Appunti Python

Sakya
sakya_tg@yahoo.it

22 febbraio 2007

Indice

Capitolo 1

Prima di scrivere codice

Prima di iniziare a scrivere codice Python, ci sono alcune peculiarità della sintassi che è necessario conoscere.

1.1 Indentazione del codice

L'indentazione in Python fa parte della sintassi. Ciò significa che gli spazi lasciati all'inizio di una riga non servono solamente a rendere più leggibile il codice, ma ne modificano il comportamento.

Esempio:

```
if a < b:
  print a
a=5
```

In questo esempio la seconda istruzione è preceduta da due spazi in quanto fa parte del corpo dell'istruzione *if*. La terza riga, che non è preceduta da nessuno spazio e si trova al "livello" della prima riga, non fa parte dell'istruzione *if*.
In questo esempio è chiaro che gli spazi che precedono la seconda istruzione servono a Python per capire come interpretarla.

1.2 Operatori matematici

Gli operatori matematici in Python sono i seguenti:

Operatore	*Significato*	*Esempio*
=	Uguaglianza	a=5.0
-	Sottrazione	5.5-3.0=2.5
+	Somma	5.5+3.0=8.5
*	Moltiplicazione	5.5*2=11.0
/	Divisione	5.0/2=2.5
//	Divisione intera	5.0//2=2.0
%	Resto della divisione intera	5.0%2=1.0
pow(a,n)	Potenza *n* di *a*	pow(2,3)=8

1.3 Operatori comparativi

Gli operatori comparativi servono per costruire espressioni che verifichino il valore di una o più variabili. Essi permettono di confrontare due valori.
Gli operatori comparativi in Python sono:

Operatore	*Significato*	*Esempio*
==	Uguale	a==b
>	Maggiore	a>b
>=	Maggiore-uguale	a>=b
<	Minore	a<b
<=	Minore-uguale	a<=b
!=	Diverso	a!=b
is	Uguale (per oggetti)	a is b
is not	Diverso (per oggetti)	a is not b
in [a,b,c,..]	Contenuto nella sequenza [...]	a in [1,2,3]
not in [a,b,c,..]	Non contenuto nella sequenza [...]	a not in [1,2,3]

Al posto dell'operatore *!=* è possibile utilizzare l'operatore *<>*, ma quest'ultimo è deprecato, ed è quindi consigliato l'utilizzo di *!=*
La differenza tra l'operatore `==` e l'operatore `is` consiste nel fatto che quest'ultimo restituisce True solo se a sinistra ed a destra dell'operatore si trova lo stesso oggetto, mentre l'operatore `==` confronta i valori degli oggetti.

Esempio:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> c = a
# a is b restituisce False perché sono due
# oggetti distinti, anche se i valori che contengono
# sono uguali.
>>> a is b
False
# a == b restituisce True perché i due oggetti
# contengono gli stessi valori.
```

```
>>> a == b
True
# a is c restituisce True perché sono
# lo stesso oggetto (ed i valori sono
# sicuramente uguali)
>>> a is c
True
>>> a == c
True
```

1.4 Gli operatori logici

Gli operatori logici servono a collegare tra loro confronti tra fattori.
Gli operatori logici in Python sono:

- *and* restituisce vero solo se sia il fattore di sinistra che quello di destra è vero.

  Esempio:
  ```
  >>> a = 1
  >>> b = 1
  >>> c = 2
  >>> a == b and c == 2
  True
  >>> a == b and b == c
  False
  ```

- *or* restituisce vero solo se almeno uno dei due fattori è vero.

  Esempio:
  ```
  >>> a = 1
  >>> b = 2
  >>> a == 1 or b == 1
  True
  >>> a == 1 or b == 2
  True
  >>> a == 2 or b == 1
  False
  ```

I confronti possono essere concatenati nel seguente modo:

```
a == b == c
```

che equivale a scrivere:

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

1.5 Le condizioni

Una condizione è una espressione che può assumere il valore di vero o falso.
Una condizione è composta da uno o più confronti tra fattori utilizzando gli operatori comparativi.
I singoli confronti sono legati tramite operatori logici.

Esempio:

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> a == 1 and b == 1 or c == 3
True
```

Valutando una condizione Python procede da sinistra verso destra. Appena il risultato della condizione è stato calcolato la verifica si interrompe ed il risultato è restituito.

Esempio:

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> a == 1 and b == 2 or c == 3
True
```

Nell'esempio appena riportato il confronto tra c e 3 non viene mai eseguito perché dopo aver confrontato a con 1 e b con 2 la condizione è già vera. Il terzo confronto viene evitato perché ininfluente sul risultato della condizione.

Nelle condizioni è possibile utilizzare le parentesi per indicare l'ordine nel quale risolvere le condizioni.

Esempio:

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> a == 2 and (b == 1 or c == 3)
False
>>> a == 2 and b == 1 or c == 3
True
```

Capitolo 2

Tipi di dati

2.1 Plain integers

Numeri interi nell'intervallo da -2147483648 a 2147483647.

Esempio:

```
>>> a=5
>>> b=3
>>> a*b
15
>>> a/b
1
>>> a-b
2
>>> a+b
8
>>> a*b
15
```

2.2 Long integers

Numeri interi di illimitate cifre.

2.3 Booleans

Numero che può assumero solo i valori zero ed uno. Se viene convertito a stringa restituisce "True" per il valore 1 e "False" per il valore 0.

Esempio:

```
>>> a=True # Attenzione alla maiuscola true è <> True
>>> a True
```

```
>>> a * 2 # a vale 1 2
>>> a=False # Attenzione alla maiuscola false è <> False
>>> a False
>>> a * 2 # a vale 0 0
```

2.4 Floating Point

Numeri a doppia precisione. Il limite di questo tipo di variabili dipende dall'architettura della macchina sulla quale il programma viene eseguito.

Esempio:

```
>>> a=5.0 # Scrivendo a=5 la variabile a sarebbe stata
>>> b=3.0 # di tipo integer!
>>> a*b
15.0
>>> a/b
1.6666666666666667
>>> a+b
8.0
>>> a-b
2.0
>>> a*b
15.0
```

2.5 Complex

Rappresentazione dei numeri complessi tramite due numeri di tipo Floating Point. Uno rappresenta la parte reale (estratto con l'attributo *real*) e l'altro la parte immaginaria (estratta con *imag*). La funzione *complex(real, imag)* restituisce il numero complesso con parte reale *real* e parte immaginaria *imag*.

Esempio:

```
>>> a=1-3j
>>> a.real
1.0
>>> a.imag
-3.0
>>> complex(3,1) GM_WAITOPPONENTS
(3+1j)
```

2.6 Le sequenze

Una sequenza in python rappresenta una serie finita di elementi indicizzata con numeri interi e positivi.

La funzione len() restituisce il numero di elementi presente nella serie. L'indice di una sequenza inizia sempre da zero, quindi gli indici validi vanno da 0...len(seq)

L'elemento *n* di una sequenza *a* viene rappresentato con *a[n]*

Per selezionare più elementi contigui di una sequenza è possibile utilizzare *a[i:j]*. L'istruzione restituirà tutti gli elementi della sequenza a con indice maggiore o uguale di i (>= i) e minore di j (< j). La sezione estratta sarà una nuova sequenza, con l'indice che parte da zero.

Le sequenze, si dividono in due grandi insiemi: le sequenze immutabili (quelle che contengono dati che non possono essere modificati una volta creata la sequenza) e le sequenze modificabili (i cui dati possono essere modificati).

2.6.1 Strings

Contiene caratteri. Ogni carattere è rappresentato da una stringa di lunghezza uno. La funzione ord() restituisce il numero intero che rappresenta un carattere, la funzione chr() restituisce il carattere rappresentato dal un codice. Le stringhe possono essere rappresentate fra apici o fra doppi apici.

