2004/08

Stefano Caroselli Patrizia Mentrasti

INTRODUZIONE

AL LINGUAGGIO

INDICE

PREMESSA

Ci proponiamo in questi appunti di presentare al lettore il linguaggio di programmazione Python in maniera semplice e con qualche paragone con il C.

Il Python è un potente linguaggio di programmazione, semplice da apprendere, con efficienti strutture dati di alto livello e che consente anche un facile approccio alla programmazione orientata ad oggetti.

Questo manuale approfondisce [3], e pertanto vengono saltate le descrizioni di alcuni tipi di dato e delle assegnazioni semplici; liste e dizionari vengono invece ripresentati.

La maggior parte quanto qui contenuto è reperito nei seguenti siti internet:
sito ufficiale sul python: http://www.python.org/
sito in italiano sul python: http://www.python.it/
in particolare in [1] e [2].

Si veda anche [4] per ulteriori approfondimenti.

1. INTRODUZIONE

1.1. Linguaggi "tradizionali" e linguaggi di script

Esistono moltissimi linguaggi di programmazione ad alto livello, ognuno con particolari caratteristiche, vantaggi e svantaggi. Rispetto a come vengono tradotti, possiamo classificarli in:

- Linguaggi compilati
- Linguaggi interpretati
- Linguaggi pseudo-compilati

In genere quando si parla di linguaggio si fa riferimento ad un compilatore, ossia ad un programma che riceve in input un file di testo con sintassi e dopo averlo tradotto in linguaggio macchina, o in linguaggio a livello più basso, ne esegue le istruzioni. Per questo possiamo considerare i linguaggi tradotti per mezzo di un compilatore, definiti linguaggi **compilati**, come "tradizionali".

Un importante linguaggio compilato è il C : nel C ogni singolo file sorgente (estensione .c) viene compilato separatamente e, se non vengono segnalati errori, viene generato un file oggetto (estensione .o). A questo punto viene eseguito un linking, il cui scopo è unificare i vari moduli-oggetto e trasformare i riferimenti tra i files da relativi ad assoluti, generando un file eseguibile (estensione .exe). Se in fase di compilazione di un sorgente vengono segnalati errori il processo si interrompe con una segnalazione di errore.

I linguaggi **interpretati** al contrario non devono essere compilati e linkati, ma hanno bisogno di essere interpretati da un programma apposito, l'interprete.

Ad esempio il Basic è un linguaggio interpretato: i codici sorgente preparati in uno script vengono interpretati *riga per riga*. A differenza di quanto accade per i linguaggi compilati, quindi, non si perde tempo a compilare un programma ma viene direttamente eseguito. Come svantaggio però si ha che può aumentare di molto il tempo di esecuzione di un programma. Inoltre gli errori possono essere segnalati quando il programma è in parte eseguito.

Infine ci sono i linguaggi **pseudo-compilati**, una via di mezzo tra i primi due. Infatti essi vengono compilati senza produrre file oggetto: il codice pseudo-compilato rimane nella CPU del computer, e viene automaticamente linkato.

Questi linguaggi, come ad esempio il Perl, sono ottimali per script di piccole dimensioni, ma possono creare problemi con script complessi (la CPU verrebbe sovraccaricata di dati).
Il Python rientra in quest'ultima categoria.

1.2. Il Python: caratteristiche principali

Possiamo elencare alcuni principali caratteristiche:

- L'interprete del Python può essere usato in modo interattivo con un proprio prompt di comandi (richiamato con IDLE in Windows e Python2 in Linux).
- Eredita molti aspetti del C, quali alcuni aspetti semantici, o la costruzione di alcuni tipi di dato.
- È capace di interagire con altri linguaggi, in particolare con il C e il C++.
- La sua grammatica lavora anche in uno script salvato con estensione .py, avvertendo ad ogni linea l'utente di eventuali errori.
- Essendo un linguaggio pseudo-compilato, avvia una pseudo-compilazione prima di interpretare uno script, ma l'oggetto viene memorizzato in un file fisico (estesione .pyc). Per questo aspetto è simile al linguaggio Java.
- Il codice è molto più chiaro rispetto ad altri linguaggi per l'assenza di dichiarazioni, e per l'utilizzo dell'indentazione per raggruppare le istruzioni in blocchi (nel C ad esempio erano necessarie le graffe { }).
- È estensibile, ossia può essere modificato da chiunque, migliorandone la qualità.

1.3. Come installare Python

I file d'installazione sono reperibili sul sito:

http://www.python.org/download.

Le spiegazioni e gli esempi presenti in questo documento si basano sulla versione 2.3.3 del software per Windows, e sono stati testati anche dal promt di Dos (in Windows) e in ambiente Linux.

- Sotto Dos / Windows: una volta scaricato il file `Python-2.3.3.exe`, è sufficiente mandarlo in esecuzione (doppio click sotto Windows) per avviare l'installazione, e seguire poi le istruzioni che seguono.
- Sotto Linux: molte distribuzioni di Linux hanno Python già installato. Se comunque si volesse aggiornare la propria versione si deve scaricare l'archivio `Python-2.3.3.tgz` e seguire questi passi:
  i. estraete i file dell'archivio in una directory.
  ii. aprite la shell e spostatevi nella directory appena creata.
  iii. digitate `make install`.

  Per ogni eventuale problema è presente nell'archivio anche un file di guida per l'installazione.

1.4. Il prompt dei comandi

Una volta installato nel proprio PC, l'ambiente Python interattivo può essere richiamato:

- Sotto Dos: andando nella directory di installazione del programma (generalmente è `C:\Python**\` dove `**` è la versione del software) e digitando `python`.

- Sotto Windows: facendo doppio click sul file python.exe presente nella directory di installazione del programma (è possibile creare un collegamento sul Desktop).

- Sotto Linux: digitando `python` dalla shell.

Dopo una breve descrizione del software installato, viene visualizzato il **prompt primario** (>>>) e il programma resta in attesa di un input dell'utente.

Possiamo ora eseguire calcoli, definire funzioni, importare file, ecc.

Il **prompt secondario**[1] (`...`) viene visualizzato ogni qual volta si batte [INVIO] non avendo completato un'istruzione, o nel caso in cui tale istruzione si divide in più passi (ad es. durante la definizione di una funzione).

