k < j$. Questa sintassi ammette molte variazioni e in particolare un terzo argomento che indica il passo (che può essere anche negativo): esempi nella prossima scheda.

L'ultimo elemento di una sequenza `a` è denotato con `a[-1]`. Studiare attentamente i seguenti esempi.

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

a='0123456789'
print (a[2:5]) # 234

a='oceanografia'
print (a[2:5]) # ean

a=(0,1,2,3,4,5,6,7)
print (a[2:5]) # (2, 3, 4)

print (a[-1]) # 7
print (a[4:-1]) # (4, 5, 6)
print (a[-2]) # 6
print (a[4:-2]) # (4, 5)

print (a[3:]) # (3, 4, 5, 6, 7)
print (a[:5]) # (0, 1, 2, 3, 4)

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

Sezioni possono trovarsi anche alla sinistra di un'assegnazione, ma solo in caso di liste (tuple e stringhe non sono mutabili).

```
a=[0,1,2,3,4,5,6]
a[2]=12
print (a) # [0, 1, 12, 3, 4, 5, 6]

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

a[1:3]=[21,22,23,24,25,26,27]
print (a) # [0, 21, 22, 23, 24, 25, 26, 27, 4, 5, 6]

a[2:4]=[0]
print (a) # [0, 21, 0, 24, 25, 26, 27, 4, 5, 6]

a[1:1]=[31,32,33]
print (a) # [0, 31, 32, 33, 21, 0, 24, 25, 26, 27, 4, 5, 6]
# Non e' stato eliminato nessun elemento.

a[1:2]=[41,42,43]
print (a) # [0, 41, 42, 43, 32, 33, 21, 0, 24, 25, 26, 27, 4, 5, 6]
# E' stato eliminato un elemento.
```

IV. MATEMATICA ELEMENTARE

Aritmetica

Interi possono avere un numero arbitrario di cifre.

Il quoziente intero di due interi `a` e `b` si ottiene con `a//b`, quindi ad esempio `28//11` è uguale a 2, e il resto di `a` modulo `b` è dato da `a%b`. Il quoziente reale di due numeri reali (che possono essere anche interi) `a` e `b` si ottiene con `a/b`.

Purtroppo gli operatori di divisione intera e di resto intero, come peraltro in C, non funzionano correttamente (dal punto di vista matematico) per un divisore negativo: ad esempio `40%(-6)` è `-2`, mentre in matematica si vorrebbe che il resto r modulo un divisore b soddisfi $0 \leq r < |b|$.

Le potenze x^a (ad esponenti reali) si ottengono con `x**a` oppure con `pow(x,a)`.

```
print (88/5, 88//5) # 17.6 17
print (divmod(88,5)) # (17, 3)

print (2**4,pow(2,4)) # 16 16

print (22**22) # 3414278773642195573966467235 84

print (2**16, 2**32) # 65536 4294967296

import math

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

Sintassi abbreviata per gli operatori binari: Invece di `a = a+b` si può anche scrivere `a+=b`. Abbreviazioni simili sono possibili per gli altri operatori binari. Quindi:

```
a = a+b   ....   a+=b        a = a-b   ....   a-=b
a = a*b   ....   a*=b        a = a/b   ....   a/=b
a = a//b  ....   a//=b       a = a%b   ....   a%=b
```

Per calcolare il fattoriale `n!` di un numero naturale `n` possiamo usare `math.factorial(n)`. Dopo `import math` con

```
for n in [4,5,6,7]: print('',n,'',math.factorial(n))
for n in (10,20,30,40): print(n,'',math.factorial(n))
```

otteniamo

```
 4  24
 5  120
 6  720
 7  5040
10  3628800
20  2432902008176640000
30  265252859812191058636308480000000
40  815915283247897734345611269596115894272000000000
```

Arrotondamento

Per arrotondare un numero `x` ad `n` cifre dopo il punto decimale si usa `round(x,n)`.

Per arrotondare `x` all'intero più vicino a sinistra si può usare `math.floor(x)`, per arrotondare all'intero più vicino a destra si usa `math.ceil(x)`. Per `int(x)` vedere pag. 10. Queste espressioni sono di tipo intero, a differenza da `round(x,0)`, come si vede dagli esempi:

```
a=13.345895

u=int(a); print (pow(u,40))
# 36118864808453144592992087764134015654 4317

u=math.floor(a); print (pow(u,40))
# 36118864808453144592992087764134015654 4317

u=round(a,0); print (pow(u,40))
# 3.61188648085e+44

print (math.floor(12.3)) # 12
print (math.floor(12.99)) # 12
print (math.ceil(12.3)) # 13
print (math.ceil(12.99)) # 13

print (int(12.3)) # 12
print (int(12.99)) # 12

print (round(12.3,0)) # 12.0
print (round(12.5,0)) # 12.0
print (round(12.99,0)) # 13.0
```

Numeri razionali

Il modulo `fractions` permette di calcolare con numeri razionali.

Il modulo contiene anche la funzione `fractions.gcd` che calcola il massimo comune divisore (in inglese *greatest common divisor*) di due numeri interi.

```
import fractions

numraz=fractions.Fraction

print (numraz(77,12))
# 77/12

print (numraz(77,11))
# 7

print (numraz(3,8)+numraz(6,7)+1)
# 125/56

a=numraz(16,9)
print (pow(a,3))
# 4096/729

d=fractions.gcd(4692,1648)
print(d) # 4
```

Funzioni iperboliche

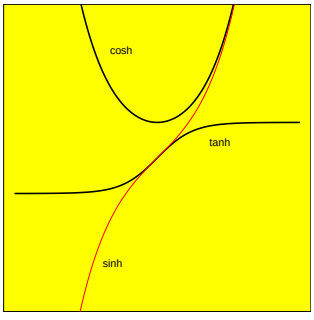
Le funzioni iperboliche hanno importanti applicazioni tecniche; la tangente iperbolica $\tanh$ è anche nota sotto il nome di *funzione sigmoidea* e viene spesso utilizzata per modellare la crescita di popolazioni (*crescita logistica*) o la formazione di impulsi, ad esempio nelle *reti neuronali*.

Le funzioni iperboliche sono così definite:

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

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

Le loro inverse portano il prefisso *ar* (non *arc*) che sta per *area*; quindi arcosh si legge *area coseno iperbolico*.



Il modulo math

Il modulo `math` contiene le principali funzioni mathematiche elementari per numeri reali. L'elenco che segue ne contiene le più importanti, tralasciando il prefisso `math.` (quindi invece di `cos` bisogna scrivere `math.cos`, invece di `e` bisogna scrivere `math.e` ecc.) con a destra il significato matematico.

`cos(x)`	$\cos x$
`sin(x)`	$\sin x$
`tan(x)`	$\tan x$
`acos(x)`	$\arccos x$
`asin(x)`	$\arcsin x$
`atan(x)`	$\arctan x$
`atan2(y,x)`	$\arctan y/x$
`cosh(x)`	$\cosh x$
`sinh(x)`	$\sinh x$
`tanh(x)`	$\tanh x$
`acosh(x)`	$\operatorname{arcosh} x$
`asinh(x)`	$\operatorname{arsinh} x$
`atanh(x)`	$\operatorname{artanh} x$
`exp(x)`	e^x
`log(x)`	$\log x$
`log(x,b)`	$\log_b x$
`log10(x)`	$\log_{10} x$
`sqrt(x)`	$\sqrt{x}$
`fabs(x)`	$\lvert x\rvert$
`floor`	$[x]$ (intero più vicino a sinistra)
`ceil`	intero più vicino a destra
`e`, `pi`	e, π (costanti)
`factorial(n)`	$n!$
`fsum(a)`	somma degli elementi di a

$\arccos$, $\arcsin$ e $\arctan$ sono le funzioni inverse delle funzioni trigonometriche $\cos$, $\sin$ e $\tan$. Il prefisso *arc* sta per *arco*.

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

Numeri complessi

Un numero complesso è una coppia $z = (x, y)$ di numeri reali che viene anche scritta nella forma $z = x + iy = x + yi$. L'insieme $\mathbb{C}$ dei numeri complessi coincide quindi con il piano reale $\mathbb{R}^2$.

L'addizione di numeri complessi è l'addizione vettoriale nel piano, quindi $(x + yi) + (u + vi) = (x + y) + (u + v)i$, la molteplicazione viene eseguita applicando la regola $i^2 = -1$, per cui $(x+yi)(u+vi) = (xu - yv) + (xv + yu)i$. In questo modo $(\mathbb{C}, +, \cdot)$ diventa un campo (o corpo) nel senso dell'algebra.

Sia $z = x+yi$ con $x, y \in \mathbb{R}$. Allora $\overline{z} := x-yi$ si chiama il conjugato complesso di z. Il valore assoluto di z (la sua lunghezza come vettore del piano) è uguale a $|z| = \sqrt{x^2 + y^2} = \sqrt{z\overline{z}}$.

Dalla rappresentazione polare di un punto nel piano abbiamo infine una rappresentazione $z = |z|e^{i\alpha}$ con l'angolo α univocamente determinato modulo 2π tranne che per $z = 0$.

Per $z \neq 0$ si ha $1/z = \overline{z}/|z|^2$ e in particolare $1/i = -i$.

La funzione esponenziale $\underset{z}{\bigcirc}\, e^z : \mathbb{C} \longrightarrow \mathbb{C}$ è definita in modo che

$$e^{x+yi} = e^x(\cos y + i \sin y)$$

Questa formula vale anche se x ed y sono numeri complessi qualsiasi; il caso più importante, da cui quello generale deriva, è

$$e^{i\alpha} = \cos\alpha + i \sin\alpha$$

per $\alpha \in \mathbb{R}$. Si noti in particolare che i numeri della forma $e^{i\alpha}$ con $\alpha \in \mathbb{R}$ sono esattamente i punti della circonferenza unitaria!

Le funzioni trigonometriche complesse

Esiste un intimo e a prima vista sorprendente legame tra le funzioni trigonometriche, le funzioni iperboliche e la funzione esponenziale, se le vediamo nell'ambito dei numeri complessi. Sia z un numero complesso. Dalle relazioni

$$e^{iz} = \cos z + i \sin z \qquad e^{-iz} = \cos z - i \sin z$$

troviamo

$$\cos z = \frac{e^{iz} + e^{-iz}}{2} \qquad \sin z = \frac{e^{iz} - e^{-iz}}{2i}$$

e quindi un'espressione di seno e coseno in termini della funzione esponenziale.

Se confrontiamo queste formule con le definizioni del seno e del coseno iperbolici nella scheda 4.7, vediamo inoltre che

$$\begin{array}{lll} \cos z = \cosh iz & \sin z = -i \sinh iz & \tan z = -i \tanh iz \\ \cosh z = \cos iz & \sinh z = -i \sin iz & \tanh z = -i \tan iz \end{array}$$

Esercizio: Eseguire i conti.

Vediamo così che la funzione esponenziale, prototipo di una funzione a velocissima crescita, è invece limitata se considerata sulla retta immaginaria $i\mathbb{R}$, mentre seno e coseno, prototipi di funzioni limitate, sono limitate solo sulla retta reale, mentre sulla retta immaginaria crescono come l'esponenziale reale.

Il modulo cmath

In Python il numero complesso $4 + 5i$ viene rappresentato nella forma `4+5j`. Per i bisogna scrivere `1j` e se `b` è una variabile, naturalmente `b*1j` e non `bj`. La parte reale e la parte immaginaria di `z` si ottengono con `z.real` e `z.imag`, il coniugato complesso con `z.conjugate()` (non dimenticare le parentesi), il valore assoluto con `abs(z)`.

Il modulo `cmath` contiene le funzioni trigonometriche, iperboliche ed esponenziali (stavolta con il prefisso `cmath.`) per i numeri complessi, e in più alcune funzioni ausiliarie che sono così definite:

`cmath.phase(z)`	...	`math.atan2(z.imag,z.real)`
`cmath.polar(z)`	...	`(abs(z),phase(z))`
`cmath.rect(r,alfa)`	...	`r*cmath.exp(alfa*1j)`

```
import cmath

z=4+3j

print (abs(z)) # 5.0
print (cmath.phase(z)) # 0.643501108793

print (cmath.polar(z)) # (5.0, 0.6435011087932844)

print (cmath.cosh(z)) # (-27.0349456031+3.85115333481j)

print (cmath.cos(1j*z)) # (-27.0349456031+3.85115333481j)
```

Gli operatori aritmetici possono essere applicati direttamente in espressioni che contengono numeri complessi:

```
z=4+5j; w=6-1j
print (z+w, z-w, z*w)
print (z/w)
print (pow(z,3))

# (10+4j) (-2+6j) (29+26j)
# (0.513513513514+0.918918918919j)
# (-236+115j)
```

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); cfr. pag. 8 per l'uso `round`, `math.floor` e `math.ceil`.

Se `a` è una stringa, si può indicare un numero intero `b` compreso tra 2 e 36 come secondo argomento e allora `int(a,base=b)` è l'intero rappresentato in base `b` dalle cifre che compongono la stringa `a`; la preimpostazione è `base=10`.

Per `b`> 10 le lettere vengono usate come nella rappresentazione esadecimale, quindi $a = 10, \ldots, f = 15, g = 16, \ldots, z = 35$. Se la stringa inizia con `-`, viene calcolato il corrispondente intero negativo.

```
print (*[int(x) for x in (3.5,4,6.9,-3.7,-14.1)])
# 3 4 6 -3 -14

print (int('-345',base=10))
# -345
print (int('-345',base=8))
# -229 # Perche' 3*64+4*8+5=229

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

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

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

`hex` fornisce la rappresentazione esadecimale di un numero intero nel formato utilizzato da Python:

```
print (*[hex(x) for x in (15,92,187,-64, -325)])
# 0xf 0x5c 0xbb -0x40 -0x145
```

`bin` restituisce la rappresentazione 2-adica (o binaria) di un intero:

```
print (bin(13))
# 0b1101

print (*map(bin,[15,6,88,14]))
# 0b1111 0b110 0b1011000 0b1110

print (*['%d = %s' %(x,bin(x)) for x in [15,6,88]], sep=', ')
# 15 = 0b1111, 6 = 0b110, 88 = 0b1011000
```

Nonostante l'ultimo esempio, `0b1101` non è una stringa, ma un'altra rappresentazione di 13:

```
print (int('1101',2))
# 13

print (0b1101)
# 13

print (0b1101+14)
# 27

print (0b1101+14+0xf)
# 42

print ('0b1101')
# 0b1101
```

La funzione `str` converte oggetti di Python in stringhe e può essere utilizzata per trasformare un numero in una stringa.

```
print (str(120)) # 120
print (str(0x23)) # 35 (come stringa)
print (str(0b1111)) # 15 (come stringa)
```

Insiemi

Se `u` è una sequenza o una sequenza virtuale, allora l'istruzione `A=set(u)` definisce un insieme i cui elementi sono gli oggetti contenuti in `u`. Gli insiemi così ottenuti sono modificabili e non possono essere elementi di un altro insieme.