Esempio:

```
>>> 'prova stringa'
'prova stringa'
>>> 'c\'era una volta'
"c'era una volta"
>>> "c'era una volta"
"c'era una volta"
>>> '"A"'
'"A"'
```

Una stringa può essere composta da più righe. Per continuare una stringa sulla riga successiva, utilizzare il carattere \

Esempio:

```
>>>>a = "Questa è la prima parte \
...  e questa è la seconda"
>>> print a Questa è la prima parte e questa è la seconda
```

Per andare a capo con la stringa, utilizzare i caratteri \n\

Esempio:

```
>>> a = "Questa è la prima riga \n\
...e questa è la seconda"
>>> print a Questa è la prima riga
e questa è la seconda
```

Le stringhe possono essere sommate e moltiplicate:

Esempio:

```
>>> a = "Ciao" + "!" # <-- Avrebbe avuto lo stesso effetto

>>> print a * 5 # scrivere a = “Ciao” “!”
Ciao!Ciao!Ciao!Ciao!Ciao!
```

Essendo una sequenza, in una stringa si possono manipolare i caratteri (che sono gli elementi della sequenza). I caratteri di una stringa vengono numerati nel seguente modo:

C	**i**	**a**	**o**		**m**	**o**	**n**	**d**	**o**
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Esempio:

```
>>> a = "Ciao mondo"
>>> len(a) 10
>>> a[0] 'C'
>>> a[0:2] 'Ci'
>>> a[:2] # Il valore di default è 0 'Ci'
>>> a[5:10] 'mondo'
>>> a[5:]  # Il valore di default è len(a) 'mondo'
>>> a[-2] 'd'
>>> a[-2:]  'do'
>>> a[:-2] 'Ciao mon'
```

2.6.1.1 Convertire altri tipi di dati in stringhe

Per convertire altri tipi di dati in valori stringa sono disponibili le funzioni *str()* e *repr()*.

str() è fatto per convertire tipi di dati che sono già abbastanza “umanamente comprensibili”, mentre *rep()* è fatto per restituire la rappresentazione del dato in modo leggibile dall'interprete.

Esempio:

```
>>> str('prova')
'prova'
>>> repr('prova')
"'prova'"
>>> str(12.45)
'12.45'
>>> repr(12.45)
'12.449999999999999'
```

Molti tipi di dati vengono convertiti nello stesso modo da *str()* e da *repr()*, le stringhe ed i floating point hanno invece una rappresentazione differente.

2.6.1.2 Formattare stringhe con %

L'operatore % può essere usato per formattare numeri. Esso legge la parte alla sua sinistra come una stringa di formattazione per *sprintf()* e la applica alla parte alla sua destra.

Esempio:
```
>>> print '%5.3f' % 1.2377895543
1.238
```

A sinistra dell'operatore % è possibile indicare più stringhe di formattazione. In questo caso a destra deve essere indicato un tuple.

Esempio:
```
>>> print '%5.3f   %5.3f' % (1.342267, 62.3342)
1.342   62.334
```

La sintassi della formattazione con % è la seguente:

```
%(chiave) flag dimensione.precisione tipo_conversione
```

- la `chiave` (opzionale) identifica il nome della variabile che si intende formattare. Questo ha senso solo se si utilizzano più stringhe di formattazione.

  Esempio:
  ```
  >>> print '%(uno) 5.3f   %(due)5.3f' % {'uno':1.342267,
  'due':62.334}
  ```

- il `flag` (opzionale) può essere uno dei seguenti:
  - # utilizza la forma alternativa.
  - **0** utilizza il carattere "0" per il padding.
  - **-** giustifica a sinistra.
  - **spazio** inserisce uno spazio a sinistra dei numeri positivi.
  - + inserisce il segno + o - prima del valore numerico formattato.

- `dimensione` (opzionale) indica quanti caratteri dovrà occupare la stringa formattata. Se viene impostato a * allora la lunghezza verrà letta dal valore numerico.

- `precisione` (opzionale) precisione (numero di decimali dopo la virgola). Se viene impostato a * allora verrà letta dal valore numerico.

- `tipo_conversione` può assumere uno dei seguenti valori:

Tipo Conv.	Descrizione	
d, i	intero decimale con segno	
o	intero ottale senza segno	
u	intero decimale senza segno	
x	intero esadecimale senza segno (lettere minuscole)	
X	intero esadecimale senza segno (lettere maiuscole)	
e	virgola mobile in formato esponenziale (lettere minuscole)	`'numero`
E	virgola mobile in formato esponenziale (lettere maiuscole)	`'numero`
f, F	virgola mobile in formato decimale	`'numero %`
g	come "e" se l'esponente > -4 o inferiore alla precisione, altrimenti come "f"	
G	come "E" se l'esponente > -4 o inferiore alla precisione, altrimenti come "F"	
c	carattere singolo	
s	stringa convertita da un qualsiasi oggetto Python tramite str()	
r	stringa convertita da un qualsiasi oggetto Python tramite repr()	
%	inserisce il carattere %	

2.6.1.3 Metodi per le stringhe

Le stringhe hanno molti metodi utili alla loro gestione e manipolazione.

- *capitalize()* rende la prima lettera maiuscola e tutte le altre minuscole.

  Esempio:
  ```
  >>> a = 'tEsTo'
  >>> a.capitalize()
  'Testo'
  ```

- *center(n)* giustifica al centro una stringa all'interno di un'altra stringa di lunghezza *n*

  Esempio:
  ```
  >>> a.center(10)
  ' tEsTo '
  ```

- *count(car)* restituisce il numero delle ricorrenze del carattere *car* in una stringa. La ricerca tiene conto del maiuscolo/minuscolo.

  Esempio:
  ```
  >>> a = 'tEsTo'
  >>> a.count('t')
  1
  >>> a.count('T')
  1
  ```

- *expandtabs(num)* espande tutti i caratteri TAB che ci sono nella stringa e li sostituisce con una stringa lunga *num* caratteri. Se non viene specificato un *num* viene usato il valore di default 8.

Esempio:

```
>>> a = 'testo' + chr(9) + 'di prova' # chr(9) è il carattere
TAB
>>> a 'testo\tdi prova'
>>> a.expandtabs(10)
'testo di      prova'
>>> a.expandtabs(8)
'testo di   prova'
>>> a.expandtabs(9)
'testo di    prova'
```

- *find(sottostringa [,inizio [, fine]])* restituisce l'indice della prima ricorrenza partendo da sinistra della *sottostringa* nell'intervallo *inizio* - *fine* (sono interpretati come nella sintassi *stringa[inizio:fine]*). L'indice restituito fa sempre riferimento all'intera stringa, anche se vengono usati *inizio* e *fine*. Se la *sottostringa* non è presente nella stringa viene restituito il valore -1.

  Esempio:

  ```
  >>> a = 'testo lungo'
  >>> a.find('o')
  4
  >>> a.find('o', 2, 5)
  4
  >>> a.find('o', 2, 3)
  -1
  ```

- *rfind(sottostringa [,inizio [, fine]])* stesso funzionamento di *find()* ma la ricerca inizia da destra e non da sinistra.

- *index(sottostringa [,inizio [, fine]])* identico a *find()* con l'unica differenza che genera un errore se la *sottostringa* non viene trovata.

  Esempio:

  ```
  >>> a = "testo"
  >>> a.index('e') 1
  >>> a.index('z')
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  ValueError:  substring not found
  ```

- *rindex(sottostringa [,inizio [, fine]])* uguale a *index()* ma la ricerca inizia da destra e non da sinistra.

- *join(lista)* restituisce una stringa composta dagli elementi della lista concatenati con il contenuto della stringa come separatore.