È possibile inserire dei commenti, ossia frasi d'aiuto che vengono ignorate dall'interprete, ma sono utili per descrivere il listato: un # "nasconde" all'interprete tutti i commenti scritti dopo di esso fino a fine riga[2].

Oltre all'ambiente interattivo, è possibile far eseguire direttamente un file sorgente, ossia uno script, con estensione .py, contenente proprie istruzioni, tramite il comando:

```
./Python nomefile.py
```

dove nomefile è il nome del file che si vuole eseguire. Tale possibilità viene esaminata nel par. 6, dove si descrive la creazione e l'uso dei moduli.

Per terminare il programma è sufficiente digitare un comando di EOF, che ad esempio in Linux è [CTRL-D].

[1] In alcuni programmi recenti che funzionano sotto Windows è stato tolto il prompt secondario, e in questi casi la riga inizia senza alcun simbolo.

[2] Qui i commenti vengono scritti con un carattere più piccolo anche se per l'interprete non c'è distinzione di dimensioni.

2. PER COMINCIARE

2.1. Un primo esempio

Consideriamo le seguenti istruzioni[3], che danno come risultato la stampa dei primi termini della serie di Fibonacci.

Ricordiamo che tale serie è caratterizzata dalla proprietà che i primi due suoi termini sono 0 e 1, e ogni termine successivo è uguale alla somma dei due termini ad esso precedenti.

Utilizziamo l'istruzione di controllo `while`.

```
>>>  a, b = 0, 1
>>>  while b<15:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
13
```

Osservazioni:

- Le variabili `a` e `b` non sono state dichiarate prima dell'assegnazione. In generale non è necessario dichiarare le variabili, in quanto tale operazione avviene automaticamente con l'assegnazione di un valore.
- Notiamo che è in un'unica riga sono state fatte due assegnazioni. Nel Python infatti è possibile fare assegnazioni multiple, come spiegheremo più avanti.
- Il ciclo **`while`** si comporta come nel C: continua ad essere eseguito fintantoché la condizione di controllo `b<10` è vera; tale condizione ha come risultato un valore booleano (0 = falso, 1 = vero)[4].
- A differenza del C non servono le parentesi per individuare la condizione di controllo. Essa inizia dopo la parola chiave `while` e termina con i due punti. Eventuali parentesi servono per costruire condizioni più elaborate.
- Il corpo dell'istruzione `while` è indentato rispetto al resto del programma, e per tutta la durata di questa istruzione il prompt dei comandi è quello secondario (`...`) ad indicare che deve essere terminata l'ultima istruzione inserita. Per far capire all'interprete dove finisca il ciclo è stato necessario lasciare una riga vuota (battendo due volte il tasto [INVIO] dopo l'ultima istruzione)

[3] In questo e nei successivi esempi, le righe in cui non son presenti né il prompt primario, né quello secondario sono da considerarsi come output prodotto dall'interprete.

[4] In generale la condizione di controllo può essere una qualsiasi espressione. Infatti, come sarà spiegato in seguito, da ogni espressione si può ricavare un valore booleano.

- La funzione **`print()`** stampa su schermo i valori dei suoi argomenti. Possono essere stampate anche stringhe, ed esse verranno stampate senza apici.

Ecco alcuni esempi di come si comporta la funzione `print`:

```
>>>  x = 10**3
>>>  y = 3.14
>>>  print 'x vale', x,'; pi greco =', y
x vale 1000 ; pi greco = 3.14
```

Ogni argomento viene stampato con uno spazio di distanza dal precedente.

Dopo aver stampato, l'interprete passa automaticamente a riga nuova. Se si vuole che la stampa avvenga sulla stessa riga, è necessario aggiungere una virgola anche dopo l'ultimo argomento (ogni valore viene stampato con uno spazio di distanza dal precedente).

```
>>>  a, b = 0, 1
>>>  while b<15:
...       print b,
...       a, b = b, a+b
...
1 1 2 3 5 8 13
```

Per ricevere invece il testo digitato da tastiera si utilizza la funzione **`input()`**: come argomento si può inserire una stringa che viene stampata al momento di ricevere il testo.

```
>>>  x = input("inserisci un numero ")
inserisci un numero
```

L'interprete resta in attesa di ricevere un valore, che sarà poi assegnato alla variabile x.

Un'altra funzione importante è **`type()`**: essa restituisce il tipo di valore del suo argomento.

```
>>>  type("matematica")
<type 'str'>
```

2.2. Il ruolo dell'indentazione

L'indentazione è fondamentale nel Python, al contrario di quanto avviene in altri linguaggi come il C, dove ha spesso solamente una funzione estetica.

Infatti un blocco di programma non possiede un'istruzione che ne indichi l'inizio e una che ne indichi la fine, ma l'intera sezione di codice facente parte di un blocco è caratterizzata da un "rientro sinistro" maggiore rispetto al resto del codice.

Spesso avviene che un'istruzione complessa, come la definizione di una funzione, abbia al suo interno altre istruzioni complesse, come un ciclo *for*.

In questo caso il "rientro sinistro" del ciclo è maggiore di quello della funzione, che a sua volta è maggiore di quello del resto del programma.

```
PROGRAMMA                    → inizio programma)
————————————
————————————
DEF.  FUNZIONE               → inizio funzione
      ————————————
      ————————————
      CICLO  FOR             → inizio ciclo
            ——————————
            ——————————       → fine ciclo
      ————————————
      ————————————
      ————————————
      CICLO WHILE            → inizio nuovo ciclo
            ——————————       → fine nuovo ciclo
      ————————————
      ————————————           → fine funzione
————————————
————————————                 → fine programma
```

2.3. Le stringhe

Una **stringa** è una sequenza finita di lettere[5] racchiuse tra singoli (o doppi) apici; ogni lettera è considerata quindi come una stringa di un solo elemento.

```
>>>  'x'
'x'
>>>  "Ciao"
'Ciao'
```

È possibile sommare due stringhe, o moltiplicare una stringa per uno scalare:

```
>>>  'Hello' + 'World'
'HelloWorld'
>>>  'Questa ' + "e' " + 'una ' 'frase'
"Questa e' una frase"
>>>  'Questa e\' una frase'
"Questa e' una frase"
>>>  "Ciao" * 3
'CiaoCiaoCiao'
```

[5] In genere per *lettera* si intende un simbolo appartenete ad un determinato alfabeto. Qui useremo questa parola per indicare semplicemente un generico carattere stampabile con la tastiera.