Con `B=frozenset(u)` otteniamo invece un insieme non modificabile che può essere elemento di un altro insieme.

Operatori definiti sia per insiemi modificabili che per insiemi non modificabili:

`A == B`	vero se $A = B$
`A != B`	vero se $A \neq B$
`x in A`	vero se $x \in A$
`x not in A`	vero se $x \notin A$
`A | B`	$A \cup B$
`A & B`	$A \cap B$
`A - B`	$A \setminus B$
`A <= B`	vero se $A \subset B$
`A >= B`	vero se $A \supset B$
`max(A)`	elemento più grande di A
`min(A)`	elemento più piccolo di A
`len(A)`	$\lvert A \rvert$

Operatori definiti solo per insiemi modificabili:

`A.add(x)`	$A = A \cup \{x\}$
`A.clear()`	$A = \emptyset$
`A.remove(x)`	$A = A \setminus \{x\}$ (errore se $x \notin A$)

Invece che con `set` insiemi possono essere anche rappresentati tramite parentesi graffe come nella notazione matematica (e da non confondere con l'uso delle parentesi graffe per i dizionari):

```
A=set([1,2,5,9])
B={1,2,9,5}
if A==B: print('Sono uguali.')
# Sono uguali.
```

Anche per gli insiemi esiste la rappresentazione implicita, analoga a quella per le liste (cfr. pag. 5):

```
C={x*x*x for x in range(4)}
print (C)
# 0, 1, 27, 8}

D={x-1 for x in C if x>1}
print (D)
# {26, 7}
```

Le orbite di un sistema dinamico finito

X sia un insieme ed $f : X \longrightarrow X$ un'applicazione. Allora la coppia (X, f) si chiama un *sistema dinamico*. Per ogni $x \in X$ definiamo l'*orbita* di x rispetto ad f come

$$o(x, f) := \{f^n(x) \mid n \in \mathbb{N}\} = \{x, f(x), f(f(x)), \ldots\}$$

Se X è finito, naturalmente anche ogni orbita è finita.

Per calcolare l'orbita, possiamo usare la seguente funzione (il `while` verrà discusso nel prossimo capitolo):

```
def orbita (x,f):
  A=set([x])
  while 1:
    x=f(x)
    if x in A: return A
    A.add(x)
```

Calcolare le orbite del sistema dinamico $(\{0, \ldots, 10\}, \bigcirc_x (3x+7)\%11)$ e controllare il risultato con un piccolo programma in Python.

V. ISTRUZIONI DI CONTROLLO

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,end=' ')
```

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, mentre numeri diversi da zero, liste, tuple e stringhe non vuote hanno 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,end=' ')
# True False
```

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
```

L'operatore ternario di decisione

In Python l'operatore ternario di decisione (o di diramazione) viene usato nella forma `a if C else b`. Questa espressione ha il valore `a` se la condizione `C` è soddisfatta, altrimenti è uguale a `b`. `a` viene calcolato solo nel primo caso, `b` solo nel secondo.

L'operatore di diramazione è molto importante nell'informatica teorica, ad esempio nella teoria dei *diagrammi binari di decisione* nello studio delle funzioni booleane e nelle trattazioni teoriche è spesso scritto nella forma $[C, a, b]$. In C appare nella forma `C ? a : b`. Esempi:

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

print (fatt(7))
# 5040

def segno (x): return 1 if x>0 else 0 if not x else -1

print (*[segno(x) for x in (2,0,-3)])
# 1 0 -1

def numbin (n,k):
  return 1 if k==0 else 0 if n<k else (n*numbin(n-1,k-1))//k

for i in range(7): print (numbin(6,i))
```

Il numero binomiale $\binom{n}{k}$ è definito come il numero dei sottoinsiemi a k elementi di un insieme ad n elementi e può essere scritto nella forma $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n-1}{k-1}\frac{n}{k}$ per $0 \le k \le n$, mentre è ovviamente uguale a 0 se $k > n$.

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: s=1
  elif x==0: s=0
  else: s=-1
  return s
```

Mancanza dello switch

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.

---

for

La sintassi di base del `for` ha la forma

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

dove `v` è una sequenza o una sequenza virtuale.

Dal `for` si esce con `break` (o naturalmente, da una funzione, con `return`), mentre `continue` interrompe come in C il passaggio corrente di un ciclo e porta all'immediata esecuzione del passaggio successivo, cosicché

```
for x in range(0,21):
  if x%2>0: continue
  print (x,end=' ')
```

stampa sullo schermo i numeri pari compresi tra 0 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:

```
print()
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 range(4):
  for j in range(5):
    print (i+j,end=' ')
  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 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,end=' ')
print()
# 11 12 13 24

v=['Aa','Bb','Cc','Dd','Ee']
for x,y in v: print (y+'.'+x,end=' ')
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,end=' ')
print()
# 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,end=' ')
print()
# 0 13 2 10 15 16 6 4 7 11 5 14 9 8 1 3
```

pass

`pass` è un'istruzione che non effettua alcuna operazione e viene usata quando la sintassi richiede un'istruzione, senza che debba essere eseguita un'azione.

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(range(len(v)),v)`:

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

e=enumerate(v)
print (e)
# <enumerate object at 0x5373f0>

a=list(enumerate(v))
b=list(zip(range(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.
```

Somme trigonometriche I

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 Ruffini 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 $\sum_{n=0}^{N} a_n \varphi(n, x)$.

any ed all

Se `v` è una sequenza, `all(v)` è vero se e solo se ogni elemento di `v` è vero, mentre `any(v)` è vero se e solo se almeno uno degli elementi di `v` è vero.

Se `v` è una lista implicita, si può tralasciare una coppia di parentesi.

```
print (all(range(5))) # False
print (any(range(5))) # True

print (all(x<0 for x in [2,3,-5])) # False
print (any(x<0 for x in [2,3,-5])) # True

print (all(x%2==0 for x in [2,0,4,8])) # True
print (any(x%2==1 for x in [3,4,8,10])) # True
print (any(x%2==1 for x in [4,8,10])) # False
```

try ... except

Non sempre è possibile prevedere se un'istruzione può effettivamente essere eseguita correttamente. Ciò si verifica ad esempio se dovrebbe essere aperto un file che potrebbe anche non esistere o non accessibile, o per qualche altra ragione.

Per questi casi Python prevede la costruzione `try ... except`, simile ad un `if ... else`, in cui la prima parte contiene le istruzioni che vorremmo eseguire nel caso che tutto funzioni, la seconda parte le istruzioni da effettuare nel caso che si verifichi un errore.

```
try: istruzioni per il caso che tutto vada bene
except: istruzioni per il caso che il try non funzioni
```

Nella seconda parte quindi possiamo prevedere messaggi d'errore, oppure una soluzione alternativa, oppure che non venga effettuata nessuna operazione. In quest'ultimo caso non si può semplicemente lasciar via la parte `except`, ma bisogna scrivere

```
try: istruzioni
except: pass
```

VI. FUNZIONI

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 nome, questi ultimi vengono identificati per la loro posizione; ciò implica che argomenti senza nome 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 g (x,y=3,z=4): return x+2*x*y+3*x*z

print (g(10,z=1)) # 100

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

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

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 nell'articolo *Una trappola pericolosa*. Invece

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

f(1) # 4

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

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. Esempi:

```
x=7

def f(): x=2

f(); print (x) # 7

def g(): global x; x=2

g(); print (x) # 2

def aumentachenonfunziona (u): u=u+1

u=7
aumentachenonfunziona(u)
print (u) # 7
```

Anche 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]
```

Si dovrebbe cercare di utilizzare variabili globali solo quando ciò si rivela veramente necessario. In tal caso conviene creare un elenco di tutte le variabili globali in un apposito file del progetto.

nonlocal

La dichiarazione `nonlocal` è simile a `global`, alza però di un solo livello la visibilità di una variabile. *Questa deve comunque già esistere a livello superiore*. Viene usata talvolta all'interno di funzioni annidate, ma non è molto importante:

```
x=10

def f (t):
  x=None
  def h ():
    nonlocal x; x=100
  h(); return x*t

print (f(2), x) # 200 10
```

Una trappola pericolosa

Abbiamo già visto a pagina 7 che i nomi del Python devono essere considerati come nomi di puntatori 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 indirizzo. Ciò vale 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), come abbiamo visto nell'articolo *Valori globali*. Mentre, se il programmatore ha davanti agli occhi la memoria su cui sta lavorando, ciò è piuttosto evidente, 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]

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

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

Espressioni lambda

L'espressione `lambda x: f(x)` corrisponde all'espressione matematica $\bigcirc_x f(x)$ (cfr. pag. 5). Con due variabili diventa `lambda x,y: f(x,y)`. `f(x)` deve essere un'unica espressione e soprattutto non può contenere istruzioni di assegnamento e di controllo; manca anche il `return`.

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

def u (x): return x**2
def v (x): return 4*x+1

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

Qui abbiamo ridefinito la funzione `comp` vista a pag. 3.

Un'espressione lambda è quindi essenzialmente una funzione senza nome; possiamo comunque assegnarle un nome a posteriori:

```
f=lambda x,y: x+y-1
print (f(6,2)) # 7
```

Le espressioni lambda possono essere annidate:

```
f=lambda a: lambda x: a+x
print (f(5)(8)) # 13
```

La `f` in questo esempio matematicamente corrisponde a $f = \bigcirc_a \bigcirc_x a+x$.

Un'espressione lambda senza argomenti è una funzione costante.

Il modulo inspect

Questo modulo permette di ottenere informazioni su funzioni (in verità anche su classi e moduli) talvolta molto utili nella programmazione avanzata. Elenchiamo solo alcune delle molte possibilità; per maggiori dettagli consultare la voce *inspect* nell'elenco dei moduli di Python (cfr. pag. 1).

È una buona abitudine, quando si crea una biblioteca di funzioni, aggiungere nelle righe immediatamente precedenti il testo sorgente della funzione alcuni brevi commenti o istruzioni sull'uso. A questa abitudine viene incontro la funzione `inspect.getcomments` che permette di ricavare queste righe di commento:

```
import inspect,math

# Lunghezza di un vettore v=(x,y,z)
# nello spazio 3-dimensionale.
def lunghezza (v):
  (x,y,z)=v; return math.sqrt(x*x+y*y+z*z)

print (inspect.getcomments(lunghezza))
# # Lunghezza di un vettore v=(x,y,z)
# # nello spazio 3-dimensionale.
```

Con `inspect.getfile(f)` otteniamo invece il nome del file in cui la funzione `f` è definita. Provare!

Per ottenere il testo sorgente completo della funzione `f` possiamo usare `inspect.getsource(f)`.

Applichiamo questa istruzione alla nostra funzione `lunghezza`:

```
print (inspect.getsource(lunghezza))
# def lunghezza (v):
#   (x,y,z)=v; return math.sqrt(x*x+y*y+z*z)
```

In questo modo, combinando `inspect.getsource` con `exec` o `eval`, possiamo facilmente creare programmi che si automodificano - una capacità che in qualche modo li rende simili ad esseri viventi (se assumiamo ad esempio che mediante funzioni di input/output riescono a interagire con l'ambiente e ad imparare)!

Tramite `inspect.getfullargspec(f)` otteniamo una tupla con nomi (un tipo di dati che non trattiamo, simile a un dizionario) che restituisce informazioni sugli argomenti di una funzione. Esaminare l'esempio per comprendere l'uso:

```
def f (x,y=2,z=7,*a,**u): pass

argo=inspect.getfullargspec(f)

print (argo.args)
# ['x', 'y', 'z']

print (argo.varargs)
# a

print (argo.varkw)
# u

print (argo.defaults)
# (2, 7)
```

`kw` in `varkw` significa *keywords* e si riferisce ad argomenti rappresentati da un dizionario che tratteremo più avanti.

eval

Se `alfa` è una stringa che contiene un'*espressione* valida di Python (ma non ad esempio un'assegnazione), allora `eval(alfa)` 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
```

exec

Se la stringa `alfa` contiene istruzioni di Python valide, queste vengono eseguite con `exec(alfa)`:

```
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*.

Attenzione: Quando un comando `exec` viene eseguito all'interno di una funzione, per fare in modo che le variabili che appaiono nell'argomento di `exec` vengano trattate come variabili globali, bisogna aggiungere `globals()` come secondo argomento. Esempio:

```
x=0; comando='x=17'

def g (): exec(comando,globals())

g(); print (x) # 17
```

Cosa succede senza il parametro `globals()`?

VII. GENERATORI DI SEQUENZE

Iteratori

Python 3 prevede vari tipi di sequenze virtuali, la cui precisa definizione per noi non è così importante. Ne discutiamo solo gli iteratori e i generatori.

Iteratori sono oggetti che forniscono uno dopo l'altro tutti gli elementi di una sequenza senza creare questa sequenza in memoria. In memoria si trova invece un piccolo pezzo di codice che effettua ogni volta l'operazione richiesta (cfr. quanto detto a pag. 3 nell'articolo sulle sequenze virtuali).

Gli iteratori possono essere ad esaurimento come il risultato di uno `zip` (lo stesso vale per `map` e `filter`). Questo meccanismo è molto scomodo e costituisce una facile fonte d'errore.

La funzione `list` può essere applicata agli iteratori, come abbiamo già visto in alcuni esempi, 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))`.

```
a='abcdefgh'
b=reversed(a)
print (b)
# <reversed object at 0x534870>
print (list(b))
# ['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
print (''.join(list(b)))
# Nessun risultato, perche' l'iteratore b e' stato esaurito!
c=reversed(a)
print (''.join(list(c)))
# hgfedcba
```

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 alle liste implicite (viste alle pagine 2 e 5): dobbiamo soltanto sostituire le parentesi quadre con parentesi tonde. Il prossimo elemento generato da un generatore `a` si ottiene con `next(a)`.

```
a=[n*n for n in range(8)]
print (a)
# [0, 1, 4, 9, 16, 25, 36, 49]

a=(n*n for n in range(8))
print (a)
# <generator object at 0x596e4c>

for k in range(4): print (next(a),end=' ')
# 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 la funzione `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 quadrati:

```
def quadrati ():
  n=0
  while 1: yield n*n; n+=1

q=quadrati()
for k in range(8): print (next(q),end=' ')
# 0 1 4 9 16 25 36 49

print()
for x in quadrati():
  if x>100: break
  print (x,end=' ')
# 0 1 4 9 16 25 36 49 64 81 100
```

Nel penultimo esempio abbiamo generato addirittura *una successione infinita!* Infatti ogni chiamata `next(q)` 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 range(8):
  print (next(fattoriali),end=' ')
# 1 2 6 24 120 720 5040 40320
```

Spesso però si vorrebbe una successione finita, simile a un `range`, da usare in un ciclo `for`. Allora possiamo modificare l'esempio nel modo seguente:

```
def quadrati (n):
  for k in range(n): yield k*k

print()
q=quadrati(8)
for x in q: print (x,end=' ')
# 0 1 4 9 16 25 36 49
```

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=next(numeri); 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)) # []
```

Anche nella definizione implicita di generatori mediante un'espressione tra parentesi tonde si può aggiungere una clausola `if` finale:

```
a=(x*x for x in range(10) if x%2)

for x in a: print(x,end=' ')
# 1 9 25 49 81
```

Il map semplice

Il `map` è il più classico degli operatori che generano sequenze. Nonostante ciò il `map` semplice in Python è superfluo, perché può essere sostituito dall'uso di liste implicite. Infatti l'espressione `map(f,a)` è una sequenza virtuale che definisce gli stessi elementi di `[f(x) for x in a]`.

```
def f (x): return x*x*x

a=list(map(f,range(8)))
b=[x*x*x for x in range(8)]
print (a==b)
# True
```

filter

L'espressione `filter(g,a)` è una sequenza virtuale che definisce gli stessi elementi di `[x for x in a if g(x)]`. Quindi anche l'operatore `filter` in Python è superfluo.

```
def pari (x): return not x%2

a=list(filter(pari,range(8)))
b=[x for x in range(8) if pari(x)]
print (a==b)
# True
```

Come già osservato sia `map` che `filter` restituiscono sequenze virtuali ad esaurimento.