Esempio:

```
>>> l = ['proviamo', 'a', 'metterle', 'insieme']
>>> a = " "
>>> a.join(l)
'proviamo a metterle insieme'
>>> a = "*"
>>> a.join(l)
'proviamo*a*metterle*insieme'
```

- *ljust(lung)* restituisce la stringa justificata a sinistra all'interno di una stringa di lunghezza *lung*.

  Esempio:

  ```
  >>> a = 'testo'
  >>> a.ljust(10)
  'testo     '
  >>> a.ljust(12)
  'testo       '
  ```

- *rjust(lung)* restituisce la stringa justificata a destra all'interno di una stringa di lunghezza *lung*.

  Esempio:

  ```
  >>> a = 'testo'
  >>> a.rjust(10)
  '     testo'
  >>> a.rjust(12)
  '       testo'
  ```

- *lower()* restituisce la stringa con tutte le lettere minuscole.

  Esempio:

  ```
  >>> a = "TEsTO"
  >>> a.lower()
  'testo'
  ```

- *lstrip([carattere])* restituisce la stringa senza gli spazi iniziali se nessun *carattere* viene specificato. Se un *carattere* è specificato, allore vengono cancellati tutti i *carattere* iniziali della stringa.

  Esempio:

  ```
  >>> a = "  testo"
  >>> a.lstrip()
  'testo'
  >>> a = "**testo"
  >>> a.lstrip()
  '**testo'
  >>> a.lstrip('*')
  'testo'
  ```

- *rstrip([carattere])* uguale a *lstrip()* ma elaborando la stringa da destra a sinistra.

- *replace (cerca, sost[, numero])* restituisce la stringa con tutte le sottostringhe *cerca* sostituite con la sottostringa *sost*. Se nessun *numero* viene passato vengono sotituite tutte le ricorrenze. Se viene passato un *numero*, verranno sostituite solo le prime *numero* ricorrenze.

  Esempio:
  ```
  >>> a = "OOxxPP"
  >>> a.replace("O", "1")
  '11xxPP'
  >>> a.replace("O", "1", 1)
  '1OxxPP'
  >>> a.replace("OO", "1")
  '1xxPP'
  ```

- *split([separatore [,maxsplit]])* restituisce una lista contenente la stringa divisa in parole. Per dividere le parole viene utilizzato il *separatore* (se non viene indicato viene impiegato spazio).
  Se viene indicato *maxsplit* il metodo cercherà il *separatore* solo *maxsplit* volte, restituendo tutto il resto nell'ultimo elemento della lista.

  Esempio;
  ```
  >>> a = "Testo con piu' di una parola"
  >>> a.split()
  ['Testo', 'con', "piu'", 'di', 'una', 'parola']
  >>> a.split(" ", 2)
  ['Testo', 'con', "piu' di una parola"]
  >>> a.split("o")
  ['Test', ' c', "n piu' di una par", 'la']
  ```

- *strip([carattere])* restituisce la stringa senza tutti i *carattere* iniziali e finali. Se nessun *carattere* viene specificato viene utilizzato lo spazio.

  Esempio:
  ```
  >>> a = "  testo  "
  >>> a.strip()
  'testo'
  >>> a = "**  testo**"
  >>> a.strip('*')
  '  testo'
  ```

- *swapcase()* restituisce la stringa con le maiuscole trasformate in minuscole e viceversa.

Esempio:

```
>>> a = "TeStO"
>>> a.swapcase()
'tEsTo'
```

- *translate(tabella [, carattere])* restituisce la stringa con i caratteri mappati con una *tabella* (che deve essere una stringa di 256 caratteri).
  Se viene indicato un *carattere* questo verrà eliminato dalla stringa prima di fare la decodifica.
  La decodifica cerca per ogni carattere della stringa l'elemento nella *tabella* che si trova alla posizione pari al codice ASCII del carattere da decodificare.

  Esempio:

```
>>> b = "123"
>>> a = " " * 256
>>> a = a[:49] + "x" + a[50:]  # 49 è il codice ASCII
di "1"
>>> a = a[:50] + "y" + a[51:]  # 50 è il codice ASCII
di "2"
>>> a = a[:51] + "z" + a[52:]  # 51 è il codice ASCII
di "3"
>>> b.translate(a)
'xyz'
>>> b.translate(a, "2")
'xz'
```

- *upper()* restituisce la stringa con tutti i caratteri maiuscoli.

  Esempio:

```
>>> a = "tEsto"
>>> a.upper()
'TESTO'
```

- *zfill(lunghezza)* restituisce la stringa espansa e *lunghezza* con gli spazi a sinistra riempiti di zeri. Se *lunghezza* è inferiore alla dimensione della stringa verrà restituita l'intera stringa iniziale senza modifiche.
  zfill() riconosce i segni + e - e li tratta correttamente.

  Esempio:

```
>>> a = "23"
>>> a.zfill(4)
'0023'
>>> a = "-23"
>>> a.zfill(5)
'-0023'
>>> a = "testo"
```

```
>>> a.zfill(7)
'00testo'
>>> a.zfill(2)
'testo'
```

2.6.2 Stringhe UNICODE

Contengono caratteri unicode. Per creare una stringa unicode è sufficiente anteporre una u prima dell'inizio della stringa. Per inserire un carattere unicode nella stringa, utilizzare i caratteri $\backslash un$ dove n rappresenta il codice del carattere da visualizzare.

Esempio:

```
>>> u'Ciao\u0020Mondo' # 0020 è il codice dello spazio.
u'Ciao Mondo'
>>>>a = u"Ecco le lettere accentate:  àèéìòù!"
>>>>print a
Ecco le lettere accentate:  àèéìòù!
```

L'utilità delle variabili unicode è che possono contenere i caratteri speciali di tutte le lingue, e non un solo set di caratteri ASCII.

2.6.3 Liste

Le liste possono contenere un insieme eterogeneo di dati. Una lista può essere definita da un elenco di valori separati da virgola. L'indice degli elementi parte da zero, e vi si possono eseguire le funzioni base di ogni sequenza.

Esempio:

```
>>> a = ['a', 'b', 'c', 'd']
>>> a ['a', 'b', 'c', 'd']
>>> a[0] 'a'
>>> a[0:]  ['a', 'b', 'c', 'd']
>>> a[0:2] ['a', 'b']
>>> a[-2] 'c'
>>> a[2]*3 'ccc'
```

A differenza delle stringhe, le liste possono essere modificate.

Esempio:

```
>>> a[1]='x' # Sostituzione di un valore
>>> a
['a', 'x', 'c', 'd']
>>> a[0:2] = [1, 2] # Sostituzione di più valori
>>> a [1, 2, 'c', 'd'] # Elementi numerici ed alfabetici
```

E' possibile nidificare le liste.

Esempio:

```
>>> a = [1, 2, 3, 'Fine']
>>> b = [a, 'Qui', 'Quo', 'Qua']
>>> a
[1, 2, 3, 'Fine']
>>> b
[[1, 2, 3, 'Fine'], 'Qui', 'Quo', 'Qua']
>>> b[0]
[1, 2, 3, 'Fine']
>>> b[0][0]
1
>>> b[0][1]
2
>>> a[0]='999' # Viene modificato anche b!!!
>>> b
[['999', 2, 3, 'Fine'], 'Qui', 'Quo', 'Qua']
```

2.6.3.1 Metodi per le liste

Le liste hanno diversi metodi per essere gestite. Essi sono:

- *append(x)* Aggiunge un elemento al temine della lista (equivalente a *a[len(a):] = [x]*).

  Esempio:

  ```
  >>> a = [1, 2, 3, 4]
  >>> print a
  [1, 2, 3, 4]
  >>> a.append(7)
  >>> print a
  [1, 2, 3, 4, 7]
  ```

- *extend(L)* aggiunge al termine della lista tutti gli elementi della lista L (equivalente a *a[len(a):] = L*)

  Esempio:

  ```
  >>> a = [1, 2, 3, 4]
  >>> b = [5, 6, 7]
  >>> a.extend(b)
  >>> print a
  [1, 2, 3, 4, 5, 6, 7]
  ```

- *insert(i ,x)* inserisce l'elemento x prima dell'elemento della lista con indice pari a i.