Osservazioni:

- Nel sommare due stringhe è facoltativo l'utilizzo dell'operatore +
- Per evitare errori, conviene utilizzare i doppi apici se nella frase è presente un apostrofo, e viceversa è necessario utilizzare gli apici singoli qualora nella frase siano presenti doppi apici.
- In alternativa, è possibile anteporre al simbolo che può creare problemi un \ in modo tale che l'interprete lo consideri semplicemente parte del testo.

La funzione **`raw_input()`** si comporta come la funzione `input`, ma può ricevere qualsiasi carattere (non solo valori); tali simboli vengono unificati in una stringa.

Altre funzioni che manipolano le stringhe saranno descritte nel paragrafo riguardante le liste.

2.4. Assegnazione di variabili

L'**assegnazione** avviene per mezzo dell'operatore = .

Le variabili *non hanno bisogno di essere dichiarate*. La loro dichiarazione avviene automaticamente con la loro prima assegnazione.

```
>>>  a = 3
>>>  b = 5*(6**2)
```

Nel caso si abbia un'istruzione del tipo `variabile1 = variabile2` , l'interprete estrae il valore assegnato alla `variabile2` e lo assegna anche alla `variabile1` .

```
>>>  a = 12
>>>  b = 23
>>>  a = b*2
>>>  a
46
>>>  b
23
```

È possibile fare assegnazioni più complesse o multiple[6] e di tipo diverso:

```
>>>  x = y = 2
>>>  a, b = 25, 'dicembre'
>>>  a
25
>>>  b
'dicembre'
```

[6] Per comprendere meglio le assegnazioni multiple si veda anche il paragrafo sulle *tuple*.

2.5. Variabili booleane

Una precisazione va fatta nei confronti delle variabili **booleane**.

I valori logici sono `True` (vero) e `False` (falso)[7]. Tutti i valori in Python possono essere considerati valori booleani e come tali possono formare espressioni logiche.

In genere l'interprete associa a valori "vuoti" un valore Falso. Ad esempio saranno considerati falsi valori come `0,(),[],"",None`.

Tutti gli altri valori saranno considerati veri.

Anche gli operatori logici `and` e `or` possono essere applicati ad ogni valore, ma il loro comportamento è leggermente diverso. Vediamo due esempi.

```
>>>  c = a and b
```

l'interprete si comporta nel seguente modo:

- se a è vera, allora c vale b
- se a è falsa, allora c vale a

usando una sintassi del C, tale assegnazione corrisponde all'istruzione:

```
c = (a) ? b : a
```

```
>>>  c = a or b
```

al contrario in questo caso l'interprete si comporta così:

- se a è vera, allora c vale a
- se a è falsa, allora c vale b

usando la sintassi precedente, tale assegnazione corrisponde all'istruzione C:

```
c = (a) ? a : b
```

Osserviamo che questi operatori coincidono con i normali operatori logici quando vengono applicato ai valori True e False, mentre in altre situazioni possono non dare risultati booleani.

```
>>>  a = True
>>>  b = not a
>>>  a or b
True
>>>  a and b
False
>>>  3 or 0
3
>>>  "ciao" and "mondo"
'mondo'
```

La funzione predefinita (cioè "built in") **`bool()`** restituisce il valore logico del suo argomento.

[7] Attenzione alle maiuscole: `True` e `False` sono riconosciuti dall'interprete, mentre `true, TRUE, false, FALSE` no.

3. STRUTTURE DATI

3.1. Liste

Una **lista** è un tipo di variabile rappresentante una sequenza di valori, ognuno identificato in modo univoco da una chiave numerica; i valori memorizzati in una lista possono anche essere di tipo diverso tra loro (intero, stringa, ecc.).

Le liste seguono le stesse regole delle stringhe per quanto riguarda le assegnazioni e le operazioni di somma e di prodotto per uno scalare. D'altronde la maggior parte delle cose che saranno dette di seguito riguardo alla liste si applica anche alle stringhe.

Una lista è definita per mezzo di parentesi quadre, al cui interno sono elencati i diversi valori, separati da virgole.

```
>>>  list = [ 2, 6, 'auto', False ]
```

Nell'esempio è stata creata la lista `list` contenente quattro elementi: due valori interi, una stringa e un valore booleano.

La **lista vuota** è la lista non contenente alcun elemento, ed è definita mediante due parentesi quadre: `[]` .

Gli elementi di una lista sono indicizzati secondo l'ordine in cui compaiono nella definizione. Il primo elemento ha indice 0, il secondo 1, il terzo 2, e così via (lo stesso accade per le stringhe). Per esempio se volessimo accedere al valore 6 dovremmo scrivere `list[1]`. È possibile richiamare gli elementi anche con un'indicizzazione negativa, come mostrato di seguito.

Inserire un indice a cui non è assegnato nessun valore genera un errore.

Possiamo estrarre una **sottosequenza** di una lista, indicando l'indice iniziale (che è compreso) e l'indice finale (che è escluso), separati da due punti.

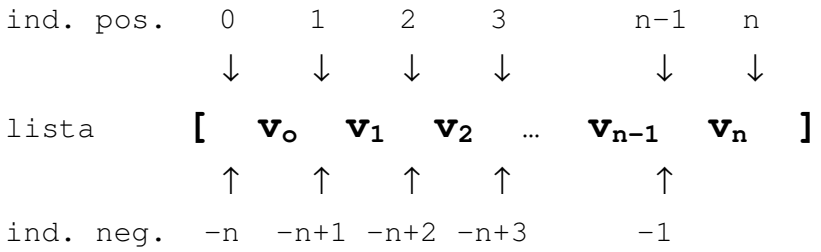
Se uno dei due indici non è espresso, viene sostituito con l'estremo (iniziale o finale, sempre compresi) della lista.

Osserviamo che nel richiamare una sottosequenza (e quindi usando i due punti) è possibile inserire qualsiasi indice, anche fuori dalla dimensione della lista, *senza generare alcun errore*. In queste situazioni l'interprete segue regole precise per riconoscere la sottosequenza.