Filtri (definiti tramite `filter` o mediante una lista implicita con clausola `if` finale) si usano ad esempio nella gestione di banche di dati; possiamo 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 = [x for x in imp if x[1]>=3000]
for x in imp3000: print (x[0],end=' ')
# Verdi Bianchi
```

Il map multivariato

Siano $a^1 = (a^1_0, a^1_1, \ldots)$, …, $a^m = (a^m_0, a^m_1, \ldots)$ sequenze (o sequenze virtuali) ed f una funzione di m variabili.

Allora $\mathrm{map}(f, a^1, \ldots, a^m)$ è una sequenza virtuale che corrisponde alla sequenza $(f(a^1_0, \ldots, a^m_0), f(a^1_1, \ldots, a^m_1), \ldots)$.

Nel caso $m = 1$ ritroviamo il `map` semplice, nel caso $m = 2$, date due sequenze $a = (a_0, a_1, \ldots)$ e $b = (b_0, b_1, \ldots)$ e una funzione f di due variabili, l'espressione $\mathrm{map}(f, a, b)$ definisce (virtualmente) la sequenza $(f(a_0, b_0), f(a_1, b_1), \ldots)$.

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

print()
u=map(somma,(2,5,3,8),(3,2,1,4),(9,1,5,6))
print(u)
# <map object at 0x538ad0>
print(list(u))
# [14, 8, 9, 18]
# Infatti 2+3+9=14, 5+2+1=8, 3+1+5=9, 8+4+6=18
```

Per $m \geq 2$ il `map` non può essere sostituito del tutto dall'uso di liste implicite. Solo aiutandoci con `zip` riusciamo ad imitare l'ultimo esempio con una lista implicita:

```
u=[somma(x,y,z) for x,y,z in zip((2,5,3,8),(3,2,1,4),(9,1,5,6))]
print(u)
# [14, 8, 9, 18]
```

Anche dal punto di vista teorico l'uso di `map` è preferibile (per costruzioni avanzate) a quello talvolta più intuitivo delle liste implicite.

Il risultato di `f` può naturalmente essere anche una sequenza:

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

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

u=map(f,a,b); print (list(u))
# [(3, 1), (6, 0), (13, -3), (0, 0), (6, -2)]
```

Cosa succede se la funzione `f` in un `map` è l'identità? È facile convincersi che ciò riproduce lo `zip`:

```
def id (*a): return a

u=map(id,(1,2,3))
print (list(u))
# [(1,), (2,), (3,)]

u=map(id,(11,12,13),(21,22,23))
print (list(u))
# [(11, 21), (12, 22), (13, 23)]
```

In versioni precedenti di Python qui invece di `id` si poteva addirittura un po' abusivamente utilizzare `None`.

functools.reduce

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

$$\begin{aligned}
\mathrm{functools.reduce}(f, [a_1]) &= a_1 \\
\mathrm{functools.reduce}(f, [a_1, a_2]) &= a_1 \cdot a_2 \\
\mathrm{functools.reduce}(f, [a_1, a_2, a_3]) &= (a_1 \cdot a_2) \cdot a_3 \\
\mathrm{functools.reduce}(f, [a_1, a_2, a_3, a_4]) &= ((a_1 \cdot a_2) \cdot a_3) \cdot a_4 \\
&\ldots
\end{aligned}$$

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

$$\begin{aligned}
&\mathrm{functools.reduce}(f, [a_1, \ldots, a_n], a_0) \\
&\qquad := \mathrm{functools.reduce}(f, [a_0, \ldots, a_n])
\end{aligned}$$

Nella *genetica algebrica* (che esprime le leggi di Mendel) si usa la composizione non associativa

$$a \cdot b := \frac{a+b}{2}$$

```
import functools

reduce=functools.reduce

def f (x,y): return (x+y)/2

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

Se per numeri reali $x, y > 0$ definiamo

$$x \cdot y := \frac{1}{x} + y$$

otteniamo le *frazioni continue*, con i coefficienti elencati all'indietro:

```
def frazcont (a):
  def f (x,y): return 1/x+y
  return reduce(f,reversed(a))

s=frazcont([2,3,1,5])
print(s)
# 2.26086956522
print (52/23)
# 2.26086956522
```

VIII. ALGORITMI ELEMENTARI

Il crivello di Eratostene

Definizione 18.1. Siano $a, b \in \mathbb{Z}$. Diciamo che a *divide* b e scriviamo $a|b$, se b è un multiplo di a, cioè se esiste $k \in \mathbb{Z}$ tale che $b = ka$, ovvero se e solo se $b \in \mathbb{Z}a$.

Osservazione 18.2. È chiaro che $a|b \iff \mathbb{Z}b \subset \mathbb{Z}a$.

Questa riduzione della relazione di divisibilità a una relazione insiemistica è molto importante e conduce alla *teoria degli ideali*, un ramo centrale dell'algebra commutativa.

Definizione 18.3. Un numero naturale p si chiama *primo*, se $p \geq 2$ e se per $d \in \mathbb{N}$ con $d|p$ si ha $d \in \{1, p\}$.

Osservazione 18.4. $n \in \mathbb{N} + 2$ sia un numero naturale ≥ 2 fissato. m sia un altro numero naturale con $2 \leq m \leq n$ che non sia primo. Allora esiste $a \in \mathbb{N}$ con $2 \leq a \leq \sqrt{n}$ tale che $a|m$.

In altre parole, se sappiamo che $m \leq n$, per verificare che m sia primo, dobbiamo solo dimostrare che m non possiede divisori tra 2 e $\sqrt{n}$.

Dimostrazione. Per ipotesi esistono $a, b \in \mathbb{N}$ tali che $ab = n$ con $a, b \geq 2$. Se si avesse $a > \sqrt{n}$ e $b > \sqrt{n}$, allora $ab > n$, una contraddizione.

Dal lemma 18.6 e tenendo conto dell'oss. 18.4 otteniamo un antico metodo per trovare i numeri primi $\leq n$, noto come *crivello di Eratostene*, che può essere facilmente programmato in Python:

```
def eratostene (n):
  v=range(2,n+1); u=[]; r=math.sqrt(n); p=2
  while p<=r: u.append(p); v=[x for x in v if x%p]; p=v[0]
  return u+v
```

Per stampare i primi ≤ 200 usiamo le seguenti istruzioni:

```
for i,p in enumerate(eratostene(200)):
  if i%16==0: print()
  print(p,end=' ')
```

```
# 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53
# 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131
# 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Esistono infiniti numeri primi

Teorema 18.5. *Ogni insieme non vuoto di numeri naturali possiede un elemento più piccolo.*

Più esplicitamente: Sia $A \subset \mathbb{N}$ con $A \neq \emptyset$. Allora esiste un $x \in A$ con $x \leq a$ per ogni $a \in A$.

Dimostrazione. Sia $A \subset \mathbb{N}$ ed $A \neq \emptyset$. Assumiamo, per assurdo, che A non abbia un elemento più piccolo. Dimostriamo che $A = \emptyset$ (una contraddizione all'ipotesi), dimostrando per induzione che $n \notin A$ per ogni $n \in \mathbb{N}$.

$n = 0$: Se fosse $0 \in A$, allora 0 sarebbe l'elemento più piccolo di A.

$n \longrightarrow n + 1$: Sia $n+1 \in A$. Per l'ipotesi di induzione avremmo $k \notin A$ per ogni $k \leq n$. Ciò implicherebbe però che $n + 1$ è l'elemento più piccolo di A.

Lemma 18.6. *Sia $n \in \mathbb{N} + 2$. Allora il più piccolo divisore maggiore di 1 di n è primo. In particolare vediamo che esiste sempre un primo che divide n.*

Dimostrazione. Siccome n stesso è un divisore maggiore di 1 di n, l'insieme dei divisori maggiori di 1 di n è non vuoto, cosicché dal teorema 18.5 segue che esiste veramente il più piccolo divisore maggiore di 1 di n; lo chiamiamo p.

Se p non fosse primo, avrebbe a sua volta un divisore q maggiore di 1 e diverso da p e quindi più piccolo di p. q sarebbe anche un divisore di n, in contraddizione alla minimalità di p.

Osservazione 18.7. Siano $a, b, d \in \mathbb{Z}$ tali che $d|a$ e $d|b$. Allora $d|a + b$ e $d|a - b$.

Dimostrazione. Esercizio (facile, ma obbligatorio).

Teorema 18.8. *Esiste un numero infinito di numeri primi.*

Dimostrazione. *Prima dimostrazione - di Hermite, la più breve:*

Dimostriamo che per ogni $n \in \mathbb{N}$ esiste un primo maggiore di n. Ma per il lemma 18.6 esiste un primo p con $p|n! + 1$. Se fosse $p \leq n$, si avrebbe $p|n!$ e quindi $p|1$ per l'oss. 18.7, e ciò è impossibile.

Seconda dimostrazione - di Euclide, la più vecchia:

Assumiamo che $p_1, \ldots, p_k$ siano tutti i numeri primi esistenti. Sia $a := p_1 \cdots p_k + 1$. Allora $a > 1$ e dal lemma 18.6 sappiamo che esiste un primo p che divide a. Per ipotesi allora p deve essere uno dei p_j. Per l'oss. 18.7 ciò implica $p_j|1$, una contraddizione.

La funzione $\pi(n)$

Definizione 18.9. Per $n \in \mathbb{N}$ sia $\pi(n)$ il numero dei primi $p \leq n$.

Teorema 18.10 (teorema dei numeri primi). $\lim_{n\to\infty} \dfrac{\pi(n)}{n/\log n} = 1.$

Dimostrazione. La dimostrazione di questo teorema ha richiesto più di cento anni, dopo che era stato congetturato dal giovane Gauß.

Nota 18.11. Il teorema dei numeri primi ha un'interpretazione molto concreta. La relazione $\lim_{n\to\infty} \dfrac{\pi(n)}{n/\log n} = 1$ dice infatti che

"All'incirca ogni ($\log n$)-esimo numero $\leq n$ è primo."

Siccome $\log n$ è approssimativamente uguale a $2.3 \log_{10} n$, possiamo anche dire che

"All'incirca ogni ($2.3 \log_{10} n$)-esimo numero $\leq n$ è primo."

Ad esempio

"All'incirca ogni 23-esimo numero $\leq 10^{10}$ è primo."

"All'incirca ogni decimo numero $\leq e^{10} = 22026$ è primo."

Il teorema di Green-Tao

Osservazione 18.12. Nonostante il teorema 18.10 la distribuzione dei numeri primi è piuttosto irregolare o almeno ancora non del tutto compresa. Sono molto difficili le asserzioni sulle distanze tra numeri primi consecutivi. È proprio qui che attorno al 2004 Green e Tao sono riusciti ad ottenere un sensazionale risultato.

Definizione 18.13. Una *progressione aritmetica* è una successione (finita o infinita) crescente di numeri naturali in cui due termini adiacenti hanno sempre la stessa distanza:

$a, a + d, a + 2d, a + 3d, \ldots,$ con $a \in \mathbb{N}$ e $d \geq 1$.

Il numero degli elementi delle successione (che può essere infinito) si chiama la lunghezza della progressione aritmetica.

Osservazione 18.14. Non esiste una progressione aritmetica infinita che consiste solo di numeri primi.

Dimostrazione. Sia $a, a + d, a + 2d, a + 3d, \ldots$, una progressione aritmetica infinita. Ciò significa che ogni termine $a + kd$ con $k \in \mathbb{N}$ appare nella progressione, in particolare $x = a + (2 + a + 2d)d$.

Però $x = a + 2d + (a + 2d)d = (a + 2d)(1 + d)$. Entrambi i fattori sono ≥ 2 perché $d \geq 1$. Perciò x non è primo.

Teorema 18.15 (Green-Tao). *Per ogni $n \in \mathbb{N} + 1$ esiste una progressione aritmetica di lunghezza n che consiste solo di numeri primi.*

Dimostrazione. La dimostrazione è complicata e consiste in una sofisticata combinazione di strumenti della teoria dei numeri, del calcolo combinatorio, dell'analisi armonica e della teoria delle probabibilità.

Lo schema di Ruffini

Sia dato un polinomio

$$f = a_0x^n + a_1x^{n-1} + \cdots + a_n \in A[x]$$

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:

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

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

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

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

$$\begin{aligned}
\alpha g &= \alpha b_0x^{n-1} + \alpha b_1x^{n-2} + \cdots + \alpha b_{n-1}\\
&= (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \dots\\
&\quad + (b_{n-1} - a_{n-1})x + b_n - a_n\\
&= (b_1x^{n-1} + b_2x^{n-2} + \cdots + b_{n-1}x + b_n)\\
&\quad - (a_1x^{n-1} + a_2x^{n-2} + \dots a_{n-1}x + a_n)\\
&= x(g - b_0x^{n-1}) + b_n - (f - a_0x^n)\\
&= xg - b_0x^n + b_n - f + a_0x^n = xg + b_n - f
\end{aligned}$$

quindi

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

e ciò implica $f(\alpha) = b_n$.

$b_0, \dots, b_{n-1}$ 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 Ruffini* (nella letteratura inglese *schema di Horner*, ma la priorità spetta a 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`.

In Python possiamo realizzare l'algoritmo di Ruffini nel modo seguente:

```
def ruffini (a,x,completo=0):
  bk=0
  if not completo:
    for ak in a: bk=bk*x+ak
    return bk
  b=[]
  for ak in a: bk=bk*x+ak; b.append(bk)
  return b
```

Nella variante semplice (con `completo=0`) otteniamo solo il valore di $f(\alpha)$, se poniamo `completo=1` la funzione restituisce la lista $[b_0, b_1, \dots, b_n]$. Perché nella terz'ultima riga non iniziamo con `else`?