Esempio:

```
>>> a = [1, 2, 3, 4]
>>> a.insert(2,0)
>>> print a
[1, 2, 0, 3, 4]
```

- *remove(x)* rimuove il primo elemento della lista il cui valore è uguale a x.

  Esempio:

```
>>> a = [1, 2, 3, 4]
>>> a = [1, 2, 1, 4]
>>> a.remove(1)
>>> print a
[2, 1, 4]
```

- *pop([i])* rimuove l'elemento con indice i e ne restituisce il valore. Se nessun indice viene passato al metodo, viene cancellato l'ultimo elemento della lista.

  Esempio:

```
>>> a = [1, 2, 3, 4]
>>> a.pop()
4
>>> a.pop(0)
1
>>> print a [2, 3]
```

- *index(x)* restituisce l'indice del primo valore nella lista che ha il valore x. Se nessun elemento in lista ha il valore x viene generato un errore.

  Esempio:

```
>>> a = [1, 2, 3, 4]
>>> a.index(3)
2
>>> a.index(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError:  list.index(x):  x not in list
```

- *count(x)* restituisce il numero di volte che il valore x ricorre nella lista.

  Esempio:

```
>>> a = [1, 2, 3, 4, 1, 5]
>>> a.count(1)
2
>>> a.count(5)
1
```

- *sort(cmp, key, reverse)* ordina i valori contenuti nella lista.
  Senza il passaggio di parametri la lista viene ordinata in modo ascendente utilizzando gli elementi della lista come chiave di ordinamento e la funzione di confronto standard `cmp(x, y)` (che restituisce un valore negativo se x $<$ y, un valore positivo se x $>$ y e zero se x $==$ y).

  Per modificare il tipo di ordinamento è possibile fornire al metodo `sort` una funzione da usare per calcolare l'ordinamento. La funzione deve essere nello stesso formato di `cmp(x, y)` e restituire un numero negativo se un valore negativo se x $<$ y, un valore positivo se x $>$ y e zero se x $==$ y.

  Il parametro `key` permette invece di specificare la chiave di ordinamento della lista.

  Esempi:

```
>>> a = [ 1, 4, 2, 3, 5]
>>> a.sort()
>>> print a
[1, 2, 3, 4, 5]

#Ordinamento con funzione ad-hoc:
>>> def confronto(elemento1, elemento2):
...  if elemento1 * 5 - 7 > elemento2 * 5 - elemento1:
...    return -1
...  elif elemento1 * 5 - 7 == elemento2 * 5 - elemento1:
...    return 0
...  elif elemento1 * 5 - 7 < elemento2 * 5 - elemento1:
...    return 1
>>> a = [1,2,3,4,5,6,7,8,9,10]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a.sort(cmp=confronto)
>>> a
[10, 9, 8, 7, 6, 5, 4, 3, 1, 2]

#Ordinamento con chiave:
>>> aa = ['b', 'ab', 'zab', 'cdef', 'fabcde']
>>> a.sort()
>>> a
['ab', 'b', 'cdef', 'fabcde', 'zab']
>>> a.sort(key=len)
>>> a
['b', 'ab', 'zab', 'cdef', 'fabcde']
```

- *reverse()* inverte l'ordine degli elementi nella lista.

Esempio:

```
>>> a = [ 1, 4, 2, 3, 5]
>>> a.reverse()
>>> print a
[5, 3, 2, 4, 1]
```

2.6.3.2 Altri comandi per la gestione delle liste

Esistono alcuni comandi che modificano il contenuto di una lista:

- *del lista[indice]* elimina l'elemento *indice* dalla lista *a*.
  E' posibile anche eliminare tutta la lista, oppure una sezione della lista.

  Esempio:

```
# Cancellazione di un elemento:
>>> a = [1, 3, 6, 2]
>>> del a[1]
>>> print a
[1, 6, 2]
# Cancellazione di più elementi contigui:
>>> a = [1, 3, 6, 2]
>>> del a[1:3]
>>> print a
[1, 6]
# Cancellazione dell'intera lista
>>> a = [1, 3, 6, 2]
>>> del a
>>> print a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError:  name 'a' is not defined
```

- *filter(funzione, sequenza)* restituisce una sequenza (dello stesso tipo di quella passata, se possibile) contenente i soli elementi per i quali la *funzione*[1] risulta vera (*True* o un numero maggiore di zero).

  Esempio:

```
>>> def tf(n):
...     if n >= 3 and n <= 10:
...       return True
...     else:
...       return False
...
>>> a = [1,2,3,4,5,6,7,8,9,0]
>>> filter(tf, a)
[3, 4, 5, 6, 7, 8, 9]
```

[1] Le funzioni sono spiegate dettagliatamente nel capitolo 6 nella pagina 51. Per ora è sufficiente sapere che una funzione è un blocco di codice che restituisce un valore.

- *map(funzione, sequenza, sequenza, ...)* restituisce una lista contenente i valori restituiti dalla *funzione* per ogni elemento della *sequenza*.
  L'istruzione *map()* può utilizzare più di una sequenza, a patto che il numero delle sequenze corrisponda al numero di parametri che la funzione accetta.

  Esempio:
  ```
  >>> def add(n):
  ...     """Funzione che aggiunge 1 al numero passato"""
  ...     return n + 1
  ...
  >>> a=[1, 5, 4, 8]
  >>> map(add, a) # La funzione accetta un parametro!
  [2, 6, 5, 9]
  >>>
  >>> def add(a,b):
  ...     """Funzione che somma due numeri"""
  ...     return a + b
  ...
  >>> a=[1, 5, 4, 8]
  >>> b=[3, 7, 1, 9]
  >>> map(add, a, b) # La funzione accetta due parametri!
  [4, 12, 5, 17]
  ```

- *reduce(funzione, sequenza, [valore_iniziale])* restituisce un singolo valore così calcolato:
  Se è presente un *valore_iniziale* (non obbligatorio) il calcolo parte passando alla *funzione* (che deve accettare due parametri) il *valore_iniziale* ed il primo elemento della sequenza, se nessun *valore_iniziale* è stato definito il primo ed il secondo elemento della sequenza vengono passati alla *funzione*.
  Il valore restituito dalla funzione viene ripassato alla *funzione* insieme all'elemento successivo della sequenza. Il calcolo si ferma al raggiungimento del temine della sequenza.
  Se nella sequenza è presente un solo valore *reduce()* restituisce il valore dell'unico elemento.
  Se la sequenza è vuota *reduce()* genera un errore se nessun *valore_iniziale* è stato definito, restituisce il *valore_iniziale* se definito.

  Esempio:
  ```
  >>> def add(a,b):
  ...     """Funzione che somma due numeri"""
  ...     return a + b
  ...
  >>> a=[1, 5, 4, 8]
  >>> reduce(add, a) # Senza valore_iniziale
  18
  ```

```
>>> # 18 = ((5 + 1) + 4) + 8
>>>
>>> reduce(add, a, 3) # Con il valore_iniziale
21
>>> # 21 = (((3 + 1) + 5) + 4) + 8
```

- *sum(sequenza)* restituisce un singolo valore calcolato esattamente come nell'esempio riportato per l'istruzione *reduce()* e cioè:
  Il primo ed il secondo elemento della sequenza vengono sommati.
  Il risultato viene sommato col terzo elemento della sequenza

- *enumerate(lista)* è utile all'interno di un ciclo per recuperare con una sola istruzione sia l'indice che il valore di un elemento di una lista.