Consideriamo ad esempio una generica lista `lis = [`v_0 `, ... ,` v_{n-1}`]` e supponiamo di richiamare `lis[i,j]`.

i. Se i (oppure j) ≥ n–1, viene considerato i (oppure j) = n.
ii. Se i (oppure j) < – n, viene considerato i (oppure j) = 0.
iii. Se – n ≤ i (oppure j) < 0, viene considerato i = n + i (oppure j = n + j)
iv. Se i ≥ j e non si verifica la (3) viene richiamata la lista vuota.

Da (*iii*) si ottiene la seguente identificazione degli indici:



Vediamo qualche esempio, che possa chiarire quanto appena spiegato:

```
>>>  a=['pasta', 'carne', 'pesce', 'frutta', 'pane']
# indice:   0        1          2          3          4       5
# in questo esempio n = 5
>>>  a[2]
'pesce'
>>>  print a[4]
pane
>>>  a[5]

Traceback (most recent call last):
  File "<pyshell#38>", line 1, in -toplevel-
    a[5]
IndexError: list index out of range
>>>  a[-1]
'pane'
>>>  a[4]='verdura'
>>>  a[1:3]
['carne', 'pesce']
>>>  a[3:1]
[]
>>>  a[1:99]
['carne', 'pesce', 'frutta', 'verdura']
>>>  a[1:]
['carne', 'pesce', 'frutta', 'verdura']
>>>  a[:]
['pasta', 'carne', 'pesce', 'frutta', 'verdura']
>>>  a[-99:99]
['pasta', 'carne', 'pesce', 'frutta', 'verdura']
>>>  a[-99:0]
[]
>>>  a[-3:0]         # è equivalente ad a[2:0]
[]
>>>  a[-3:-1]        # è equivalente ad a[2:4]
>>>  a[-5:3]         # è equivalente ad a[0:3]
['pasta', 'carne', 'pesce']
```

Osserviamo in particolare questi due casi:

```
>>>  a[-5:-1]        # è equivalente ad a[0:4]
['pasta', 'carne', 'pesce', 'frutta']
>>>  a[-5:-0]        # è equivalente ad a[0:0]
[]
```

Il primo è abbastanza ovvio: gli indici vengono *traslati* dopo lo zero; ma il secondo non tanto, in quanto l'istinto ci suggerirebbe che `a[-5:-0]` non sia altri che `a[0:5]` ossia l'intera lista. Ma questo è un *errore* in quanto `-0` coincide con `0`, non con 5.

Non vi è modo, usando due indici negativi, di richiamare l'ultimo elemento della lista.

Per ogni lista vale la seguente identità : `a = a[:] = a[:n]+a[n:]`

La funzione **`len()`** restituisce il numero di elementi di una lista.

La funzione **`del()`** elimina un elemento della lista.

3.2. Metodi per le liste

Esistono funzioni particolari, chiamate **metodi** che consentono di operare su un determinato oggetto. Tali metodi seguono una sintassi leggermente differente dalle normali funzioni, agendo come dei *campi* definiti all'interno dell'oggetto in questione.

Se volessimo applicare il metodo *method()* all'oggetto *object* dovremmo scrivere:

```
>>>  object.method()
```

Per quanto riguarda le liste, esistono diversi metodi. Ecco un parziale elenco:

`object.insert(i,x)`
: inserisce l'elemento x nella posizione i della lista `object`; gli elementi dalla posizione i in poi scalano di uno

`object.append(x)`
: inserisce l'elemento x alla fine della lista

`object.extend`(**L**)
: inserisce la nuova lista L alla fine della lista

`object.index(x)`
: restituisce la prima occorrenza di x nella lista

`object.remove(x)`
: elimina dalla lista il primo elemento x che compare

`object.sort()`
: effettua un ordinamento lessicografico degli elementi

`object.reverse()`
inverte gli elementi della lista (rispetto all'ordinamento precedente)

`object.count(x)`
restituisce il numero di occorrenze di x nella lista

`object.pop(i)`
restituisce l'elemento di indice i, togliendolo dalla lista; se non è specificato l'indice, si toglie l'ultimo elemento

3.3. Tuple

Le **tuple** sono un tipo di dati simili alle liste e alle stringhe, in quanto sono una sequenza di elementi a cui sono assegnati degli indici numerici. Ma al contrario di queste *sono immutabili*, non è possibile cioè modificarne il contenuto, né tantomeno inserire o estrarre elementi.

Una tupla è rappresentata mediante una coppia di parentesi tonde all'interno delle quali sono elencati gli elementi, separati da virgole. Gli elementi possono essere di tipo diverso: numeri, lettere, stringhe, liste, o altre tuple.

Al momento della definizione le parentesi tonde sono opzionali: è sufficiente elencare gli elementi separandoli da virgole. Sarà poi l'interprete a unificare i vari elementi in una tupla (*tuple packing*).

Osservazioni.

- Le parentesi sono obbligatorie se si vuole definire la **tupla vuota** `()`.
- Nel caso si voglia definire una tupla avente un solo elemento, questo deve essere seguito comunque da una virgola.

```
>>>  t = 'Do', 'Re', 'Mi', 'Fa', 'Sol'
>>>  t[0]
'Do'
>>>  t
('Do', 'Re', 'Mi', 'Fa', 'Sol')
>>>  u = t, (1, 2, 3, 4, 5)
>>>  u
(('Do', 'Re', 'Mi', 'Fa', 'Sol'), (1, 2, 3, 4, 5))
>>> a = ()
>>> b = 'Italia',     # la virgola indica che si tratta di una tuple
>>> len(a)
0
>>> len(b)
1
>>> b
('Italia',)
```

Data una tupla, è possibile assegnare contemporaneamente a più variabili distinte i valori di una tupla (*sequenze unpacking*)

```
>>>  x, y, z, j, k = t
>>>  x
'Do'
>>>  y
'Re'
>>>  z
'Mi'
>>>  j
'Fa'
>>>  k
'Sol'
```

Possiamo osservare che l'assegnazione multipla di variabili altro non è che la combinazione di un *tuple packing* e di un *sequenze unpacking*.

Infine citiamo due utili funzioni.

`list()`
: riceve in input una tupla e restituisce una lista con gli stessi valori.

`tuple()`
: riceve in input una lista e restituisce una tupla con gli stessi valori.

3.4. Dizionari

I **dizionari** sono strutture dati i cui elementi (**dati**) sono indicizzati mediante **chiavi** definite dall'utente. È possibile operare sugli elementi di un dizionario: possiamo modificare i vari dati, cancellarli, estrarne una parte, o inserirne di nuovi.