```
a=[3,5,6,8,7]

print(ruffini(a,10))
# 35687

print(ruffini(a,10,completo=1))
# [3, 35, 356, 3568, 35687]
```

Una frequente applicazione dello schema di Ruffini è il calcolo del valore corrispondente a una rappresentazione binaria, esadecimale o, più in generale, b-adica.

Otteniamo così $(100110111)_2$ come

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

e $(AF7305E)_{16}$ come

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

u=ruffini([1,0,0,1,1,0,1,1,1],2)
print (u) # 311

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

Somme trigonometriche II

Per calcolare somme trigonometriche della forma $\sum_{n=0}^{N} a_n \cos nx$ possiamo usare lo schema di Ruffini, ottenendo così un algoritmo notevolmente più efficiente del calcolo diretto visto a pagina 13. La rappresentazione

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

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 $z_2 := e^{-ix} = 1/z_1$, 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 Ruffini. Per l'esponenziale complesso utilizziamo il modulo `cmath` di Python (pag. 9).

```
import cmath

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

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

print (sommacoseni(a,0.2))
# (14.7458647279+1.7763568394e-15j))
```

Il termine immaginario è un artefatto numerico e in verità uguale a zero.

Il più piccolo divisore

Per trovare il più piccolo divisore $p > 1$ di un numero intero $n \neq \pm 1, 0$ (cfr. lemma 18.6) possiamo ancora usare il fatto che, se n non è primo, si deve avere $p \leq \sqrt{n}$ (oss. 18.4).

```
def divmin (n):
  r=int(math.sqrt(n))
  for d in range(2,r+1):
    if not n%d: return d
  return n
```

Lo sviluppo di Taylor

Sia dato un polinomio

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

Fissato α, con lo schema di Ruffini 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 Ruffini (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

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

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

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

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 usare la funzione `ruffini` definita a pag. 19 con il parametro `completo=1`, calcolando così non solo il valore b_n, ma tutti i b_k. Con essa otteniamo la funzione `taylor` che ripete il procedimento fino a quando il vettore dei coefficienti è vuoto:

```
import copy

def taylor (a,x):
  v=copy.deepcopy(a); w=[]
  while v: v=ruffini(v,x,completo=1); 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

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

L'uso di `v=copy.deepcopy(a)` nella funzione `taylor` non è necessario; sarebbe corretto anche `v=a`.

Bisogna stare molto attenti però in questi casi. Che qui si possa anche fare a meno di `copy.deepcopy` dipende *esclusivamente* dal fatto che la prima istruzione `v.pop()` viene eseguita *dopo* un'assegnazione `v=ruffini(...)` che libera `v` dal legame con `a`.

Le serie di Taylor vengono studiate in dettaglio nei corsi di Analisi. Infatti molte funzioni reali (non solo i polinomi, ma anche le funzioni trigonometriche e iperboliche, la funzione esponenziale, i quozienti di polinomi) possono, per ogni punto x_0 del loro dominio di definizione ed ogni x sufficientemente vicino ad x_0 (e spesso per ogni $x \in \mathbb{R}$) essere rappresentate mediante la loro serie di Taylor:

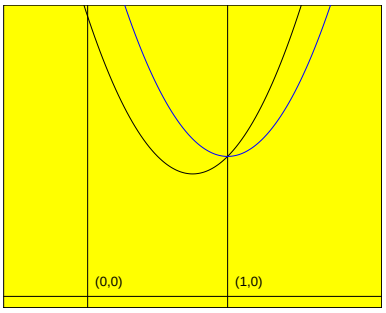
$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

dove $f^{(k)}(x_0)$ è la k-esima derivata di f nel punto x_0. Questa rappresentazione fornisce molte informazioni sul comportamento della funzione vicino al punto x_0.

Anche quando f è una funzione polinomiale la serie di Taylor in x_0 è interessante, perché ci dice come la f si esprime in x_0. In questo caso naturalmente la serie di Taylor è formalmente un polinomio in $x - x_0$ dello stesso grado di quello di partenza. Abbiamo ad esempio

$$2 - 3x + 2x^2 = 1 + (x - 1) + 2(x - 1)^2$$
$$3 - 4x + 2x^2 = 1 + 2(x - 1)^2$$

e vediamo che le due funzioni coincidono in $x_0 = 1$, ma vicino ad $x_0 = 1$ il secondo polinomio si comporta come la funzione $y = 1+2x^2$ in $x = 0$, il primo invece piuttosto come la retta $y = 1 + x$.



Alcune importanti serie di Taylor (richieste, senza le dimostrazioni, all'esame):

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - + \ldots$$
$$\cosh x = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} + \ldots$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - + \ldots$$
$$\sinh x = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} + \ldots$$
$$\frac{1}{1 - x} = 1 + x + x^2 + x^3 + x^4 + \ldots \quad \text{per } |x| < 1$$

Si osservi che $\cosh x + \sinh x = e^x$ e si provi a verificare che $\cos x + i \sin x = e^{ix}$, in accordo con quanto visto alle pagg. 8-9.

La distanza di Hamming

La distanza di Hamming è definita come il numero delle coordinate in cui due elementi di $\{0, 1\}^n$ differiscono. Essa in Python può essere calcolata con la seguente semplicissima funzione:

```
def hamming (a,b):
  return sum(map(lambda x,y: x!=y,a,b))
```

Esempio:

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

print(hamming(a,b)) # 7
```

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

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. Essa 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

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

for n in list(range(21)) + [48,77,106,135,164,194,221]:
  print ("%3d  %s" %(n,strdalista(rapp2(n,cifre=8))))
```

con

```
  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
 13  0 0 0 0 1 1 0 1
 14  0 0 0 0 1 1 1 0
 15  0 0 0 0 1 1 1 1
 16  0 0 0 1 0 0 0 0
 17  0 0 0 1 0 0 0 1
 18  0 0 0 1 0 0 1 0
 19  0 0 0 1 0 0 1 1
 20  0 0 0 1 0 1 0 0
 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
```

Se usiamo invece

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

otteniamo gli elementi dell'ipercubo 2^8 (cfr. pagina 22).

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.

Di nuovo 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]
```

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`, come visto a pag. 10:

```
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),end=' ')
# 8 c 18 3c 50 ff 100

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

L'ultimo esempio è un comodo metodo per ricordarsi il numero telefonico del Dipartimento di Matematica o un qualsiasi altro numero telefonico che non inizia con 0. Ma bisogna saper tornare indietro:

```
def rappcomenumero (a,base):
  def decodifica(x):
    s=ord(x)
    if 48<=s<=57: return s-48
    return s-87
  v=map(decodifica,a)
  return ruffini(v,base)

print (rappcomenumero('kvjm',36)) # 974002
```

In verità, per tornare indietro possiamo però semplicemente usare la funzione `int`, come abbiamo già fatto a pag. 10:

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

Per numeri rappresentati in forma binaria o esadecimale possiamo similmente usare `int` con `base=2` risp. `base=16`.

In `rappcomestringa` e `rappcomenumero` abbiamo usato che `ord(x)` è il codice ASCII di un carattere `x`, e che i numeri 0-9 corrispondono ai codici ASCII 48-57, le lettere minuscole ai codici ASCII 97-122, per cui dobbiamo sottrarre 87, se vogliamo associare ad `a` il valore 10, ecc., mentre `chr(m)` è il carattere con codice ASCII `m`.

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

Nella fondazione insiemistica della matematica $0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, ecc., quindi anche $\{0, 1\}^n = 2^n$.

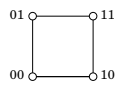
Sia $X = \{1, \ldots, n\}$ con $n \geq 0$.

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

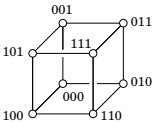
$n = 0$:

$n = 1$:

$n = 2$:



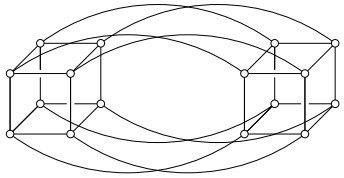
$n = 3$:



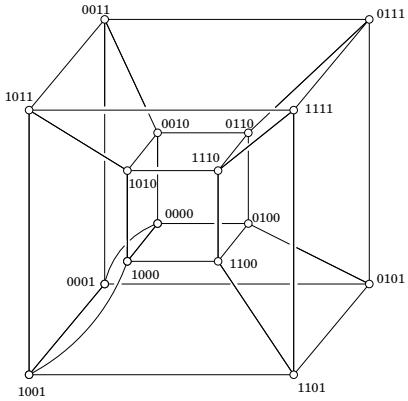
$n = 4$: L'ipercubo a 4 dimensioni si ottiene dal cubo 3-dimensionale attraverso la relazione

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

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 $X = \{1, 2, 3, 4\}$).

Impostare il limite di ricorsione

Se definiamo il fattoriale con

```
def fatt (n):
  if n<=1: return 1
  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 della pila utilizzata per l'esecuzione delle funzioni, inizialmente impostato a 1000. Si può ridefinire questo limite con la funzione `sys.setrecursionlimit`:

```
import sys

sys.setrecursionlimit(2000)
print (fatt(1998)) # Funziona.
```

Il limite di ricorsione si ottiene con `sys.getrecursionlimit`:

```
sys.setrecursionlimit(2000)
print (sys.getrecursionlimit())
# 2000
```

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 range(n-1): a,b=a+b,a
  return a
```

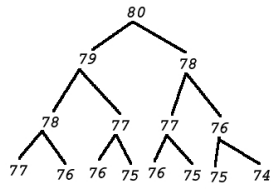
Possiamo visualizzare i numeri di Fibonacci da F_0 a F_12 e da F_{50} a F_{61} con le seguenti istruzioni:

```
for n in range(13): print (fib(n),end=' ')
# 1 1 2 3 5 8 13 21 34 55 89 144 233

print()
for n in range(50,62): print (fib(n))

# Output:

20365011074
32951280099
53316291173
86267571272
139583862445
225851433717
365435296162
591286729879
956722026041
1548008755920
2504730781961
4052739537881
```

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, quindi il nostro computer 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 range(50,61): print (fibs(n)[0])
```

ottenendo in modo fulmineo il risultato.

Un generatore per i numeri di Fibonacci

Imitando la funzione `fib` possiamo creare un generatore per i numeri di Fibonacci:

```
def generafib (n):
  a=0; b=1
  for k in range(n): a,b=a+b,a; yield a

fib16 = generafib(16)
for x in fib16: print (x,end=' ')
# 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

La lista dei numeri di Fibonacci

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

```
def fiblista (n):
  if n==0: return [1]
  v=[1,1]
  for i in range(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.

Il massimo comune divisore

Definizione 23.1. 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.

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 24.1. *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 mcdr (a,b):
  if a<0: a=-a
  if b<0: b=-b
  if b==0: return a
  return mcdr(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

$$a = \alpha b + c$$
$$d = bx' + cy'$$

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

a=7464; b=3580
print (mcde(a,b)) # (4, 106, -221)
```

sort

Sappiamo dalla pagina 6 che il metodo `sort` permette di ordinare una lista. Per una lista `a` la sintassi completa è

```
a.sort(key=None,reverse=False)
```

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=str.lower` possiamo ordinare una lista di stringhe alfabeticamente, prescindendo dalla distinzione tra minuscole e maiuscole.

Consideriamo invece una lista `a` minore di una lista `b` se `a` ha meno elementi di `b`:

```
v=[[6,2,0],[9,1],[4,5,8,3],[3,1,7]]
v.sort(key=len)
print (v)
# [[9,1],[6,2,0],[3,1,7],[4,5,8,3]]
```

La mediana

La *mediana* di una sequenza *ordinata* di numeri $(a_1, \ldots, a_n)$ è definita come $a_{\frac{n-1}{2}+1}$ se n è dispari e $\dfrac{a_{\frac{n}{2}} + a_{\frac{n}{2}+1}}{2}$ se n è pari.

Se la sequenza non è ordinata, dobbiamo prima ordinarla!

Possiamo quindi programmare una funzione per calcolare la mediana, tenendo conto del fatto che in Python la numerazione degli elementi di una sequenza inizia da 0.

```
def mediana (*a):
  a=list(a); a.sort(); n=len(a)
  if n%2: return a[n//2]
  return 0.5*(a[n//2-1]+a[n//2])

print (mediana(3,2,1,6,4,8,5))
# 4
print (mediana(*[3,2,1,6,4,8,5]))
# 4
print (mediana(3,2,1,6,4,8,5,2))
# 3.5
```

Infatti nel primo esempio la sequenza ordinata è $(1, 2, 3, 4, 5, 6, 8)$. La sequenza è di lunghezza dispari, quindi la mediana coincide con l'elemento nel mezzo che è uguale a 4.

Nel secondo esempio la sequenza ordinata è $(1, 2, 2, 3, 4, 5, 6, 8)$. La sequenza è di lunghezza pari, quindi la mediana coincide con la media aritmetica dei due elementi centrali, cioè di 3 e 4.

Esistono algoritmi diretti per il calcolo della mediana senza effettuare un ordinamento, ma sono molto complicati.

Gli algoritmi del contadino russo

Esiste un algoritmo leggendario del contadino russo per la moltiplicazione di due numeri, uno dei quali deve essere un numero naturale. Nella formulazione matematica ricorsiva questo algoritmo si presenta nel modo seguente. Sia f la funzione di due variabili definita da $f(x,n) := nx$. Allora

$$f(x,n) = \begin{cases} 0 & \text{se } n = 0 \\ f(2x, n/2) & \text{se } n \text{ è pari } \neq 0 \\ x + f(x, n-1) & \text{se } n \text{ è dispari} \end{cases}$$

In Python ciò corrisponde alla funzione

```
# Moltiplicazione russa.
def f (x,n):
  if n==0: return 0
  if n%2==0: return f(x+x,n/2)
  return x+f(x,n-1)
```

Naturalmente il prodotto nx die due numeri in Python lo otteniamo più semplicemente con `n*x`. L'algoritmo può però essere utile in un contesto di calcolo (in un $\mathbb{N}$-*modulo*, come si dice in algebra, ad esempio in un *gruppo abeliano*) per il quale il Python non fornisce direttamente una funzione per la moltiplicazione con fattori interi. Lo stesso vale per il calcolo di potenze x^n (che, per numeri, in Python si ottengono con `x**n`). Anche qui esiste un algoritmo del contadino russo che può essere formulato così:

Sia g la funzione di 2 variabili definita da $g(x,n) := x^n$. Allora

$$g(x,n) = \begin{cases} 1 & \text{se } n = 0 \\ g(x^2, n/2) & \text{se } n \text{ è pari } \neq 0 \\ x \cdot g(x, n-1) & \text{se } n \text{ è dispari} \end{cases}$$

In Python definiamo la funzione nel modo seguente:

```
# Potenza russa.
def g(x,n):
  if n==0: return 1
  if n%2==0: return g(x*x,n/2)
  return x*g(x,n-1)

print (g(2,32))
# 4294967296
```

Confrontando i due casi, ci si accorge che l'algoritmo è sempre lo stesso - cambia soltanto l'operazione di $\mathbb{N}$ sugli argomenti!

Linearizzare una lista annidata

Come possiamo ottenere da una lista (o tupla) annidata

`[3,5,[6,1,5],[2,(4,5,8),6],1,(3,4)]`

la lista di tutti gli oggetti atomari (cioè che non siano liste o tuple) che in essa appaiono, nel nostro esempio la lista

`[3,5,6,1,5,2,4,5,8,6,1,3,4]`?