  Esempio:
  ```
  >>> a = [1, 4, 6]
  >>> for indice, valore in enumerate(a):
  ...     print 'Indice = ', indice, ' Valore = ', valore

  ...
  Indice = 0 Valore = 1
  Indice = 1 Valore = 4
  Indice = 2 Valore = 6
  ```

- *zip(lista1, lista2, ...)* crea una lista costituita da tuple, i cui elementi sono costituiti da un elemento della *lista1*, uno della *lista2*, ...
  L'istruzione *zip()* può essere utile in un ciclo per leggere contemporaneamente i valori da più liste contemporaneamente.

  Esempio:
  ```
  >>> a = [1, 2, 3]
  >>> b = ['a', 'b', 'c']
  >>> zip(a, b)
  [(1, 'a'), (2, 'b'), (3, 'c')]
  # Utilizzo di zip() in un ciclo:
  >>> for numero, lettera in zip(a,b):
  ...     print 'Num.  ', numero, ' Lett.  ', lettera
  ...
  Num.  1 Lett.  a
  Num.  2 Lett.  b
  Num.  3 Lett.  c
  ```

2.6.3.3 List comprehension

Per valorizzare una lista si ottiene un codice generalmente più leggibile utilizzando la sintassi chiamata *list comprehension* al posto di istruzioni come *map()* o *filter()*.

La *list comprehension* è composta da una espressione, seguita da un *for* (per la sintassi consultare la sezione 4.3 nella pagina 37) e da una serie di *if* facoltativa (per la sintassi consultare la sezione 4.1 nella pagina 34).
Se l'espressione rappresenta un tuple deve essere racchiusa tra [].

Il risultato della elaborazione è dato dalla valutazione della espressione all'interno del ciclo generato dalla istruzione *for* e valutando la serie di *if*.

Esempio:

```
# Per ogni valore x nella sequenza a
# viene valutata l'espressione 2*x
>>> a = [1, 3, 6, 2]
>>> [2*x for x in a]
[2, 6, 12, 4]
>>>
# Per ogni valore x nella sequenza a
# viene valutata l'espressione 2*x solo
# se x >= 3
>>> a = [1, 3, 6, 2]
>>> [2*x for x in a if x >= 3]
[6, 12]
# Per ogni valore x nella sequenza a
# viene valutata l'espressione [2*x, 3*x] solo
# se x >= 3
>>> [[2*x, 3*x] for x in a if x >= 3]
[[6, 9], [12, 18]]
# Per ogni valore x nella sequenza a
# e della sequenza b viene valutata l'espressione x*y
>>> a = [1, 3, 6, 2]
>>> b = [2, 4, 7, 3]
>>> [x*y for x in a for y in b]
[2, 4, 7, 3, 6, 12, 21, 9, 12, 24, 42, 18, 4, 8, 14, 6]
```

2.6.3.4 Uso delle liste in modalità LIFO e FIFO

Le liste ed i suoi metodi si prestano ad essere usate sia in modalità FIFO (first-in-first-out) che LIFO (last-in-first-out).

Per utilizzare le liste in modalità FIFO (cioè creare una lista dove il primo elemento inserito è il primo che viene processato):

- Aggiungere gli elementi alla lista col metodo *append(x)*.
  In questo modo l'elemento viene aggiunto in fondo alla lista.
- Per sapere quale elemento processare utilizzare il metodo *pop(0)*.
  Passando zero alla funzione *pop*() verrà restituito il primo elemento della lista e contemporaneamente verrà cancellato da essa.

Per utilizzare le liste in modalità LIFO (cioè creare una lista dove l'ultimo elemento inserito è il primo ad essere processato):

- Aggiungere gli elementi alla lista col metodo *append(x)*.
- Per sapere quale elemento processare utilizzare il metodo *pop()*.
  Richiamando il metodo *pop()* senza nessun parametro, verrà restituito l'ultimo elemento in lista e contemporaneamente verrà cancellato da essa.

2.6.4 Tuple

Un *tuple* è costituito da una serie di elementi separati da virgola e può essere nidificato.

Esempio:

```
>>> t = 1, 2, 'a'
>>> print t
(1, 2, 'a')
# Touple nidificato:
>>> t2 = t, (4, 5, 6)
>>> print t2
((1, 2, 'a'), (4, 5, 6))
```

Definendo un tuple le parentesi sono opzionali, anche se nella maggior parte dei casi diventano obbligaotorie se il touple è incluso in una istruzione più lunga.

Esempio:

```
>>> t = 1, 2, 3
>>> print t
(1, 2, 3)
>>> print t[0]
1
>>> t = (1, 2, 3)
>>> print t
(1, 2, 3)
```

I tuple, così come le stringhe, sono immutabili. Non è possibile, cioè modificare uno dei suoi elementi.
Un tuple può contenere dati modificabili (come una lista).

Esempio:

```
>>> a = [0, 1, 2]
>>> t = a, 1, 2
>>> print t
([0, 1, 2], 1, 2)
>>> a[0]=3
>>> print t
([3, 1, 2], 1, 2)
```

Vista la sintassi per definire un tuple sorge il problema di come definirne uno vuoto o con un elemento solo.
Per creare un tuple vuoto è sufficiente non inserire nessun dato tra le parentesi (che in questo caso diventano obbligatorie), mentre per definire un tuple con un solo elemento è sufficiente inserire il valore da assegnare seguito da una virgola.

Esempio:
```
# Definizione di un tuple vuoto
>>> t = ()
>>> print t
()
# Definizione di un tuple con un solo elemento
>>> t = 'prova',
>>> print t
('prova',)
```

L'istruzione utilizzata per definire un tuple (*t = 1, 2, 3, 4*) viene chianata *tuple packing*. L'operazione contraria (*sequence unpacking*) è supportata e viene utilizzata per trasferire il valore deglie elementi presenti nel tuple in variabili.

Esempio:
```
# Tuple packing:
>>> t = 1, 2, 3
# Sequence unpacking
>>> x, y, z = t
>>> print x
1
>>> print y
2
>>> print z
3
```

2.6.5 Il confronto tra sequenze

Le sequenze possono essere confrontate con oggetti dello stesso tipo. Il confronto avviene in ordine *lessicografico*.
Prima viene confrontato il primo elemento di una sequenza con il primo elemento dell'altra.
Se questi elementi sono differenti, confrontando il singolo elemento si ha la risposta del confronto tra le sequenze.
Se questi elementi sono uguali il confronto procede di elemento in elemento fino al termine delle sequenze.
Se tutti gli elementi sono uguali allora le due sequenze vengono considerate uguali.

Se una delle due sequenze è uguale alla porzione iniziale dell'altra, allora la prima è considerata minore della seconda.

Esempio:

```
>>> [1,2,3]==[1,2,3]
True
>>> [1,2,4]<[1,2,3]
False
>>> [1,2,3]<[1,2,3,4]
True
```

2.7 Dizionari

I dizionari differiscono dalla sequenze perché come indice utilizzano un campo alfanumerico (chiamato *key* e che può essere di un qualsiasi tipo immutabile*)* e non un numero intero.
Anche un tuple può essere utilizzato come chiave, ma a condizione che contenga solo stringhe, numeri o tuple. Se un tuple contiene anche indirettamente dei dati modificabili, allora non potrà essere utilizzato come chiave. Nemmeno le liste, dato che sono modificabili, possono essere utilizzate come chiave.

Un dizionario è una coppia di valori disordinati *chiave : valore*. La chiave deve essere univoca (in quanto utilizzata per fare riferimento ad un elemento come l'indice di una sequenza).
Per definire un dizionario si utilizzano le parentesi graffe *{}* ed i valori devono essere scritti nel formato chiave:valore.