Un dizionario è definito con una coppia di parentesi graffe {} – rimanevano solo quelle… – al cui interno i dati sono elencati nella forma *chiave : valore*, e separati fra loro da virgole.

Le chiavi *devono* essere uniche all'interno del dizionario: un dizionario non può contenere due dati distinti aventi chiavi uguali. Il ruolo di chiave può essere assunto da qualunque dato di tipo immutabile (anche gli stessi interi).

```
>>>  diz = {chiave1 : valore1, chiave2:valore2, ...}
```

È possibile richiamare un determinato valore esistente tramite la chiave associata, allo stesso modo di quanto avviene nelle liste, oppure modificare un dato esistente, sostituendo il suo valore:

```
>>>  diz[chiave1]
valore1
>>>  diz[chiave1]=valore3
```

Esistono vari *metodi* per operare sui dizionari.

Eccone alcuni:

`object.keys()` restituisce la lista di tutte le chiavi del dizionario

`object.has_key(t)` controlla se la chiave t è presente

`object.values()` restituisce la lista di tutte i valori del dizionario

`object.sort()` effettua un ordinamento lessicografico delle chiavi

La funzione **`dict()`** restituisce un dizionario, avendo come input una lista di coppie (una tupla).

Ecco alcuni esempi riassuntivi:

```
>>> tel = {'andrea': 409857, 'giorgio': 413985}
>>> tel['giulia'] = 412733
>>> tel
{'giorgio':412785, 'andrea':413957, 'giulia':409833}
>>> tel['andrea']
409857
>>> del tel['giorgio']
>>> tel['simona'] = 412769
>>> tel
{'andrea':413957, 'giulia':409833, simona': 412769}

>>> tel.keys()
['andrea', 'giulia', 'simona']
>>> tel.has_key('giulia')
True

>>>  dict  (  [('fabrizio',  406275),  ('ugo',  421633),
('anna', 412634)] )
{ 'fabrizio':406275, 'ugo':421633, 'anna':412634 }
```

4. INIZIAMO A PROGRAMMARE

4.1. Strutture di controllo

Abbiamo visto nell'esempio iniziale la sintassi dell'istruzione `while`.
Studiamo ora le altre principali strutture di controllo.

- L'istruzione **`if`**.

La sintassi di questa istruzione è:

```
if [condizione1]:
     [istruzione1]
elif [condizione2]:
     [istruzione2]
elif [condizione3]:
     [istruzione3]
else:
     [condizione4]
```

Tutti gli `elif` sono opzionali, così come `l'else`.
La parola `elif` sta a significare `else if` e serve a concatenare più condizioni, ottenendo quello che in C si aveva con l'istruzione `switch ... case`. *I due punti dopo le condizioni sono obbligatori.*

```
>>>  x = int(raw_input("Inserire un intero: "))
>>>  if x < 0:
...       print 'Numero negativo'
...  elif x == 0:
...       print 'Zero'
...  else:  print 'Numero positivo'
...
```

Le istruzioni devono essere indentate, per far capire al programma che non sono parti di codice indipendente. Nel caso di un istruzione composta da una sola riga, è possibile scriverla dopo i due punti della relativa condizione.

- L'istruzione **`for`**.

L'istruzione for del Python è leggermente diversa da quella del C: l'iterazione avviene sugli elementi di un qualsiasi sequenza, nell'ordine in cui compaiono nella sequenza.
La sintassi è la seguente:

```
for [elemento i-mo] in [sequenza] :
     [istruzioni]
```

Anche in questo caso i due punti sono obbligatori.

Vediamo un esempio:

```
>>> # Lunghezza di alcune stringhe:
... a = ['giallo', 'rosso', 'blu']
>>> for x in a:
...     print x, len(x)
...
giallo 6
rosso 5
blu 3
```

Nel caso si voglia utilizzare il `for` in modo "classico", abbiamo bisogno di una lista di interi. Ad esempio per ottenere: *per i che va da 0 a 3* dovremmo scrivere:

```
for i in [0, 1, 2, 3]:
```

e questo può creare problemi nel caso i numeri siano molti di più.

Per risolvere questo problema esiste una utilissima funzione, **`range`**`()`, che genera una lista contenente una progressione aritmetica.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Se alla funzione vengono passati due interi m, n come argomento, viene generata una lista da m fino a n–1. Se alla funzione vengono passati tre interi m, n, p come argomento, viene generata una lista da m fino a n–1, con un incremento p.

```
>>> range(12, 16)
[12, 13, 14, 15]
>>> range(0, 40, 8)
[0, 8, 16, 24, 32]
>>> range(-100, -300, -50)
[-100, -150, -200, -250]
```

Ora è più semplice utilizzare il for; la condizione *per i che va da 3 a 100* si può tradurre:

```
>>>  for i in range(3, 101):
```

- Altre funzioni di controllo.

La funzione **`break`**, come in C, interrompe forzatamente un ciclo.

La funzione **`continue`** salta forzatamente al passo successivo del ciclo.

La funzione **`pass`** …non fa niente![8]

[8] In realtà serve per riempire spazi vuoti, come quando ci sono condizioni che non danno particolari istruzioni.

4.2. Operazioni su liste

Le **operazioni su liste** (list comprehension) sono una maniera coincisa e semplice di utilizzare l'istruzione `for`. Si applica quando si deve effettuare un'operazione su ogni elemento di una lista, e si vuole ottenere una *lista di risultati*.

La sintassi è:

```
>>>  [comando for elemento in lista if condizioni]
```

L'interprete restituisce una nuova lista, creata nel seguente modo:

su ogni elemento della lista iniziale che verifica le necessarie *condizioni* viene eseguita l'istruzione; il risultato di questa operazione è inserito nella nuova lista.

Le condizioni sugli elementi sono opzionali.

```
>>> vec = range(8)
>>> [3*x for x in vec]
[0, 3, 6, 9, 12, 15, 18, 21]
>>> [x**2 for x in vec if x > 3]
[16, 25, 36, 49]
>>> [x**2 for x in vec if x < 0]
[]
>>> [[x,(2*x**2)+(x*6)-5] for x in vec]
[[0,-5],  [1,3],  [2,15],  [3,31],  [4,51],  [5,75],
[6,103], [7,135]]

>>> vec1 = [8, 6]
>>> vec2 = [-4, 7]
>>> [x*y for x in vec1 for y in vec2]
[-32, 56, -24, 42]
>>> [x+y for x in vec1 for y in vec2]
[4, 15, 2, 13]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[-32, 42]
```

4.3. Errori

Come in ogni altro linguaggio di programmazione, anche il Python gestisce un controllo per segnalare eventuali errori comparsi durante l'esecuzione di una istruzione.