Per questo compito possiamo usare la funzione

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

Applichiamo la funzione al nostro esempio:

```
a=[3,5,[6,1,5],[2,(4,5,8),6],1,(3,4)]
print (linearizzalista(a))
# [3, 5, 6, 1, 5, 2, 4, 5, 8, 6, 1, 3, 4]
```

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 grando 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; in questo modo possiamo controllare se le tre regole di gioco non sono state violate.

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

a=[0,1,2,3]; b=[]; c=[]
hanoi(a,b,c)
print(a,b,c)
```

Otteniamo l'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] []
```

IX. STRINGHE

Le costanti del modulo string

Il modulo `string` contiene alcune utili costanti le quali vengono normalmente usate come contenitori di caratteri.

```
import string

print (string.ascii_letters)
# abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

print (string.ascii_lowercase)
# abcdefghijklmnopqrstuvwxyz

print (string.ascii_uppercase)
# ABCDEFGHIJKLMNOPQRSTUVWXYZ

print (string.digits)
# 0123456789

print (string.hexdigits)
# 0123456789abcdefABCDEF

print (''.join([x for x in string.hexdigits \
    if not x in string.ascii_lowercase]))
# 0123456789ABCDEF

print (string.octdigits)
# 01234567

stampabili=string.printable
# Stringa che contiene i caratteri ASCII stampabili.
u=[i for i in range(128) if i>=32 and chr(i) not in stampabili]
print (u) # [127]

print (string.punctuation)
# !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

bianchi=string.whitespace
# Stringa che contiene i caratteri bianchi.
u=[i for i in range(128) if chr(i) in bianchi]
print (u) # [9, 10, 11, 12, 13, 32]
```

upper, lower e swapcase

Per trasformare tutte le lettere di una stringa in maiuscole risp. minuscole si usano i metodi `upper` e `lower`:

```
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
```

capitalize e string.capwords

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:

```
import string

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
```

Unione di stringhe con + e join

Stringhe possono essere unite, come tutte le sequenze, mediante l'operatore `+` oppure, come abbiamo già visto in alcuni esempi, 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. Altri esempi:

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

def numtel (numero,prefisso='0532'): # Cfr. pag. 14.
  return '-'.join([prefisso,numero])

print (numtel('974002'))
# 0532-974002
```

`join` viene usato piuttosto frequentemente, in genere con i separatori `' '` e `'\n'` oppure ad esempio `', '`.

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`.

`b` adesso è una lista, dalla quale ricaviamo una parola usando il metodo `join`:

```
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
```

Ricerca in stringhe

Per la ricerca in stringhe sono disponibili i seguenti metodi. Come osservato a pagina 6, `index` e `count` possono essere usati anche per stringhe. `index` non è però 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
```

Si noti che il metodo `count` per stringhe accetta fino a 3 argomenti, a differenza dall'omonimo metodo per sequenze generali (pag. 6) che accetta solo un argomento.

Sostituzioni in stringhe

Cesare codificava talvolta i suoi messaggi sostituendo ogni lettera con la lettera che si trova a 3 posizioni dopo la lettera originale, calcolando in maniera ciclica. Assumiamo di avere un alfabeto a 26 lettere (senza maiuscole, spazi o interpunzioni):

```
def cesare (a):
  o=ord('A') # Usiamo ord e chr viste a pag. 21.
  def codifica (x): n=(ord(x)-o+3)%26; return chr(o+n)
  return ''.join([codifica(x) for x in a])

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

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`.

```
print ('andare, creare, stare'.replace('are','ava'))
# andava, creava, stava

print ('ararat'.replace('ara','ava')) # avarat
```

split

Questo metodo, usato molto spesso (insieme all'analogo ma più potente metodo `split` delle espressioni regolari), permette di decomporre una stringa in una lista di stringhe. `a` ed `s` siano stringhe, $\mathtt{n} \in \mathbb{N}+1$.

`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 ([x.strip() for x in a.split(',')])
# ['Roma', 'Como', 'Pisa']

b='0532 Comune di Ferrara'
print (b.split(' ',1))
# ['0532', 'Comune di Ferrara']
```

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]
```

Stringhe formattate I

Abbiamo già usato varie volte 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 a 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 `float` 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:

```
u=50000/4.3; v=20
a='u=%6.2f\nv=%6.2f' %(u,v)
print (a)

# u=11627.91
# 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	`float`
%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 %%. Se però manca il % esterno, il carattere `'%'` non è considerato speciale:

```
print ('%d %%' %10)
# 10 %
print ('20 %%')
# 20 %%
print ('20 %')
# 20 %
```

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 può essere specificato il numero dei caratteri da usare; con `*` questo numero iventa 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=[x*x for x in (0,1,2)]
print ('I primi tre quadrati sono %d, %d e %d.' %tuple(u))
# I primi tre quadrati sono 0, 1 e 4.
```

Tramite stringhe formattate si possono anche costruire semplici tabelle. Con

```
stati=dict(Afghanistan=[652000,21000],Giordania=[89000,6300],
  Kazakistan=[2725000,14900],Mongolia=[1587000,2580],
  Turchia=[780000,65000],Uzbekistan=[448000,25000])

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

otteniamo

```
stato       | superficie/kmq | ab/1000
--------------------------------------
Afghanistan |         652000 |   21000
Giordania   |          89000 |    6300
Kazakistan  |        2725000 |   14900
Mongolia    |        1587000 |    2580
Turchia     |         780000 |   65000
Uzbekistan  |         448000 |   25000
```

Una tabella del coseno:

```
import math

for k in range(10):
  x=10*k*math.pi/180
  print ('| %2d | %6.3f |' %(10*k,math.cos(x)))

# Output:

|  0 |  1.000 |
| 10 |  0.985 |
| 20 |  0.940 |
| 30 |  0.866 |
| 40 |  0.766 |
| 50 |  0.643 |
| 60 |  0.500 |
| 70 |  0.342 |
| 80 |  0.174 |
| 90 |  0.000 |
```

In una delle prossime versioni di Python potrebbe essere abolita la sintassi classica per la formattazione di stringhe che abbiamo imparato su questa pagina. È stato invece introdotto un nuovo metodo di formattazione che descriveremo adesso. Questo metodo, in fondo superfluo, sembra comunque più fragile e più difficile da memorizzare.

Stringhe formattate II

Una stringa formattata (nella nuova sintassi) è un'espressione della forma `a.format(...)`, dove `a` è una stringa e i puntini corrispondono a parametri indicati secondo le regole che adesso elencheremo. `a` contiene dei campi speciali, definiti da parentesi graffe che corrispondono ai campi `%...s`, `%...f` ecc. dei formati in stile C.

Ogni campo speciale consiste o soltanto di un identificatore, oppure da un identificatore seguito da un doppio punto e una sigla di specifica.

Esempio per il primo caso:

```
a='Mi chiamo {0}, abito a {1}.'.format(nome,residenza)
```

Esempio per il secondo caso:

```
a='x={0:4.2f}, y={1:4.2f}'.format(u,v)
```

Sono possibili anche forme più generali che adesso esamineremo.

Gli identificatori possono essere numeri naturali oppure anche nomi che vengono usati come parametri con nome di funzioni, ad es.

```
a='Mi chiamo {a}.'.format(a=nome)
```

Una caratteristica interessante è che un identificatore può anche corrispondere a una struttura composta `A`, di cui possono essere usate le componenti, ad es. `A[i]` oppure `A.d`.

Alcuni esempi, da studiare con attenzione:

```
a='largo {0} cm, alto {1} cm'.format(30,15)
print (a)
# largo 30 cm, alto 15 cm

b='largo {lar} cm, alto {alt} cm'.format(lar=30,alt=15)
print (b)
# largo 30 cm, alto 15 cm

print ('{0} {x} {2} {1}'.format(10,11,12,x=99))
# 10 99 12 11

u=[1,5,7,3,4]

for i in range(4): print ('U[{0}] = {1}'.format(i,u[i]))
# U[0] = 1
# U[1] = 5
# U[2] = 7
# U[3] = 3

print ('W[{r[2]}] = {u[2]}'.format(r=range(4),u=u))
# W[2] = 7

class punto:
  def __init__ (A,x,y,z): A.x=x; A.y=y; A.z=z

P=punto(3,5,1)
print ('x={p.x}, y={p.y}, z={p.z}'.format(p=P))
# x=3, y=5, z=1

latino = {'casa': 'domus','villaggio': 'pagus',
  'nave':'navis', 'campo':'ager'}
voci=sorted(latino)
for x in voci: print ('{0} = {1}'.format(x,latino[x]))
# campo = ager
# casa = domus
# nave = navis
# villaggio = pagus

for x in voci: print ('{0:9} = {1}'.format(x,latino[x]))
# campo     = ager
# casa      = domus
# nave      = navis
# villaggio = pagus
```

Se manca l'estensione `.format(...)`, le parentesi graffe in una stringa vengono riprodotte così come sono. In una stringa formattata si possono usare `{{` per `{` e `}}` per `}` (con qualche eccezione).

Per comprendere le sigle di specifica nelle nuove sigle di formato, elenchiamo le equivalenze con le corrispondenti sigle di formato in stile C. `a` sia un identificatore.

```
%s    ... non necessita di una specifica
%d    ... non necessita di una specifica
%f    ... {a:f}
%x    ... {a:x}
%X    ... {a:X} (esadecimali maiuscole)
%c    ... {a:c}
%8s   ... {a:8}
%-8s  ... {a:<8}
%4.3f ... {a:4.3f}
```

Con `{a:^}` si ottiene un allineameno centrale.

Altri esempi:

```
for x in (0,1,0.0,[],(),[0],None,'','alfa'):
  print ('{0:<6}{1}'.format(x,bool(x)))

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

m=124; x=87345.99
a='{0:3} {1:<12.4f}'.format(m,x)
print (a)
# 124 87345.9900

u=50000/4.3; v=20
a='u={u:6.2f}\nv={v:6.2f}'.format(u=u,v=v)
print (a)
# u=11627.91
# v= 20.00

a=58; b=11; c=6
print ('{a:2x}{b:2x}{c:2x}'.format(a=a,b=b,c=c))
# 3a b 6
print ('{a:02x}{b:02x}{c:02x}'.format(a=a,b=b,c=c))
# 3a0b06

a='Ferrara'; b='Roma'; c='Rovigo'
m=max(len(a),len(b),len(c))
print ('|{a:{m}}|\n|{b:{m}}|\n|{c:{m}}|'.format(a=a,b=b,c=c,m=m))
# |Ferrara|
# |Roma   |
# |Rovigo |
```

Il modulo textwrap

Il modulo `textwrap` fornisce le funzioni `wrap` e `fill`. Se `a` è una stringa, allora `textwrap.wrap(a,width=70,...)` è una lista di stringhe, nessuna delle quali è più lunga di quanto indicato nel parametro opzionale `width`, mentre `textwrap.fill(a,width=70,...)` è il testo che si ottiene unendo le stringhe ottenute con `textwrap.wrap` mediante `'\n'` ed è quindi un'abbreviazione per `'\n'.join(textwrap.wrap)`.

Il parametro opzionale `break_long_words` è inizialmente impostato a 1 e in tal caso parole più lunghe di `width` vengono spezzate; ciò non accade se si pone questo parametro uguale a zero.

X. FILE, CARTELLE, MODULI

Lettura e scrittura di file

Per leggere un file utilizziamo la funzione

```
def leggifile (nome):
  f=open(nome,'r'); testo=f.read(); f.close(); return testo
```

che ci restituisce il contenuto del file in un'unica stringa. La sigla `'r'` sta per *read*. Per poter scrivere un testo su un file, definiamo la funzione

```
def scrivifile (nome,testo):
  f=open(nome,'w'); f.write(testo); f.close()
```

Quando un file viene aperto con la modalità di scrittura usando la sigla `'w'` (per *write*) come secondo argomento di `open`, 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'` (per *append*):

```
def aggiungifile (nome,testo):
  f=open(nome,'a'); f.write(testo); f.close()
```

Queste operazioni possono naturalmente essere utilizzate anche direttamente (al di fuori di una funzione):

```
f=open('uff','w')
f.write('Maria Tebaldi\n')
f.write('Roberto Magari\n')
f.write('Erica Rossi\n')
f.close()
```

Talvolta nelle operazioni di lettura e scrittura bisogna specificare la codifica dei caratteri, aggiungendo come terzo argomento di `open` ad esempio `encoding='latin-1'` oppure `encoding='utf-8'`.

Il modulo time

`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, pag. 247.
`time.sleep(s)`	Aspetta `s` secondi, ad esempio `time.sleep(0.25)`.

Questo modulo contiene alcune funzioni per tempo e data, di cui abbiamo elencato le più importanti. Esempi:

```
import time

t=time.localtime(); print (t)
# time.struct_time(tm_year=2010, tm_mon=12, tm_mday=3,
# tm_hour=21, tm_min=0, tm_sec=2, tm_wday=4, tm_yday=337,
# tm_isdst=0)

print (time.strftime('%d %b %Y, %H:%M',t))
# 03 Dec 2010, 21:01

print (time.ctime())
# Fri Dec  3 21:01:22 2010

print (time.time())
# 1291406482.53
for i in range(1000000): pass
print (time.time())
# 1291406482.67
```

Comandi per file 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 file 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`:

```
import os

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')
```

Queste funzioni sono utili e vengono chieste spesso all'esame.

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:

```
import sys

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`.

`sys.argv` viene spesso utilizzato 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.

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 f at 0x53a4b0>, '__builtins__': ...
'__file__': '.../prog-3101', '__package__': None,
'x': 7, '__name__': '__main__',
'eseguifile': <function ... >, '__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

Per creare una variabile il cui valore è una stringa che coincide con il nome della variabile, potremmo definirla 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, Carla) # Maria Carla
```

J. Orendorff: Comunicazione personale, febbraio 1998.

eseguifile

Purtroppo in Python 3 è stata abolita la funzione `execfile` con cui era possibile eseguire i comandi contenuti in un file. La possiamo imitare con la seguente funzione:

```
def eseguifile (nome):
  f=open(nome,'r',encoding='latin-1')
  testo=f.read(); f.close(); exec(testo,globals())
```

L'argomento `globals()` in `exec` fa in modo che i nomi di variabili e le istruzioni di assegnamento in `testo` valgano globalmente, come abbiamo già visto a pag. 15.

---

os.system e sys.exit(0)

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 file nella cartella attiva nel file *catalogo*.

Per uscire da Python dall'interno di un programma si usa `sys.exit(0)`.