Gli errori segnalati possono essere di sintassi o eccezioni.

- Errori di **sintassi**.

Gli errori di sintassi si verificano quando l'interprete individua una espressione o un comando sintatticamente scorretto, ad esempio (errore piuttosto frequente) se si sono dimenticati i due punti in un'istruzione come il `for` o il `while`.

Nella maggior parte delle versioni Python l'interprete, oltre a segnalare la presenza di un errore, riporta la linea che contiene l'errore, ed evidenzia la parola successiva a quella in cui è occorso l'errore.

```
>>>  while i < n pass
     File "<stdin>", line 1
     while i < n pass
                    ^
SyntaxError: invalid syntax
```

In questo caso la parola viene evidenziata per mezzo di un ^ sotto la sua ultima lettera.

- **Eccezioni**.

Può accedere che l'interprete non segnali alcun errore in fase di compilazione (la sintassi di un'espressione è corretta), ma che la sua esecuzione si interrompa avvisando la presenza di una eccezione. Questo avviene quando ad esempio vengono effettuate operazioni non corrette matematicamente, come una divisione per zero, o la radice quadrata di un numero negativo nel campo reale.

In questi casi viene segnalata il tipo d'eccezione per mezzo di una stringa contenente il nome dell'eccezione e una breve descrizione del problema.

```
>>>  sqrt (-5)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
Value error: math domain error
```

Esistono alcune funzioni specifiche in grado di forzare una determinata eccezione (il comando `raise`), o di gestire i segnali di output (il costrutto `try ... except`) in modo da testare un programma in situazioni differenti, e per studiarne i casi limite.

5. LE FUNZIONI

5.1. Definire una funzione

È possibile usare funzioni definite dal programmatore anche nell'ambiente interattivo; ovviamente queste vengono cancellate quando l'interprete viene chiuso.

In alternativa è possibile crearle in appositi file di testo, detti **moduli**, in modo tale da poter essere richiamate in ogni momento.

Vediamo ora come definire una funzione nell'ambiente Python interattivo.

Iniziamo con un esempio.

Vogliamo creare una funzione che stampi i primi termini della serie di Fibonacci.

```
>>>  def fib(n):     # stampa la serie di Fibonacci
...        """Stampa la serie di fibonacci fino ad
n."""
...        a, b = 0, 1
...        while b < n:
...        print b,
...        a, b = b, a+b
...
```

ora chiamiamo la funzione appena definita:

```
>>>  fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La definizione inizia con la parola chiave **`def`**, che deve essere seguita dal nome della funzione, da parentesi tonde (anche se non compaiono argomenti) *e dai soliti due punti*. Il corpo della funzione ovviamente deve essere indentato, per le ragioni viste prima.

Il ruolo dei tripli apici nella prima riga è quella di racchiudere un commento; questo però, a differenza dei commenti #, non viene del tutto ignorato dall'interprete, ma è considerato una "doc-string", cioè una stringa che documenta la funzione. Ovviamente è opzionale.

La chiamata di una funzione genera una nuova tabella dei simboli, in modo da non far confusione tra variabili locali e variabili globali.

Quando in una funzione viene richiamata una variabile, l'interprete cerca il suo nome *prima nella tabella locale*, *dopo in quella globale e infine nella tabella built in*. Le assegnazioni creano sempre variabili locali.

Di conseguenza, in generale è impossibile assegnare un valore ad una variabile "globale" all'interno di una funzione; è necessario specificare all'interprete di trattare tale variabile come globale, per mezzo dell'istruzione `global`.

```
>>>  a=5
>>>  def azzera1:
...       a=0
...
>>>  azzera1
>>>  a
5
>>>  def azzera2:
...       global a        # ora la variabile a è globale
...       a=0
...
>>>  azzera2
>>>  a
0
```

Gli argomenti della funzione vengono assegnati nella tabella dei simboli locale non appena la funzione viene chiamata, tramite una *chiamata per valore* (dove tale valore è il *riferimento* ad un oggetto, non il valore di un oggetto).

Quindi *una funzione non è in grado di modificare direttamente i propri parametri*.

```
>>>  a=5
>>>  def azzera(x):
...       x=0
...
>>>  azzera(a)
>>>  a
5
```

5.2. L'istruzione `return`

Come accade nel C, anche nel Python non c'è molta differenza tra procedure e funzioni. Infatti se volessimo esser precisi, le funzioni definite nel paragrafo precedente sono delle procedure, e nessuna delle tre restituisce un valore.

Per restituire un valore in output con l'esecuzione della funzione si utilizza l'istruzione **`return`**.

Se questa non è presente o se compare senza alcun argomento, la funzione assume il valore default: `None`.

```
>>> def somma(x, y):
...       '''restituisce la somma di due numeri'''
...       return x+y
...
>>>  somma(1, 4)
5
```

In questo esempio la funzione somma ha un valore intero.

Non appena la funzione incontra un `return`, essa assume il valore del suo argomento e torna al programma esterno.

Possono comparire anche più istruzioni `return`; in questo caso la funzione assume il valore del primo `return` che incontra, e viene interrotto il flusso di esecuzione delle istruzioni.

```
>>> def fatt(n):
...       '''restituisce   il   fattoriale   di   un
numero'''
...       if n<0: return
...       else:
...            a=1
...            if n!=0:
...                 for i in range(1,n+1): a=a*i
...       return a
...
>>>  fatt(4)
24
>>>  fatt(-5)
>>>
```

In questo la caso la funzione assume valore intero se $n \geq 0$, mentre assume valore `None` se $n < 0$ (e non viene stampato alcun risultato).

5.3. Parametri default e keywords

Nel definire una funzione è possibile specificare un valore predefinito per un determinato parametro; in questo caso quando si richiama la funzione non è necessario inserire un valore per tale parametro, in quanto già la funzione stessa gliene assegna uno.

```
>>>  def stampa(n='nessun testo'):
...       '''stampa una stringa'''
...       print n
...
>>>  stampa(9)
9
>>>  stampa('ciao')
ciao
>>>  stampa()
nessun testo
```

I valori predefiniti sono calcolati solo una volta per ogni funzione definita.

```
>>>  x=3
>>>  def stampa(n=x):
...       '''stampa una stringa'''
...       print n
...
```

```
>>>  stampa()
3
>>>  x=2
>>>  stampa()
3
```

Gli argomenti di una funzione possono essere costituiti da campi della forma: `keyword = valore`.

Questa sintassi è utile nel caso vi siano più argomenti predefiniti, e se ne vogliano assegnare solo in parte.

In ogni caso *gli argomenti predefiniti devono sempre seguire argomenti normali*.

Ecco alcuni esempi corretti:

```
>>>  def stampa(x, y=0, z=1):
...       '''stampa tre valori'''
...       print x,y,z
...
>>>  stampa(2)                  # viene assegnato 'x'
2 0 1
>>>  stampa(2,4)                # vengono assegnati 'x' e 'y'
2 4 1
>>>  stampa(2,z=4)              # vengono assegnati 'x' e 'z'
2 0 4
>>>  stampa(0,5,4)              # vengono assegnati 'x' 'y' e 'z'
0 5 4
>>>  stampa(0,z=5,y=4)          # vengono assegnati 'x' 'z' e 'y'
0 4 5
>>>  stampa(x=12,z=5,y=4)       # vengono assegnati 'x' 'z' e 'y'
12 4 5
```

Ecco ora alcuni esempi che generano errori:

```
>>>  stampa(2,5,y=4)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
TypeError: stampa() got multiple values of keyword
argument 'y'              # viene assegnata 'y' due volte.

>>>  stampa()
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
TypeError: stampa() takes at least 1 argument (0
given)          # non è assegnata 'x', che non ha valori predefiniti.

>>>  stampa(x=5,z=y)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
NameError: name 'y' is not defined
# anche se 'y' e 'z' hanno valori predefiniti, tali valori non sono usati
# se non dopo aver appurato che non ci siano valori assegnati.
```

5.4. Funzioni con numero variabile di parametri

Se abbiamo bisogno di trattare un numero arbitrario di argomenti possiamo utilizzare parametri del tipo: **argument* oppure ***argument*.

Un parametro preceduto da un asterisco sta ad indicare alla funzione che in quel punto saranno immessi un numero arbitrario di valori (per mezzo di una *lista*), mentre un parametro preceduto da due asterischi indica invece che in quel punto saranno immessi valori per mezzo di un *dizionario*.

Per evitare errori, i parametri con un asterisco devono essere definiti prima dei parametri con due asterischi.

```
>>>  def negozio(articolo, *arguments, **keywords):
...       print '-- Per caso avete', articolo, '?'
...       for arg in arguments: print arg
...       print '-'*40
...       keys = keywords.keys()
...       keys.sort()
...       for   kw   in   keys:   print   kw,   ':',
keywords[kw]
...
```

Richiamiamo ora la funzione.

```
>>>  negozio('della crescenza', "No, mi dispiace",
...       "sono desolato.",
...       "E' completamente finita",
...       cliente='Mario',
...       gestore='Antonio',
...       scena='Al Negozio')
```

Si ha il seguente risultato:

```
-- Per caso avete della crescenza ?
No, mi dispiace
Sono desolato.
E' completamente finita
----------------------------------------
cliente : Mario
gestore : Antonio
scena : Al negozio
```

In questo esempio l'interprete associa il primo parametro ad *articolo*, i successivi ad **arguments*, finché non arriva a dati di dizionario, che assocerà quindi a ***keywords*.

6. MODULI

Se usciamo dall'interprete per poi riavviarlo osserviamo che tutte le definizioni fatte in precedenza sono state cancellate dalla memoria del programma.

Per poter conservare le proprie funzioni o i propri programmi, è necessario memorizzarli in opportuni file di testo, e successivamente richiamarli con l'interprete[9].

Questa operazione è definita come *creare uno script*.

Se inoltre volessimo creare un programma complesso, che converrebbe dividere in più file per migliorarne la gestione, avremmo bisogno di definire alcune funzioni in file appoggio, per poi richiamarle in maniera rapida all'interno dello script.

Tali file di appoggio sono detti **moduli**, e sono anch'essi file di testo, contenenti definizioni e istruzioni, ma sono dipendenti da altre istruzioni. Il nome del file di testo in cui è memorizzato un modulo è dato dal nome del modulo e dall'estensione .py .

Ad ogni modulo richiamato (e quindi compilato) l'interprete aggiunge una costante `__name__` in cui viene memorizzato il nome del modulo richiamato, sotto forma di stringa.

Ogni modulo utilizza una propria tabella dei simboli, indipendente dal programma che richiama il modulo.

Vediamo un esempio. Abbiamo creato un file di testo "fibo.py" e ora vogliamo inserirci le definizioni di due funzioni che studino la serie di Fibonacci: la prima ne stampa il risultato, la seconda ne restituisce i valori in una lista.

```
def fib(n)
     '''stampa la serie di Fibonacci fino a n'''
     a, b = 0, 1
     while b < n:
          print b,
          a, b = b, a+b

def fib2(n):
     '''restituisce la serie di Fibonacci'''
     result = []
     a, b = 0, 1
     while b < n:
          result.append(b)
          a, b = b, a+b
     return result
```

A questo punto salviamo le modifiche, chiudiamo il file, e apriamo l'interprete. Al prompt diamo il seguente comando:

[9] In alcune versioni precedenti di Python, per poter compilare uno script in ambiente Linux, il testo deve iniziare con una delle seguenti istruzioni: `#!/usr/bin/python` oppure `#!/usr/bin/python2`. Tali istruzioni sono ignorate se il file è compilato in Windows.

```
>>>  import fibo     # solo il nome del modulo, senza estensione
```

In questo modo non viene aggiunta nessuna nuova definizione nella tabella dei simboli, solamente il nome del modulo. Le funzioni possono essere richiamate come metodi relativi al modulo.

```
>>>  fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>  fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>  fibo.__name__
'fibo'
>>>  fib = fibo.fib
>>>  fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Un modulo può contenere anche istruzioni eseguibili, a scopo di inizializzare il modulo. Tale inizializzazione avviene solo la prima volta che il modulo viene richiamato.