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 file) con l'estensione `.pth` che contiene i nomi delle nuove cartelle di ricerca nell'apposita cartella, che si può trovare esaminando la lista `sys.path` con

```
for x in sys.path: print(x)
```

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, ad esempio

```
from math import cos,sin,exp
```

---

help

Con `help(oggetto)` si ottengono informazioni su un oggetto di Python. Provare

```
help(print)
help(dict)
```

XI. DIZIONARI

Modifica di un dizionario

Abbiamo incontrato esempi di dizionari nelle pagine 4 e 28. 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 = {('Roberto','Rossi','Trento') : 4000,
('Camillo','Rossi','Roma') : 8000,
('Mario','Gardini','Pisa') : 3400}

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

voci=list(stipendi)
voci.sort(key=lambda x: (x[1],x[0]))
for x in voci:
  print ('%-9s %-9s %-7s %5d' %(x[0],x[1],x[2],stipendi[x]))

# Output:

Mario     Gardini   Pisa     3400
Cesare    Neri      Roma     4500
Camillo   Rossi     Roma     8000
Roberto   Rossi     Trento   4000
```

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

Se `d` è un dizionario, `list(d)` 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 questo esempio.

Con `del d[x]` la voce `x` viene cancellata dal dizionario. Questa istruzione provoca un errore, se la voce `x` non esiste. Cfr. pag. 33.

---

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. Nel codice genetico a fianco potremmo quindi scrivere

```
codgen1=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}
```

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[x]) for x in chiavi if x in diz])
  # oppure return {x:diz[x] for x in chiavi if x in diz}

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

Per verificare che un dizionario `d` contiene una voce `x`, si può usare `x in d`.

Invertire un dizionario

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

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

print (list(diz.items()))
# [('a', 1), ('c', 3), ('b', 2), ('y', 6), ('x', 5), ('z', 7)]
```

Possiamo usare questo metodo per invertire un dizionario invertibile.

```
def inverti (diz):
  return dict([(y,x) for x,y in diz.items()])
  # oppure return {y:x for x,y in diz.items()}

diz=dict(a='0100',b='1100',i='0010')

def decodifica (a,diz):
  inv=inverti(diz); p=''
  for k in range(0,len(a),4): p+=inv[a[k:k+4]]
  return p

codifica='11000100110011000010'
print (decodifica(codifica,diz))
# babbi
```

Il codice genetico

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

```
codgen1 = {'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'}

codgen = {}
for aminoacido in codgen1:
  v=codgen1[aminoacido].split()
  for x in v: codgen[x]=aminoacido

dna='ttagattgcttggagtcatacttagataca'; n=len(dna)

for i in range(0,n,3):
  tripla=dna[i:i+3]
  print (codgen[tripla],end=' ')

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

In questo esempio abbiamo prima creato una tabella `codgen1` 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 `codgen` 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.

Uguaglianza di dizionari

Due dizionari sono uguali, quando possiedono le stesse voci e per ogni voce gli stessi valori:

```
u=dict(a=7,b=10)
v={'b':10, 'a':7}
w=dict(a=7,b=10,c=None)

print (u==v) # True
print (u==w) # False
```

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 (**diz): print (diz)
```

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, nel dizionario che viene generato i nomi delle variabili vengono trasformati in stringhe che corrispondono alle voci del dizionario.

Dizionari impliciti

Anche per i dizionari esiste una notazione implicita. La sintassi si evince dal seguente esempio in cui una funzione f viene trasformata nel dizionario delle coppie $(x, f(x))$ (matematicamente nel *grafico* di f):

```
def f (x): return x*x

grafico={x:f(x) for x in range(12) if x%2}

print (grafico)
# {1: 1, 3: 9, 5: 25, 7: 49, 9: 81, 11: 121}
```

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.

Il metodo get per dizionari

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.

```
diz={1:1, 3:9, 6:36, 8:64}

print (*[diz.get(x,0) for x in range(10)])
# 0 1 0 9 0 0 36 0 64 0
```

È molto utile!

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]
```

Nella funzione `hanoi` a pag. 25 abbiamo usato `del` per realizzare l'analogo di `pop` all'inizio di una stringa, come nella seguente funzione:

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

a=[1,7,2,5]; print (popin(a)) # 1
print (a) # [7,2,5]
```

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

Il metodo clear per dizionari

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

```
diz=dict(a=17,b=18,c=33,d=45)
print (diz)
# {'a': 17, 'c': 33, 'b': 18, 'd': 45}
diz.clear(); print (diz) # {}
```

XII. ESPRESSIONI REGOLARI

Insiemi di parole

Un'*espressione regolare* è una formula che descrive un *insieme di parole*. Questo importante concetto dell'informatica teorica è entrato in molti linguaggi di programmazione, soprattutto nel mondo Unix. Esponiamo nel seguito il formato generale delle espressioni regolari secondo la sintassi usata in Perl e come questo formato viene realizzato in Python tramite il modulo `re`.

Una parola che non contiene caratteri speciali come espressione regolare corrisponde all'insieme di tutte le parole che la contengono (ad esempio `alfa` è contenuta in `alfabeto` e `stalfano`, ma non in `stalfino`).

`^alfa` indica invece che `alfa` deve trovarsi all'inizio della parola (o della riga, a seconda della mancanza o presenza del modificatore `re.M`, come vedremo), `alfa$` che si deve trovare alla fine. È come se `^` e `$` fossero due lettere invisibili che denotano inizio e fine della parola (o della riga).

Il carattere (spazio) viene trattato come gli altri, quindi con `a alfa` si trova `kappa alfa`, ma non `alfabeto`.

Il punto `.` denota un carattere qualsiasi diverso dal carattere di nuova riga oppure un carattere qualsiasi a seconda della mancanza o presenza del modificatore `re.S`.

Un asterisco `*` non può essere usato da solo e indica una ripetizione arbitraria (quindi anche l'assenza) dell'espressione che lo precede. Perciò `a*` sta per le parole `a`, `aa`, `aaa`, ..., e anche per la parola vuota. Per questa ragione `alfa*ino` comprende `alfino` e ad esempio `alfaaino`. Per escludere la parola vuota si usa `+` al posto dell'asterisco.

Ad esempio `+` indica almeno uno e possibilmente più spazi vuoti. Questa espressione regolare viene usata spesso per separare le parti di una stringa. Per esempio `r *, +s` comprende `alfar , sbeta`, ma non `alfar ,sbeta`.

Il punto interrogativo `?` dopo un carattere (o un'espressione regolare α) indica che quel carattere (o α) può apparire una volta oppure mancare, quindi `alfa?ino` comprende `alfino` e `alfaino`, ma non `alfaaino`.

Le parentesi quadre vengono utilizzate per indicare *insiemi di caratteri* oppure il complemento di un tale insieme. `[aeiou]` denota le vocali minuscole e `[^aeiou]` tutti i caratteri che non sono vocali minuscole. È il cappuccio `^` che (all'interno di parentesi quadre, quando si trova all'inizio) denota il complemento.

Quindi `r[aeio]ma` comprende `rima` e `romano`, mentre `[Rr][aeio]ma` comprende anche `Roma`.

Si possono anche usare trattini per indicare insiemi di caratteri successivi nell'ordine alfabetico naturale, ad esempio `[a-zP]` corrisponde all'insieme di tutte le minuscole dell'alfabeto comune insieme alla `P` maiuscola, e `[A-Za-z0-9]` rappresenta i cosiddetti caratteri alfanumerici, cioè le lettere dell'alfabeto insieme alle cifre decimali. Per questo insieme si può usare l'abbreviazione `\w`, per il suo complemento `\W`.

La barra verticale `|` tra due o più espressioni regolari indica che almeno una di esse deve essere soddisfatta. Per creare *gruppi* si possono usare le parentesi tonde: `(a|b|c)` è la stessa cosa come `[abc]`, `r(oma|ume)no` comprende `romano` e `rumeno`.

Per indicare i caratteri speciali `.`, `*`, `^` `(` ecc. bisogna anteporgli `\`, ad esempio `\.` indica veramente un punto e non un carattere qualsiasi.

Abbiamo raccolto nella colonna a fianco il significato preciso dei caratteri speciali nelle espressioni regolari.

Esercizio: Quali insiemi di parole sono rappresentati dalle seguenti espressioni regolari?

```
^[^AEIOUaeiou]
^([aeiou]|[p-t])
[x-z][0-9]+.*xyz?$
^<a href *= *.+?>.+?</a>$
```

---

docs.python.org/py3k/library/re.html. Descrizione del modulo *re*.

J. Friedl: Mastering regular expressions. O'Reilly 2006.

I metacaratteri

I seguenti caratteri hanno un significato speciale nelle espressioni regolari:

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

e all'interno di `[ ... ]` anche `-`. Per privare questi caratteri del loro significato speciale, è sufficiente anteporgli un `\`. In Perl e nei linguaggi della shell di Unix anche il carattere `/` è speciale. I dettagli:

`\ ...` viene usato per dare a un carattere speciale il suo significato normale.

`| ...` indica scelte alternative che vengono esaminate da sinistra verso destra.

`() ...` Le parentesi tonde vengono usate in più modi. Possono servire a racchiudere semplicemente un'espressione per limitare il raggio d'azione di un'alternativa, per distinguere ad esempio `a(u|v)` da `au|v`, oppure per catturare una parte da usare ancora. Altri usi delle parentesi tonde verranno descritti separatamente.

`[ ] ...` Le parentesi quadre racchiudono insiemi di caratteri oppure il loro complemento (se subito dopo la parentesi iniziale `[` si trova un `^` che in questo contesto non ha più il significato di inizio di parola che ha al di fuori delle parentesi quadre).

`{ } ...` Le parentesi graffe permettono la quantificazione delle ripetizioni: `a{3}` significa `aaa`, `a{2,5}` comprende `aa`, `aaa`, `aaaa` e `aaaaa`.

`^ ...` Questo carattere indica l'inizio di parola (oppure, quando è presente il modificatore `re.M`, anche l'inizio di una riga) quando non si trova all'interno delle parentesi quadre, dove, se si trova all'inizio, significa la formazione del complemento.

`$ ...` indica la fine della parola o della riga a seconda che manchi o sia presente il modificatore `re.M`.

`* ...` L'asterisco è un quantificatore e indica che il simbolo (o l'espressione) precedente può essere ripetuto un numero arbitrario di volte (o anche mancare). Un `?` altera il comportamento di `*`, come vedremo.

`+ ...` Ha lo stesso significato di `*`, tranne che il simbolo (o l'espressione) deve apparire almeno una volta. `?` altera il comportamento di `+`.

`? ...` `a?` significa che `a` può apparire o no, con preferenza per il primo caso, `a??` invece con preferenza per il secondo caso.

`*?` significa che viene scelta *la corrispondenza più breve possibile* (altrimenti viene scelta la più lunga); un discorso analogo vale per `+?`.

`. ...` sta per un singolo carattere che deve essere diverso dal carattere di nuova riga se non è presente il modificatore `re.S`.

`- ...` all'interno di parentesi quadre può essere usato per denotare un insieme di caratteri attigui. Per avere un semplice `-` all'interno delle parentesi quadre si deve usare `\-`.

Stringhe grezze

Quando una stringa tra virgolette o apici viene preceduta da `r`, i caratteri speciali definiti nelle stringhe comuni (del Python, non delle espressioni regolari) dal carattere `\` perdono il loro significato. Esempio:

```
a='prima\nseconda'
print (a)
# prima
# seconda
a=r'prima\nseconda'
print (a) # prima\nseconda
```

Ricerca con re.search

`re.search` è la funzione fondamentale per la ricerca in un testo mediante le espressioni regolari.

α sia una stringa che usiamo per rappresentare un'espressione regolare e `testo` una stringa in cui vogliamo cercare α. Allora con `re.search(α,testo)` otteniamo un oggetto (che nel seguito chiamiamo `corr`) della classe (abbreviando il termine) `MatchObject`, se la ricerca ha avuto successo, altrimenti `None`. Nel primo caso possiamo usare le seguenti componenti di `corr`:

`corr.start() ...` *indice* della prima apparizione di α (più precisamente, siccome α è usata come espressione regolare, della prima apparizione di una parola corrispondente ad α) in `testo`.

`corr.end() ...` *indice* della posizione che *segue* l'ultimo carattere della prima apparizione di α in `testo`.

`corr.re ...` l'espressione α (un `PatternObject`).

`corr.string ...` la stringa `testo`.

`corr.group() ...` la stringa trovata. Vedremo fra poco che il metodo `group` può essere usato per estrarre *gruppi* di corrispondenze da `corr`.

`corr.groups() ...` verrà spiegato fra poco.

Esempi:

```
import re

testo='0123456789012345'
corr=re.search('4[856].[27]',testo)

print (corr)
# <_sre.SRE_Match object at 0xb7b77170>

print (corr.start(), corr.end())
# 4 8
print (testo[corr.start():corr.end()])
# 4567
print (corr.group())
# 4567
print (corr.re)
# <_sre.SRE_Pattern object at 0xb7c08708>
print (corr.string)
# 0123456789012345
```

I modificatori re.I, re.M e re.S

La funzione `re.search` ammette come terzo argomento un'*espressione numerica* che è formata tramite i seguenti *modificatori*:

`re.I ...` indica di ignorare la differenza tra minuscole e maiuscole.

`re.M ...` quando presente, `^` corrisponde anche all'inizio di una riga (cioè all'inizio della stringa oppure a una posizione preceduta da un carattere di nuova riga) e similmente `$` indica anche la fine di una riga (cioè la fine della stringa oppure una posizione a cui segue un carattere di nuova riga).

`re.S ...` Il punto `.` nelle espressioni regolari sta per un carattere qualsiasi diverso dal carattere di nuova riga. Aggiungendo `re.S` si ottiene che `.` comprende anche il carattere di nuova riga; si farà così quando con `.*` si vuole denotare una successione arbitraria di caratteri che si può estendere anche su più righe.

Aiuti mnemonici:

`I` ... *ignora*
`M` ... *multiriga*
`S` ... *superpunto*

Se, mentre si usa il modificatore `re.M`, ci si vuole riferire all'inizio della stringa, si utilizza `\A` (che senza `re.M` ha lo stesso significato di `^`); mentre similmente `\Z` indica la fine della stringa anche in presenza di `re.M` (ed è invece equivalente a `$` in assenza di `re.M`); l'uso di `\Z` è in Python un po' diverso da quanto avviene in Perl.

Questi simboli perdono naturalmente il loro significato speciale all'interno di parentesi quadre.

Per attivare più di uno dei tre modificatori, essi vengono legati da `|` (`or` effettuato bit per bit), ad esempio `re.M|re.S`.

Infatti il terzo argomento è un valore numerico che inizialmente viene posto uguale a 0. I modificatori corrispondono ai seguenti valori (ma non è importante): `re.I`=2, `re.M`=8, `re.S`=16. Esempi:

```
testo='Ferrara\nRoma\nFirenze\nPisa'

corr=re.search('',testo)
print (corr)
# <_sre.SRE_Match object at 0xb7b771e0>

corr=re.search('57',testo)
print (corr) # None

corr=re.search('...$',testo)
if corr: print (corr.group()) # isa

corr=re.search('...$',testo,re.M)
if corr: print (corr.group()) # ara

corr=re.search('e.p',testo,re.I)
print (corr) # None

corr=re.search('e.p',testo,re.S|re.I)
if corr: print (corr.group())
# e
# P
```

Sigle a un carattere

Abbiamo spiegato il significato di `\A` e `\Z` nell'articolo precedente. I simboli `\n` e `\t` vengono usati come in C e indicano il carattere di nuova riga e il tabulatore (anche al di fuori delle espressioni regolari). Esistono numerosi altri metasimboli, di cui elenchiamo quelli più comuni, sufficienti in quasi tutte le applicazioni:

`\w ...` carattere alfanumerico, equivalente a `[A-Za-z0-9]`.