Esiste un altro modo di richiamare funzioni da un modulo, in modo che nella tabella dei simboli vengano inserite solo determinate funzioni, e non il nome del modulo.

In questo modo è come se tali funzioni fossero state appena definite e potranno essere richiamate come normali funzioni:

```
>>>  from fibo import fib fib2
>>>  fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Per richiamare tutte le funzioni presenti in un modulo basta usare il "carattere jolly" * dopo `import`.

La funzione **`dir()`** è utilizzata per elencare tutti i nomi definiti in un modulo[10].

Per esempio possiamo importare la libreria **`sys`** e elencare i nomi in essa contenuti. In particolare è presente la lista **`path`** che contiene un elenco di tutti i *percorsi di ricerca* dell'interprete, ossia tutte le locazioni del disco fisso che l'interprete controlla quando deve importare un modulo.

Se volessimo aggiungere un nuovo percorso (per es. il desktop di Windows 98) dovremmo scrivere:

```
>>>  from sys import path
>>>  path.append("C:\Windows\Desktop")
```

Se ora digitiamo `path` otterremo la lista di tutti i percorsi di ricerca, compreso quello aggiunto ora. Possiamo quindi richiamare moduli anche se questi non sono stati memorizzati nei percorsi di ricerca predefiniti.

[10] In realtà tale funzione ha anche altri utilizzi, e può ricevere come input un qualsiasi *oggetto*.

7. LA GRAFICA

In molte versioni del Python è presente un modulo chiamato **`turtle`** che vi consente di usare l'interprete per creare effetti grafici.

Ad esempio è possibile creare un programma che riceva come input una funzione $f(x)$ e ne disegni un grafico approssimativo $y = f(x)$.

Non c'è bisogno di fare alcuna inizializzazione grafica: appena l'interprete incontra la prima funzione definita nel modulo `turtle` fa da sé tutte le inizializzazioni necessarie e apre uno schermo in una nuova finestra a cui passa gli output grafici.

Ecco le principali funzioni presenti nel modulo:

`reset()`
cancella lo schermo e resetta tutte le impostazioni.

`clear()`
cancella lo schermo.

`up()`
alza la penna: si può spostare il cursore senza scrivere.

`down()`
abbassa la penna: ogni spostamento del cursore è *tracciato* sullo schermo.

`forward(x)`
avanza il cursore di una distanza x nella direzione impostata.

`backward(x)`
arretra il cursore di una distanza x nella direzione impostata.

`degrees()`
imposta l'unità di misura degli angoli in gradi.

`radians()`
imposta l'unità di misura degli angoli in radianti.

`left(a)`
muove a sinistra la direzione del cursore di a unità (radianti o gradi).

`right(a)`
muove a destra la direzione del cursore di a unità (radianti o gradi).

`width(x)`
imposta lo spessore della traccia.

goto(x,y)

sposta il cursore dalla posizione corrente a quella specificata dalle coordinate (x,y). Il piano cartesiano della finestra aperta ha l'origine nel centro della finestra, e una orientazione classica:

- l'asse delle ascisse va da sinistra a destra
- l'asse delle ordinate va dal basso in alto.

tracer(x)

imposta la modalità di avanzamento del cursore, 0 o 1:

0 : nessuna animazione, ma viene stampato solo il risultato finale

1 : con animazione, si vede il cursore muoversi lentamente.

color(s)

imposta il colore della traccia; s deve essere una stringa (es `s='blue'`)

color(r,g,b)

imposta il colore della traccia, dove r, g, b devono essere valori tra 0 e 1; viene impostato il colore ottenuto combinando i tre valori (indicanti le proporzioni di `"red", "blue", "green"` da mescolare)

fill(x)

imposta la modalità di riempimento dell'immagine disegnata;
la sintassi completa è complessa, conviene assegnare 1 prima di disegnare l'immagine, e 0 alla fine.

write(t,b=False)

scrive il testo t nella posizione corrente; se b è True il cursore si posiziona nell'angolo basso in fondo al testo.

circle (r,a=[max])

descrive un cerchio di raggio r partendo dalla posizione corrente in senso antiorario; il parametro a indica la porzione da descrivere (in gradi o radianti) ed è predefinito all'intera circonferenza.

demo()

è una funzione che esegue un programma d'esercitazione; contiene alcune delle istruzioni sopra descritte.

Bisogna aggiungere inoltre che il modulo `turtle` richiama a sua volta il modulo **math** contenente diverse funzioni matematiche (come il seno, il logaritmo, l'estrazione a radice, e la stessa conversione gradi ↔ radianti).

Di seguito è descritta la funzione `demo()` presente nella versione 2.3 di Python per Windows. Questa funzione è memorizzata nel file turtle.py presente tra le librerie di Python.

```
def demo():
     reset()
     tracer(1)
     up()
     backward(100)
     down()               # draw 3 squares; the last filled
     width(3)
     for i in range(3):
          if i == 2: fill(1)
          for j in range(4):
               forward(20)
               left(90)
          if i == 2:
               color("maroon")
               fill(0)
          up()
          forward(30)
          down()
     width(1)
     color("black")       # move out of the way
     tracer(0)
     up()
     right(90)
     forward(100)
     right(90)
     forward(100)
     right(180)
     down()               # some text
     write("startstart", 1)
     write("start", 1)
     color("red")         # staircase
     for i in range(5):
          forward(20)
          left(90)
          forward(20)
          right(90)       # filled staircase
     fill(1)
     for i in range(5):
          forward(20)
          left(90)
          forward(20)
          right(90)
     fill(0)              # more text
     write("end")
     if __name__ == '__main__':
          _root.mainloop()
```

BIBLIOGRAFIA

[1] G. van Rossum, « Python Tutorial » , Fred L. Drake, Jr., edit, 19 Dicembre 2003: la documentazione on-line sul Python, disponibile sul sito ufficiale, http://www.python.org/doc/

[2] Marco Buzzo, « Python, più di un semplice linguaggiodi script » , 1999, prelevabile dalla pagina web http://www.python.it/Doc/kranio-0.html

[3] S. D'Ambrosio, P. Mentrasti: *Impadronirsi di Python*, Dipartimento di Matematica dell'Università di Roma "La Sapienza", 2003 / 10

[4] D. Harms, K. McDonald, *The Quick Python Book*, Manning, Greenwich CT 2000