`\W ...` carattere non alfanumerico.

`\d ...` equivalente a `[0-9]`; il `d` deriva da *digit* (cifra).

`\D ...` `[^0-9]`.

`\s ...` spazio bianco, normalmente `\t\n\r\f` (cfr. pag. 26).

`\S ...` non spazio bianco.

La funzione re.compile

Soprattutto quando un'espressione regolare deve essere usata più volte, conviene compilarla prima del suo utilizzo. Ciò può accelerare di molto un programma. La sintassi è `u=re.compile(α,λ)`, dove λ può mancare oppure avere lo stesso significato come il terzo argomento in `re.search`. Una volta creato l'espressione regolare compilata `u`, `search` può essere usato come metodo di questo oggetto. Esempio:

```
u=re.compile('...$',re.M)

testo='prima\nseconda\nterza'
corr=u.search(testo)
if corr: print (corr.group()) # ima
```

Similmente altre funzioni del modulo `re` diventano metodi utilizzabili con espressioni regolari compilate.

Parentesi tonde e gruppi

Le parentesi tonde possono essere usate semplicemente per raggruppare gli elementi di una parte di un'espressione regolare. Allo stesso tempo però il contenuto delle parti della corrispondenza trovata viene memorizzato nei gruppi 1, 2, 3, ..., che, se il risultato della ricerca è ancora `corr`, possono essere utilizzati *al di fuori* dell'espressione regolare stessa nella forma `corr.group(1)`, `corr.group(2)`, ecc.

`corr.group(0)` è invece equivalente a `corr.group()` e denota tutta la stringa trovata. Si può anche ottenere una *tupla* contenente tutti i gruppi trovati con `corr.groups()`.

```
import re

testo='fine 65 345 era 900 - 111'
corr=re.search('(\d+) +(\d+).*?(\d+)',testo)
if corr:
  for i in range(4): print (corr.group(i))
# 65 345 era 900
# 65
# 345
# 900

print (corr.groups())
# ('65', '345', '900')
```

Quando un'espressione regolare contiene delle alternative (`|`), può accadere che la ricerca abbia successo, ma che un gruppo appaia in un'alternativa che non corrisponde alla stringa cercata. In tal caso diciamo che il gruppo non partecipa al successo della ricerca. Il metodo `groups` lo elenca allora come `None` oppure come il valore del parametro opzionale `default`:

```
testo='fine a65 345 era 900'
corr=re.search('(yy)|a(\d+) +(\d+).*?(\d+)',testo)

print (corr.groups(default='--'))
# ('--', '65', '345', '900')
```

All'interno dell'espressione regolare ci si può riferire ai gruppi già trovati con `\1`, `\2` ecc. Nelle stringhe non dimenticare di anteporre `r`:

```
testo='era 56 83567  1560'
corr=re.search(r'(\d+).*\1.*(\1)',testo)
if corr: print (corr.groups())
# ('56', '56')
```

Ricerca massimale e ricerca minimale

α`*` cerca una corrispondenza di lunghezza *massimale* con una delle parola della forma α`*`. Per ottenere una corrispondenza di lunghezza minimale si aggiunge un punto interrogativo: α`*?`. Lo stesso discorso vale per α`+` e α`{m,n}`. Esempi:

```
import re

testo='era [uno] e [due]'

corr=re.search('\[(.*)\]',testo)
if corr: print (corr.group(1))
# uno] e [due

corr=re.search('\[(.*?)\]',testo)
if corr: print (corr.group(1))
# uno
```

Attenzione: Bisogna però tener conto del punto in cui si trova l'elaborazione:

```
testo='babaaaaa'
corr=re.search('(a+)',testo)
if corr: print (corr.group(1))
# a
```

Parentesi tonde condizionali

Esistono anche parentesi tonde il cui contenuto non viene memorizzato nei gruppi. Esse non occupano posto!

`(?:`α`)` ... semplice parentesi non memorizzata (raccoglie solo).

α`(?=`β`)` ... α deve essere seguita da β.

α`(?!`β`)` ... α non deve essere seguita da β.

`(?<=`α`)`β ... β deve essere preceduta da α.

`(?<!`α`)`β ... β non deve essere preceduta da α.

`(?i:`α`)` ... Attiva `re.I` per il contenuto della parentesi.

`(?s:`α`)` ... Lo stesso per `re.S`.

Sostituzione con re.sub

Sostituzioni vengono effettuate tramite la funzione `re.sub` con la sintassi `re.sub(`α`,s,testo,count=0)`, dove α è un'espressione regolare, `s` la stringa con cui ogni corrispondenza con α nella stringa `testo` viene sostituita, e `count`, se > 0, indica quante sostituzioni devono essere effettuate, mentre per `count=0` vengono effettuate tutte le sostituzioni possibili. Se `u` è un'espressione regolare compilata, è possibile anche la sintassi `u.sub(s,testo,count=0)`. La stringa modificata viene restituita come risultato della funzione, la stringa originale `testo` rimane invece invariata.

```
testo='a315b883x910'
print (re.sub(r'b\d+','',testo))
# a315x910
print (testo)
# a315b883x910

print (re.sub(r'(?<=x)\d+','',testo))
# a315b883x

testo='AalEmiAreOtAea'
print (re.sub('[a-z]','',testo))
# AEAOA

print (re.sub('[aeiou]','',testo))
# AlEmArOtA

u=re.compile('[aeiou]',re.I)
print (u.sub('',testo))
# lmrt

testo='andare area dormire stare'
print (re.sub('are','ava',testo))
# andava avaa dormire stava

print (re.sub('([ai])re( |$)',r'\1va\2',testo))
# andava area dormiva stava

print (re.sub('([ai])re(?= |$)',r'\1va',testo))
# andava area dormiva stava

testo='ababacodelbababbo'
print (re.sub('aba','ibi',testo))
# ibibacodelbibibbo

print (re.sub('aba','iba',testo))
# ibabacodelbibabbo
# Si vede che la sostituzione non e' ricorsiva.
```

L'esmpio mostra come eliminare caratteri multipli da una stringa:

```
testo='brrrrhhhh... che freddo!'
u=re.compile(r'([a-z])\1+')
print (u.sub(r'\1',testo))
# brh... che fredo!
```

re.split

La funzione `re.split` spezza una stringa in corrispondenza delle apparizioni di un'espressione regolare. Esempi:

```
testo='uno due tre'
print (re.split(' +',testo))
# ['uno', 'due', 'tre']

testo='uno,due, tre  quattro'
print (re.split('[, ]+',testo))
# ['uno', 'due', 'tre', 'quattro']

testo='uno,due, tre , quattro'
print (re.split(' *, *',testo))
# ['uno', 'due', 'tre', 'quattro']

testo='uno - due - tre'
print (re.split(' +- +',testo))
# ['uno', 'due', 'tre']

testo='uno-!due**=tre'
print (re.split(r'\W+',testo))
# ['uno', 'due', 'tre']

# Una tecnica che si usa molto spesso.
testo='uno, due,tre'
print (re.split('\s*,\s*',testo))
# ['uno', 'due', 'tre']
```

Quando l'espressione separatrice contiene dei gruppi, anche questi vengono riportati nello spezzamento. Si ricordi però che le parentesi condizionali `(?...)` non definiscono gruppi:

```
testo='uno---due==tre, quattro'
u=re.compile(r'(-+|=+|\s*,\s*)')
print (u.split(testo))
# ['uno', '---', 'due', '==', 'tre', ', ', 'quattro']

testo='uno---due==tre, quattro'
u=re.compile(r'(?:-+|=+|\s*,\s*)')
print (u.split(testo))
# ['uno', 'due', 'tre', 'quattro']
```

`re.split` ammette un parametro opzionale `maxsplit` che indica il numero massimo di *separazioni* da effettuare (si ottengono quindi `maxsplit+1` parti):

```
testo='ab cde fg tha uv'
print (re.split(' +',testo,2))
# ['ab', 'cde', 'fg tha uv']
```

Ancora sull'uso di | nelle espressioni regolari

`Franco (Nero|[a-z]+)` rileva `Franco Nero` e `Franco piangeva`, ma non `Franco Gotti`.

`a|b|c` rileva una corrispondenza con `a` oppure con `b` oppure con `c`, nell'ordine dato. Però `a|abc` non applica mai ad `abc` - viene subito scoperta la corrispondenza con `a`, dopodiché l'elaborazione continua con `bc`.

Le funzioni findall e finditer

Se l'espressione regolare α non contiene gruppi, `re.findall(`α`,testo)` restituisce una lista con tutte le apparizioni non sovrapposte di α in `testo`. Anche qui, se `u` è un'espressione regolare compilata, si può usare `u.findall(testo)`. È possibile aggiungere gli stessi modificatori come in `re.search` (cfr. pag. 35).

```
testo='ab ps45 za600 bs88'

print (re.findall('[a-z]+[0-9]*',testo))
# ['ab', 'ps45', 'za600', 'bs88']
```

Se α contiene invece dei gruppi, soltanto gli elementi che corrispondono ai gruppi vengono restituiti, nel formato che si evince dai seguenti esempi. Può essere molto utile.

```
print (re.findall('([a-z]+)[0-9]*',testo))
# ['ab', 'ps', 'za', 'bs']

print (re.findall('([a-z]+)([0-9]*)',testo))
# [('ab', ''), ('ps', '45'), ('za', '600'), ('bs', '88')]
```

In questo modo possiamo ad esempio estrarre da una stringa (che potrebbe essere stata prelevata da un file) i valori di certe variabili creando un dizionario:

```
testo='a=6, b=200, at=130'
t=re.findall('([a-z]+)\s*=\s*([0-9]+)',testo)
print (t)
# [('a', '6'), ('b', '200'), ('at', '130')]
diz=dict(t)
print (diz)
# {'a': '6', 'b': '200', 'at': '130'}
```

Con `finditer` al posto di `findall` si ottiene un iteratore invece di una lista.

Lettura a triple del DNA

Nel codice genetico l'aminoacido isoleucina è rappresentato dalle triple ATA, ATC e ATT. Una stringa di DNA deve però essere letta in triple, quindi il primo ATA (che inizia dalla seconda lettera) in

TATATCTGCAATTTGATAGATCGA

non verrà tradotto in isoleucina, perché appartiene in parte alla tripla TAT iniziale e in parte ad ATC. Presentiamo prima un modo *errato* di lettura, poi una versione corretta in cui si fa uso di una lista implicita con clausola condizionale:

```
dna='TATATCTGCAATTTGATAGATCGA'

u=re.compile('AT[ACT]')

t=u.findall(dna)
print (' '.join(t))
# Output errato, perche' non sono
# queste le triple che vengono lette:
# ATA ATT ATA ATC

t=re.findall('...',dna)
s=[x for x in t if u.search(x)]
print (' '.join(s))
# ATC ATA
```

In una delle grandi banche dati biologiche abbiamo trovato la sequenza di DNA di una proteina. La parte codificante va dal nucleotide 109 al nucleotide 717. Dobbiamo però non solo estrarre questa parte codificante, usande l'indicizzazione `[108:717]`, ma anche eliminare gli spazi separatori e la numerazione che sono stati aggiunti per una maggiore leggibilità. Possiamo fare ciò in poche righe, usando le espressioni regolari:

```
a='''cgttatttaa ggtgttacat ... tatacctttc gccttacaat      60
     gtaatttctt ttcacataaa ... aatttttaat gacttacgaa     120
     ttaccaaaat taccttatac ... attttgataa agaaacaatg     180
     gaaattcact atacaaagca ... aactaaatga agcagtctca     240
     ....
     gctcgttttg gttcaggatg ... atggtaaact agaaattgtt     540
     tccactgcta accaagattc ... ctccagttct tggcttagat     600
     gtttgggaac atgcttatta ... gtcctgaata cattgacaca     660
     ttttggaatg taattaactg ... ttgacgcagc aaaataatta     720
     tcgaaaggct cacttaggtg ggtcttttta tttcta'''

u=re.compile('(\s|\d)+');
a=u.sub('',a)[108:717] # La numerazione in Python inizia da 0.
print (a)
```

XIII. CLASSI

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 orientato 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. Questa seconda difficoltà può spesso essere superata tramite l'uso di *sottoclassi*.

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.

La programmazione orientata agli oggetti è spesso un utile principio di organizzazione. Negli algoritmi numerici (in cui alcune operazioni vengono eseguite moltissime volte) bisogna però tener conto del fatto che ogni uso di una classe corrisponde all'invocazione di una funzione talvolta complessa; ciò può rallentare l'esecuzione di un programma di un fattore 2-3.

La classe vettore

Come primo esempio definiamo una classe che rappresenta vettori in $\mathbb{R}^3$. Un vettore `v` possiede componenti `v.x`, `v.y`, `v.z`. Le operazioni possibili per due vettori `v` e `w` e numeri `x`, `y`, `z`, `t` sono le seguenti:

(i) Inizializzazione `v=vettore(x,y,z)`.
(ii) Addizione `v+w`.
(iii) Moltiplicazione a sinistra o a destra con `t`: `t*v` risp. `v*t`.
(iv) Prodotto scalare `v**w`.
(v) Prodotto vettoriale `v.pv(w)`.
(vi) Lista dei coefficienti `v.coeff()`.

Possiamo realizzare questa classe nel modo seguente:

```
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 __pow__ (A,B): return A.x*B.x+A.y*B.y+A.z*B.z
  def __rmul__(A,t): return A*t
  def coeff(A): return [A.x,A.y,A.z]
  def pv (A,B):
    (x,y,z)=A.coeff(); (u,v,w)=B.coeff()
    return vettore(y*w-z*v,z*u-x*w,x*v-y*u)
```

In ogni *metodo* (cioè ogni operazione) 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 sette metodi definiti per la classe `vettore`.

---

(1) Quando una classe contiene un metodo `__init__` (detto *costruttore* della classe), questo viene automaticamente invocato quando un oggetto della classe viene creato con un'istruzione che nel nostro caso avrà la forma `vettore(x,y,z)`.

Sono possibili non solo assegnazioni della forma `v=vettore(x,y,z)` ma, come si vede nelle definizioni di `__add__`, `__mul__` e `pv`, 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.

---

(2) 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, nell'invocazione, il prefisso a cui il metodo con i suoi 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)
```

---

(3) In modo simile il metodo `__mul__` della nostra classe `vettore` permette l'uso dell'operatore `*`. Nel nostro 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]
```

---

(4) In verità in matematica si preferisce porre lo scalare prima del vettore, mentre nella definizione del metodo `__mul__` 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. È così definito anche `4*a`. In modo simile sono definiti i metodi `__radd__`, `__rdiv__`, `__rmod__`, `__rsub__`, `__rpow__`. Metodi analoghi esistono per gli operatori logici, ad esempio `__rand__`.

---

(5) 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 in `v.coeff()`.

---

(6) Il prodotto scalare si ottiene con il metodo `__pow__` della classe (che corrisponde all'operatore `**`), il prodotto vettoriale con `pv`. Ricordiamo che

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \times \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} yw - zv \\ zu - xw \\ xv - yu \end{pmatrix}$$

```
a=vettore(2,3,5); b=vettore(1,7,2)

print (a**b)
# 33.0

print (a.pv(b).coeff())
# [-29.0, 1.0, 11.0]
```

Operatori sovraccaricati

Gli operatori unari e binari matematici e logici possono essere sovraccaricati mediante metodi predefiniti nella sintassi, ma liberi nella definizione delle operazioni effettuate. Elenchiamo le più importanti di queste corrispondenze:

`__add__`	+ binario
`__sub__`	− binario
`__mul__`	*
`__div__`	/
`__floordiv__`	//
`__mod__`	%
`__pow__`	**
`__pos__`	+ unario
`__neg__`	− unario
`__lshift__`	<<
`__rshift__`	>>
`__and__`	&
`__or__`	|
`__xor__`	^
`__invert__`	~

Con `+`, `*` ecc. vengono anche sovraccaricati `+=`, `*=` ecc.:

```
a=vettore(2,3,5)
b=vettore(1,7,2)

a+=b
print (a.coeff())
# [3.0, 10.0, 7.0]

a*=2
print (a.coeff())
# [6.0, 20.0, 14.0]
```

Gli operatori ridefiniti (o *sovraccaricati*) rispettano 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]
```

Mentre gli operatori 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 necessario che questi metodi restituiscano un oggetto della classe (infatti nell'ultimo esempio entrambi i metodi visualizzano una stringa sullo schermo e restituiscono `None`).

Istruzioni indipendenti in una classe

Se la definizione di una classe contiene istruzioni indipendenti, queste vengono direttamente eseguite - prima dell'utilizzo della classe!

```
class prova: print (7)
# 7
```

In genere la presenza di istruzioni indipendenti può rendere meno trasparente il programma. Può essere comunque utile definire valori iniziali di alcuni componenti della 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 considerati componenti degli oggetti della classe.

Il metodo speciale `__str__`

Il metodo speciale `__str__` può essere usato per ridefinire il comportamento dell'istruzione `print` quando viene applicato agli oggetti di una classe. Esso dovrebbe restituire una stringa che viene poi visualizzata da `print`. Per la classe `vettore` possiamo definire

```
def __str__ (A): return '%.2f %.2f %.2f' %tuple(A.coeff())
```

Adesso è sufficiente

```
a=vettore(2,3,5); b=vettore(1,7,2)

print (a.pv(b))
# -29.00 1.00 11.00
```

Metodi impliciti

Siccome la natura degli argomenti di una funzione in Python non deve essere nota nel momento della definizione della funzione, possiamo definire funzioni che utilizzano componenti di una classe senza che questa appaia esplicitamente nella funzione.

Possiamo ad esempio creare una funzione che calcola la lunghezza di un vettore, semplicemente utilizzando i suoi coefficienti:

```
def lunghezza (v):
  x=v.x; y=v.y; z=v.z; return math.sqrt(x*x+y*y+z*z)

a=vettore(3,2,6)

print (lunghezza(a))
# 7.0
```

La funzione `lunghezza` è definita al di fuori della classe e formalmente non esiste alcun legame tra la funzione e la classe. Infatti questa stessa funzione può essere usata per ogni altra classe per la quale sono previste componenti numeriche `.x`, `.y` e `.z`.

Nel caso concreto comunque si preferirà definire un metodo della classe per calcolare la lunghezza:

```
  def lun (A): return math.sqrt(A**A)

a=vettore(3,2,6)

print (a.lun())
# 7.0
```

Il metodo speciale __call__

Un significato speciale ha anche il metodo `__call__`. Quando definito per una classe, esso permette di usare gli oggetti della classe sintatticamente 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`:

```
v=vettore(3,5,7)
print (v)
# 3.00 5.00 7.00
v(8,2,1)
print (v)
# 8.00 2.00 1.00
```

Le stesse istruzioni usate in `__call__` appaiono anche in `__init__`. È preferibile usarle una volta sola, e quindi riscriveremo il metodo `__init__` nel modo seguente:

```
def __init__ (A,x,y,z): A(x,y,z)
```

Anche nel caso di `__call__` naturalmente si è completamente liberi nella scelta delle operazioni che il metodo deve eseguire. In una libreria grafica potremmo ad esempio usare sistematicamente i metodi `__call__` per l'esecuzione di operazioni di disegno oppure anche solo per preparare la struttura geometrica di una figura, riservando ad esempio l'operatore + per effettuare il disegno:

```
class cerchio:
  def __call__ (A,parametri):
  ...
  def __pos__ (A): # Corrisponde a +.
    disegna(A)
  ...

a=cerchio()
a(x,y,r); +a
a(u,v,s); +a
```

Sottoclassi

Dopo `class animale: ...` si può definire una sottoclasse con

```
class mammifero (animale): ...
```

In principio la sottoclasse eredita tutti i componenti della classe superiore; questi possono essere successivamente ridefiniti, così come possono essere aggiunti nuovi metodi.

Un caso molto importante è la definizione di sottoclassi della classe `list`. Definiamo ad esempio una nuova classe `Vettore` che rappresenta ancora vettori di $\mathbb{R}^3$, ma utilizzando una sottoclasse di `list`. Studiare attentamente le differenze con la vecchia classe `vettore`:

```
class Vettore (list):
  def __init__(A,x,y,z): A(x,y,z)
  def __call__(A,x,y,z): del A[:]; A.extend([x,y,z])
  def __add__(A,B):
    (x,y,z)=A; (u,v,w)=B; return Vettore(x+u,y+v,z+w)
  def __mul__(A,t): (x,y,z)=A; return Vettore(t*x,t*y,t*z)
  def __rmul__(A,t): return A*t
  def __pow__ (A,B): # prodotto scalare.
    (x,y,z)=A; (u,v,w)=B; return x*u+y*v+z*w
  def pv (A,B): # prodotto vettoriale.
    (x,y,z)=A; (u,v,w)=B
    return Vettore(y*w-z*v,z*u-x*w,x*v-y*u)
  def __str__ (A): return '%.2f %.2f %.2f' %(tuple(A))
  def lun (A): (x,y,z)=A; return math.sqrt(A**A)
```

Non appare più il metodo `coeff`. Si noti l'uso di `del` e `extend` in `__call__`.

```
v=Vettore(3,4,5); w=Vettore(1,3,2)
print (v)
# 3.00 4.00 5.00

print (v+w)
# 4.00 7.00 7.00

print (v.pv(w))
# -7.00 -1.00 5.00

print (v**w) # 25
```

Metodi di confronto

Anche i metodi di confronto possono essere sovraccaricati:

`__eq__`	==
`__ne__`	!=
`__le__`	<=
`__lt__`	<
`__ge__`	>=
`__gt__`	>

Gli operatori devono essere con la stessa sintassi come gli operatori originali, mentre l'interpretazione è di nuovo completamente libera - non devono nemmeno restituire un valore booleano.

---

type e isinstance

Ogni oggetto di Python possiede un tipo unico che si ottiene tramite la funzione `type`. Con `isinstance` si può verificare se un oggetto appartiene a una classe.

```
print (type(77))
# <class 'int'>
print (isinstance(77,int))
# True

print (type([3,5,1]))
# <class 'list'>
print (isinstance([3,5,1],tuple))
# False
print (isinstance([3,5,1],(tuple,list)))
# True

print (type('alfa'))
# <class 'str'>

def f(x): pass
print (type(f))
# <class 'function'>
```

---

dir e attributi standard

Con `dir(x)` si ottengono i nomi definiti per `x`; `x` può essere una classe oppure un oggetto di una classe. Provare le seguenti istruzioni:

```
print (dir(list))
print (dir(10))
print (dir('alfa'))
print (dir(Vettore)); print (dir(vettore))
```

oppure meglio ad esempio

```
for x in dir(10): print(x)
```

`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).

XIV. EPILOGO

I linguaggi più popolari

Sul sito *tiobe.com* viene pubblicato ogni mese un indice di popolarità dei linguaggi di programmazione. Vediamo che il Python occupa uno dei primi posti in assoluto e il primo tra i quattro più noti linguaggi ad altissimo livello, essendo più popolare, in questa classifica, di Perl, Ruby e Lisp insieme. Farebbe parte di questo gruppo anche R, che si trova al 26-esimo posto (non riportato) e poco noto al di fuori della statistica.

Java e C++ sono molto utilizzati nell'informatica professionale, ma non sono linguaggi ad alto livello e quindi non particolarmente adatti alla programmazione scientifica. Questi linguaggi sono anche estremamente complessi. Lo stesso vale per Objective-C, che attualmente viene spinto fortemente dalla Apple, ma è complicato e educa a una programmazione semiautomatica poco autonoma e poco formativa. Un discorso simile vale per C# e Visual Basic in ambito Microsoft.

PHP e JavaScript sono linguaggi specializzati per la creazione di pagine web dinamiche e quindi si prestano poco allo sviluppo di software di carattere generale o scientifico-matematico.

Il C è invece forse il linguaggio più semplice, molto versatile ed efficiente, e si trova giustamente nelle prime due posizioni. Va notato che Objective-C è una vera estensione del C, per cui si può programmare in C puro anche nell'ambiente Xcode della Apple.

Position Jan 2011	Position Jan 2010	Delta in Position	Programming Language	Ratings Jan 2011
1	1	=	Java	17.773%
2	2	=	C	15.822%
3	4	↑	C++	8.783%
4	3	↓	PHP	7.835%
5	7	↑↑	Python	6.265%
6	6	=	C#	6.226%
7	5	↓↓	(Visual) Basic	5.867%
8	12	↑↑↑↑	Objective-C	3.011%
9	8	↓	Perl	2.857%
10	10	=	Ruby	1.784%
11	9	↓↓	JavaScript	1.589%
12	11	↓	Delphi	1.287%
13	18	↑↑↑↑↑	Lisp	1.109%
14	17	↑↑↑	Pascal	0.919%
15	-	↑↑↑↑↑↑↑↑↑↑	Assembly	0.864%
16	14	↓↓	SAS	0.771%
17	30	↑↑↑↑↑↑↑↑↑↑	Transact-SQL	0.758%
18	33	↑↑↑↑↑↑↑↑↑↑	RPG (OS/400)	0.717%
19	20	↑	MATLAB	0.706%
20	28	↑↑↑↑↑↑↑↑	Ada	0.679%

Mancano nella classifica i due più completi linguaggi matematici (Mathematica e Maple, entrambi commerciali e piuttosto costosi) e un altro linguaggio matematico, il Macaulay 2, gratuito e specializzato per compiti di algebra commutativa e geometria algebrica, un linguaggio ad altissimo livello che può essere anche utilizzato nella programmazione di ogni giorno. Doveva essere il linguaggio trattato nel corso, se non che si è scoperto che è difficile utilizzarlo sotto Windows.

Purtroppo l'avvento dei computer grafici e la sempre più marcata trasformazione del personal computer in un apparecchio multimediale di cui le stesse ditte produttrici sembra che vogliano ostacolare l'uso autonomo e produttivo hanno frenato visibilmente lo sviluppo dei linguaggi di programmazione, rendendoli sempre meno trasparenti e sempre più sovraccaricati di caratteristiche complicate ed effimere.

Il Python linguaggio dell'anno

Ancora dal sito della TIOBE:

Python wins the TIOBE Programming Language Award of 2010!

Programming language Python has become programming language of 2010. This award is given to the programming language that gained most market share in 2010. Python grew 1.81% since January 2010. This is a bit more than runner up Objective-C (+1.63%). Objective-C was favorite for the title for a long time thanks to the popularity of Apple's iPhone and iPad platforms. However, it lost too much popularity the last couples of months.

Python has become the de facto standard in system scripting (being a successor of Perl in this), but it is used for much more different types of application areas nowadays. Python is for instance very popular among web developers, especially in combination with the Django framework. Since Python is easy to learn, more and more universities are using Python to teach programming languages.

Un numero enigmatico

In un raccolta di enigmi di 150 anni fa si trova il seguente compito (per $n = 20$ ed $m = 7$): Siano date n caselle (con $n \in \mathbb{N} + 2$). Sia anche $m \in \mathbb{N} + 2$. Iniziando a contare dalla prima, cancelliamo la casella m-esima, poi, partendo dalla casella successiva a quella, di nuova l'm-esima tra quelle rimaste, procedendo in maniera ciclica quando necessario e passando eventualmente più volte sopra la stessa casella. Ripetiamo questa operazione fino quando rimane una sola casella. Qual'è questa casella?

Non so se esiste una formula per calcolare il numero della casella rimasta. Il compito si presta comunque per un esercizio in Python. Denotiamo il numero cercato con $L(n, m)$ e iniziamo, come d'uso in Python e in matematica, a contare da 0.

Esaminare attentamente il programma:

```
def L(n,m):
  a=list(range(n)); p=0
  while n>1: p=(m+p-1)%n; del a[p]; n-=1
  return a[0]

print('    ',end='')
for m in range(2,13): print('%2d' %m,end=' ')
print('\n','-'*37,sep='')
for n in range(2,30):
  print ('%2d |' %n,end='')
  for m in range(2,13): print('%2d' %L(n,m),end=' ')
  print('')
```

Si ottiene la seguente tabella:

```
     2  3  4  5  6  7  8  9 10 11 12
-------------------------------------
 2 | 0  1  0  1  0  1  0  1  0  1  0
 3 | 2  1  1  0  0  2  2  1  1  0  0
 4 | 0  0  1  1  2  1  2  2  3  3  0
 5 | 2  3  0  1  3  3  0  1  3  4  2
 6 | 4  0  4  0  3  4  2  4  1  3  2
 7 | 6  3  1  5  2  4  3  6  4  0  0
 8 | 0  6  5  2  0  3  3  7  6  3  4
 9 | 2  0  0  7  6  1  2  7  7  5  7
10 | 4  3  4  2  2  8  0  6  7  6  9
11 | 6  6  8  7  8  4  8  4  6  6 10
12 | 8  9  0  0  2 11  4  1  4  5 10
13 |10 12  4  5  8  5 12 10  1  3  9
14 |12  1  8 10  0 12  6  5 11  0  7
15 |14  4 12  0  6  4 14 14  6 11  4
16 | 0  7  0  5 12 11  6  7  0  6  0
17 | 2 10  4 10  1  1 14 16 10  0 12
18 | 4 13  8 15  7  8  4  7  2 11  6
19 | 6 16 12  1 13 15 12 16 12  3 18
20 | 8 19 16  6 19  2  0  5  2 14 10
21 |10  1 20 11  4  9  8 14 12  4  1
22 |12  4  2 16 10 16 16  1  0 15 13
23 |14  7  6 21 16  0  1 10 10  3  2
24 |16 10 10  2 22  7  9 19 20 14 14
```