Esempio:

```
>>> a = {'a':1, 'b':2}
>>> print a
{'a':  1, 'b':  2}
```

Per ottenere il valore abbinato ad una chiave è sufficiente utilizzare la sintassi vista con le sequenze, con la chiave al posto dell'indice numerico.
Per aggiornare il valore di un elemento del dizionario è sufficiente valorizzare l'elemento con la chiave che si intende aggiornare.
Per cancellare un elemento dal dizionario è possibile utilizzare l'istruzione *del()*.

Esempio:

```
>>> a = {'a':1, 'b':2}
# Ricerca di una chiave:
>>> print a['a']
1
# Cercare una chiave che non esiste
# genera un errore:
>>> print a['c']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError:  'c'
# Inserimento di un valore
>>>> a['c'] = 4
>>> print a
{'a':  1, 'b':  3, 'c':4}
# Aggiornamento di un valore:
>>> a['b']=3
>>> print a
{'a':  1, 'b':  3}
# Cancellazione di una chiave:
>>> del a['a']
>>> print a
{'b':  2}
```

2.7.1 Metodi per i dizionari

Anche i dizionari, ovviamente, hanno dei metodi utili alla loro gestione:

- *clear()* rimuove tutti gli elementi di undizionario.
- *copy()* restituisce un nuovo dizionario con gli stessi elementi (e non una copia degli elementi).

  Esempio:

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
>>> b = a.copy()
>>> print a
{'a':  1, 'c':  3, 'b':  2, 'd':  [1, 2]}
>>> print b
{'a':  1, 'c':  3, 'b':  2, 'd':  [1, 2]}
>>> a['a']=2
#La lista verrà modificata in entrambi i dizionari poiché
sono la stessa lista.
>>> a['d'].append(3)
>>> print a
{'a':  2, 'c':  3, 'b':  2, 'd':  [1, 2, 3]}
>>> print b
{'a':  1, 'c':  3, 'b':  2, 'd':  [1, 2, 3]}
```

- *fromkeys(chiavi, default)* crea un nuovo dizionario contenente le chiavi fornite ed un valore di default.

  Esempio:

```
>>> a = {}.fromkeys(['a','b','c','d'], 'Vuoto')
>>> print a
```

```
{'a': 'Vuoto', 'c': 'Vuoto', 'b': 'Vuoto', 'd': 'Vuoto'}
#Se non si specifica un valore di default verrà utilizzato
None:
>>> a = {}.fromkeys(['a','b','c','d'])
>>> print a
{'a': None, 'c': None, 'b': None, 'd': None}
```

- *get(chiave)* restituisce il valore dell'elemento con la chiave specificata. La differenza con la ricerca con la sintassi *dizionario[chiave]* è che il metodo *get* restituisce *None* se la chiave non esiste.

  Esempio:

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
>>> print a['e']
Traceback (most recent call last):
File "<stdin>", line 1, in ?
KeyError: 'e'
>>> print a.get('e')
None
```

- *keys()* restituisce una lista contenente le chiavi di tutti gli elementi presenti nel dizionario.
  Le chiavi di un dizionario non sono ordinate (vengono restituite in modo disordinato e casuale), per avere una lista ordinata delle chiavi utilizzare il metodo *sort()* sulla lista restituita dal metodo *keys()*.

  Esempio:

```
>>> a = {'a':1, 'b':2}
>>> print a.keys()
['a', 'b']
```

- *iterkeys()* è utile all'interno di un ciclo per scorrere le chiave degli elementi di un dizionario.

  Esempio:

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
>>> for chiave in a.iterkeys()
>>> for chiave in a.iterkeys():
... print 'Chiave: ' + chiave
...
Chiave: a
Chiave: c
Chiave: b
Chiave: d
```

- *has_key(chiave)* controlla se nel dizionario esiste una *chiave* e restituisce True o False.

Esempio:

```
>>> a = {'a':1, 'b':2}
>>> a.has_key('a')
True
>>> a.has_key('c')
False
```

- *items()* restituisce una lista contenente gli elementi del dizionario nel formato (chiave, valore). Anche in questo caso, come il metodo copy, i valori non vengono copiati.

  Esempio:

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
>>> b = a.items()
>>> print b
[('a', 1), ('c', 3), ('b', 2), ('d', [1, 2])]
>>> b[3][1].append(3)
>>> print a
{'a':  1, 'c':  3, 'b':  2, 'd':  [1, 2, 3]}
```

- *iteritems()* è utile all'interno di un ciclo per recuperare con una sola istruzione sia la chiave che il valore di un elemento di un dizionario.

  Esempio:

```
>>> a = {'a':1, 'b':2}
>>> for chiave, valore in a.iteritems():
...     print 'key = ', chiave, 'Valore = ', valore
...
key = a Valore = 1
key = b Valore = 2
```

- *pop(chiave)* restituisce il valore dell'elemento con la chiave specificata e lo rimuove dal dizionario.

  Esempio:

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
>>> a.pop('c')
3
>>> a
{'a':  1, 'b':  2, 'd':  [1, 2]}
```

- *popitem()* restituisce un elemento (in nessun ordine specifico) e lo rimuove dal dizionario.

  Esempio:

```
>>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
>>> a.popitem()
```

```
('a', 1)
>>> a.popitem()
('c', 3)
>>> a
{'b': 2, 'd': [1, 2]}
```

- *setdefault(chiave, default)* funziona come il metodo *get()* con la differenza che *setdefault* crea un nuovo elemento nel dizionario se questo non esiste.

  Esempio:
  ```
  >>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
  >>> a.setdefault('a', 'Vuoto')
  1
  >>> a.setdefault('e', 'Vuoto')
  'Vuoto'
  >>> a
  {'a': 1, 'c': 3, 'b': 2, 'e': 'Vuoto', 'd': [1, 2]}
  ```

- *update(dizionario)* aggiorna i dati di un dizionario con gli elementi di un altro. Quelli già presenti verranno aggiornati, gli elementi che non esistono verranno aggiunti.

  Esempio:
  ```
  >>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
  >>> b = {'a':5, 'b':3, 'c':2, 'd':[1,2], 'e':'Nuovo'}
  >>> a.update(b)
  >>> a
  {'a': 5, 'c': 2, 'b': 3, 'e': 'Nuovo', 'd': [1, 2]}
  ```

- *values()* restituisce una lista contenente i valori di tutti gli elementi presenti nel dizionario.

  Esempio:
  ```
  >>> a = {'a':1, 'b':2}
  >>> print a.values()
  [1, 2]
  ```

- *itervalues()* è utile all'interno di un ciclo per scorrere i valori degli elementi di un dizionario.

  Esempio:
  ```
  >>> a = {'a':1, 'b':2, 'c':3, 'd':[1,2]}
  >>> for valore in a.itervalues():
  ...  print 'Valore: ' + valore
  ...
  Valore: 1
  Valore: 2
  Valore: [1,2]
  Valore: 3
  ```

2.7.2 Altri comandi per la gestione dei dizionari

- *dict(lista)* permette di definire un dizionario partendo da una lista formata da tuple contenenti una coppia *chiave : valore*.
  Per valorizzare la lista è possibile utilizzare il *list comprehension*.

  Esempio:

```
# Definizione semplice:
>>> l = [('a', 1), ('b', 2), ('c', 3)]
>>> dict(l)
{'a':  1, 'c':  3, 'b':  2}
>>>
# Definizione attraverso list comprehension:
>>> l = [(x,x*2) for x in range(0,4)]
>>> print l
[(0, 0), (1, 2), (2, 4), (3, 6)]
>>> dict(l)
{0:  0, 1:  2, 2:  4, 3:  6}
```

Capitolo 3

Le variabili

3.1 Dichiarazione

Le variabili in python **non vengono dichiarate esplicitamente**. Una variabile viene autodichiarata quando viene utilizzata. Il tipo assegnato alla variabile dipende dal tipo del valore col quale viene valorizzata la variabile. Una variabile può anche cambiare il proprio tipo.

Esempio:

```
>>> a = 3 # La variabile è numerica
>>> a
3
>>> a = '2' # La variabile diventa stringa!
>>> a
'2'
```

3.2 Valorizzazione

Per valorizzare una variabile in Python si utilizza l'operatore = (uguale).
E' possibile valorizzare più di una variabile su di una stessa riga di codice. Questa tecnica viene chiamata *assegnazione multipla*.

Esempio:

```
>>> a, b = 0, 1
>>> a
0
>>> b
1
```

Capitolo 4

Controllo di flusso

Le istruzioni di controllo di flusso permettono di eseguire porzioni di codice al verificarsi di determinate condizioni.

4.1 If

L'istruzione *if*, comune a moltissimi linguaggi di programmazione, permette di eseguire una volta un blosso di codice al verificarsi di una condizione. La sua sintassi è la seguente:

```
if [espressione] :
  [istruzioni]
elif [espressione]:
  [istruzioni]
else:
  [istruzioni]
```

Se la prima espressione è vera, verranno eseguite le righe di codice inserite tra il simbolo : (due punti) ed il primo *elif* o *else* che segue. Se nessun *elif* o *else* è definito nell'*if* verranno eseguite tutte le istruzioni prima del termine dell'istruzione *if*.
Se le seconda espressione è vera, verranno eseguite le righe di codice inserite tra il simbolo : (due punti) ed il primo *elif* o *else* che segue o il termine dell'istruzione *if*. Se nessuna delle condizioni precedenti è vera, verranno eseguite le righe di codice inserite tra il simbolo : (due punti) ed il termine dell'istruzione *if*.

Esempio:

```
>>> a=5
>>> if a==3:
...    print 'a è uguale a tre'
...  elif a==4:
...    print 'a è uguale a quattro'
```

```
...  elif a==5:
...    print 'a è uguale a cinque'
...  else:
...    print 'a è un altro numero'
a è uguale a cinque
```

Utilizzando l'istruzione *elif* è bene fare attenzione al fatto che l'istruzione *if*, quando trova una condizione vera ed esegue una porzione di codice, salta al termine dell'intera istruzione *if*, senza eseguire controlli sulle condizioni successive.

Esempio:

```
>>> a=3
>>> if a==3:
...    print 'tre'
...  elif a==3:
...    print 'ancora tre'
...
tre
```

Dal precedente esempio è evidente che una volta eseguita la prima istruzione vera (*print 'tre'*) il controllo delle condizioni si interrompe ed il controllo passa alla prima istruzione successiva all'*if*.

4.2 While

L'istruzione *while* permette di eseguire un blocco di istruzioni finché una condizione è verificata. La sua sintassi è:

```
while [espressione]:
  [istruzioni]
  [break]
  [continue]
else:
  [istruzioni]
```

Finché l'espressione è vera, verranno eseguite le righe di codice comprese tra il simbolo : (due punti) e l'istruzione *else* (o il termine dell'istruzione *while* se nessun *else* è definito).
Le istruzioni comprese tra l'*else* e la fine dell'istruzione *while* vengono eseguite quando la condizione è falsa (sia che lo sia la prima volta che viene eseguito il ciclo *while*, sia che lo diventi durante il ciclo).

Esempio:

```
>>> a, b = 0, 5
>>> while a < b:
```

```
...     a = a + 1
...     print 'Valore di a = ' + str(a)
...   else:
...     print 'Altrimenti'
...
Valore di a = 1
Valore di a = 2
Valore di a = 3
Valore di a = 4
Valore di a = 5
Altrimenti
```

Si può notare come l'istruzione definita nel blocco *else* sia stata eseguita una volta che l'espressione è divenuta falsa.

Esempio:

```
>>> a, b = 0, 5
>>> while a > b:
...     a = a + 1
...     print 'Valore di a = ' + str(a)
...   else:
...     print 'Altrimenti'
...
Altrimenti
```

In questo esempio è possibile notare come le istruzioni del blocco *else* vengano eseguite se l'espressione è falsa alla prima esecuzione del ciclo *while*.

All'interno di un ciclo *while* è possibile utilizzare le istruzioni *break* e *continue* per modificare il funzionamento del ciclo durante la sua esecuzione.

- *break* permette di interrompere l'esecuzione del ciclo passando il controllo alla prima riga che segue l'istruzione *while*. In questo modo si evita di eseguire il blocco di codice *else*.
- *continue* permette di saltare le restanti righe del blocco che si sta eseguendo ed il controllo ritorna alla prima riga del ciclo *while* (la quale testa la condizione).

  Esempio:

```
>>> a, b = 0, 5
>>> while a < b:
...     a = a + 1
...     print 'Valore di a = ' + str(a)
...     if a == 3:
...       break
...     else:
```

```
...       continue
...   else:
...     print 'Altrimenti'
...
Valore di a = 1
Valore di a = 2
Valore di a = 3
```

4.3 For

Il ciclo *for* permette di eseguire un blocco di istruzioni un numero finito di volte. La sua sintassi è:

```
for [lista_target] in [espressione]:
  [istruzioni]
  [break]
  [continue]
else:
  [istruzioni]
```

L'espressione dovrebbe essere una sequenza. Il ciclo *for* verrà eseguito una volta per ogni elemento della sequenza, ordinando gli elementi in senso ascendente secondo il loro indice.
Durante ogni esecuzione del ciclo l'elemento della sequenza viene trasferito nella *lista_target*.
Quando viene raggiunto il termine della sequenza viene eseguito il blocco di istruzioni *else* e quindi il ciclo termina.
Come nel ciclo *while* anche qui è possibile utilizzare i comandi *break* e *continue* per controllare il comportamento del ciclo.

Esempio:

```
>>> a=['a', 'b', 'c', 'd']
>>> for x in a:
...     print x
...   else:
...     print 'Fine'
...
a
b
c
d
Fine
```

Utilizzando i cicli *for* è necessario fare molta attenzione alle modifiche che vengono fatte all'interno del ciclo alla sequenza utilizzata come espressione. Python quando inizia un ciclo del *for* si salva l'indice del prossimo elemento che verrà processato. Se durante l'esecuzione del ciclo viene rimosso dalla sequenza l'elemento in uso o uno precedente l'elemento successivo verrà saltato. Questo avviene perché cancellando un elemento gli indicidi tutti gli elementi successivi diminuiscono di uno.

Esempio:

```
>>> a=['a', 'b', 'c', 'd']
>>> for x in a:
...     print x
...     a.remove(x)
...   else:
...     print 'Fine'
...
a
c
Fine
```

Per evitare questo problema è possibile utilizzarfe una copia della sequenza per il ciclo *for* in modo che le modifiche alla sequenza non ne influenzino il funzionamento. Nell'esempio seguente vengono eliminati gli elementi della sequenza *a*, ma il ciclo *for* procede correttamente.

Esempio:

```
>>> a=['a', 'b', 'c', 'd']
>>> for x in a[:]:
...     print x
...     a.remove(x)
...   else:
...     print 'Fine'
...
a
b
c
d
Fine
>>> a
[]
```

Per utilizzare il ciclo *for* numerico (simile a quello che si trova in molti linguaggi di programmazione) è possibile utilizzare la funzione *range()* che restituisce una sequenza di numeri. La sua sintassi è:

```
range(valore_minimo, valore_massimo, passo)
```

Il *valore_minimo* è il numero dal quale far partire la serie, il *valore_massimo* indica a quale numero la serie deve interrompersi, il *passo* indica la quantità che viene aggiunta tra un valore e l'altro e deve essere un numero intero.

Esempio:

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

I due comandi restituiscono la stessa sequenza di numeri in quanto il default del *passo* è 1.
E' possibile utilizzare anche valori negativi per il *passo*:

Esempio:

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

Combinando la sintassi del ciclo *for* con la funzione *range* si possono ottenere cicli numerici.

Esempio:

```
>>> for a in range(1,5,1):
...     print a
...
1
2
3
4
```

4.4 Try

L'istruzione *try* viene utilizzate per intercettare gli errori (una analisi più dettagliata della gestione degli errori viene affrontata nel capitolo 8 nella pagina 63).Permette di eseguire un blocco di istruzioni al verificarsi di un determinato errore. Esistono due sintassi per questo comando.
La prima sintassi è:

```
try:
  [istruzioni]
except [(errore1, errore2,...), target]:
  [istruzioni]
else:
  [istruzioni]
```

- L'istruzione *try* tenta di eseguire il blocco di codice.

- Se si verifica un errore viene cercato nella collezione di *except* un blocco atto a gestire l'errore che si è verificato.
- Se non esiste tale blocco viene generato un errore standard.
- Se esiste un blocco specifico per l'errore verificatosi, allore viene eseguito il blocco associato all'*except* e quindi il controllo passa al termine dell'istruzioner *try*.
- Se l'istruzione protetta da *try* non restituisce un errore viene eseguito (se definito) il codice del blocco *else* e l'istruzione *try* termina.

Una istruzione *except* può intercettare più errori contemporaneamente, è sufficiente separare con una virgola gli errori da gestire.

Gli errori che è possibile intercettare sono:
ArithmeticError, AssertionError, AttributeError, EOFError, EnvironmentError, FloatingPointError, IOError, ImportError, IndentationError, IndexError, KeyError, LookupError, MemoryError, NameError, NotImplementedError, OSError, OverflowError, ReferenceError, RuntimeError, StandardError, SyntaxError, SystemError, TabError, TypeError, UnboundLocalError, UnicodeDecodeError, UnicodeEncodeError, UnicodeError, UnicodeTranslateError, ValueError, ZeroDivisionError

Esempio:

```
>>> a = 'prova'
>>> a = a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError:  cannot concatenate 'str' and 'int' objects
>>> a = 'prova'
>>> try:
...     a = a + 1
...   except TypeError:
...     print 'Non si può!'
...   else:
...     print 'Fatto'
...
Non si può!
>>> a = 5
>>> try:
...     a = a + 1
...   except TypeError:
...     print 'Non si può!'
...   else:
...     print 'Fatto.  a = ' + str(a)
...
```

```
Fatto.  a = 6
```

È possibile scrivere una istruzione *except* senza definire quali errori deve intercettare. In questo modo l'istruzione intercetterà tutti gli errori. Questo costrutto può essere utile come ultima istruzione *except* di una lista. Si deve fare attenzione nell'intercettare tutti gli errori perché così facendo potrebbero venire nascosti errori reali.

Esempio:

```
>>> try:
...     x = int('aa')
...   except NameError:
...     print "NameError"
...   except:
...     print "Altro Errore"
...
Altro Errore
```

Dopo il nome dell'exception da gestire è possibile indicare una variabile *target* che viene definita di tipo *exception* e contiene informazioni sull'errore che si è verificato.
L'oggetto *target* può fornire la descrizione dell'errore e la lista dei parametri che sono stati passati all'errore:

Esempio:

```
>>> try:
...     x = int("aa")
...   except ValueError, dettaglio:
...     print dettaglio
...     print dettaglio.args
...   print dettaglio.__doc__
...
invalid literal for int():  aa
('invalid literal for int():  aa',)
'Inappropriate argument value (of correct type).'
```

Senza indicare nessuna proprietà *dettaglio* restituisce il testo dell'errore che si è verificato.
args contiene i parametri dell'errore (il primo parametro è solitamente pari alla descrizione dell'errore).
La proprietà *__doc__* contiene una descrizione della possibile causa dell'errore.

La seconda sintassi dell'istruzione *try* è:

```
try:
  [istruzioni]
finally:
  [istruzioni]
```

Il funzionamento di questa seconda sintassi è leggermente differente da quello precedente:

- L'istruzione *try* tenta di eseguire il blocco di codice.
- Se non viene generato nessun errore viene eseguito il blocco di codice *finally* e il controllo passa al termine dell'istruzione *try*..
- Se viene generao un errore viene eseguito il blocco di codice *finally*.
- Se il blocco di codice *finally* genera un errore, contiene un *break* o un *return* l'errore iniziale della *try* viene perso.

Nel blocco di istruzioni *finally* è illegale l'uso di *continue*.

4.5 Gli iteratori

Quando si scrive una istruzione *for..in* per eseguire un ciclo su un oggetto (come una lista o una stringa) Python richiama in realtà la funzione *iter()* per calcolare l'iteratore del ciclo.
Questa funzione può essere utilizzata anche manualmente e, restituendo un oggetto di tipo *iterator*, mette a disposizione il metodo *next* che permette di spostarsi sull'elemento successivo dell'iteratore.
Quando viene raggiunto il termine dell'iteratore viene generato un errore *StopIteration*.

Esempio:

```
>>> lista = [1, 2, 'a', 3, 'd']
>>> i = iter(lista)
>>> i.next()
1
>>> i.next()
2
>>> i.next()
'a'
>>> i.next()
3
>>> i.next()
'd'
>>> i.next()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
StopIteration
```

4.6 I generatori

Un generatore è una funzione che genera un iteratore. Utilizza al suo interno l'istruzione *yield* (che può essere utilizzata solo all'interno di una funzione) ogni volta che deve restituire un valore. Ogni volta che il metodo *next()* viene richiamato, il generatore ricomincia l'elaborazione dall'ultimo elemento elaborato.

Esempio:

```
>>> def generatore(lista):
...     for l in lista:
...       yield l
...
>>> i = generatore([1,5,7,2,8])
>>> i.next()
1
>>> i.next()
5
>>> i.next()
7
>>> i.next()
2
>>> i.next()
8
```

Capitolo 5

Lettura e scrittura di files

5.1 Apertura di un file

Python ha una classe *file* espressamente dedicata alla lettura ed alla scrittra di files in formato testo.

Per aprire un file viene utilizzata l'istruzione *open()*. La sua sintassi è:

```
open(nome_file, modalità)
```

L'istruzione restituisce un oggetto file aperto in una specifica *modalità*.
Le *modalità* supportate sono:

- *r* file in sola lettura. Se il file non esiste viene generato un errore.
- *w* file in sola scrittura. Un file esistente con lo stesso nome viene sovrascritto.
- *a* file in scrittura (append). Se il file esiste già i dati vengono aggiunti ad esso.
- *r+* file in lettura/scrittura

Se non viene specificata una *modalità* viene assunto *r* come default.

Esempio:

```
>>> file = open('/tmp/prova', 'w')
>>> file = open('/tmp/prova', 'r+')
>>> file = open('/tmp/prova2', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError:  [Errno 2] No such file or directory:  '/tmp/prova2'
```

5.2 Metodi e proprietà per i files

A parte l'apertura del file, che va eseguita con una istruzione *open()*, tutte le altre possibilità di gestione del file sono messe a disposizione dai suoi metodi.

- *close()* se il file è aperto lo chiude, altrimenti non fa nulla (senza generare errori). Imposta la proprietà *closed* a True.
- *closed* proprietà bolleana che restituisce True se il file è chiuso o False se è aperto.

  Esempio:
  ```
  >>> file = open('/tmp/prova', 'r')
  >>> file.closed
  False
  >>> file.close()
  >>> file.closed
  True
  ```

- *fileno()* restituisce un numero intero che identifica il file (file-descriptor). Genera un errore se il file è chiuso.
- *flush()* scrive il contenuto del buffer in memoria su disco.
- *isatty()* restituisce True se il file è connesso ad un dispositivo tty, False se non lo è.
- *mode* proprietà contenente la modalità di apertura del file (la stessa *modalità* utilizzata nell'istruzione *open()*)

  Esempio:
  ```
  >>> file = open('/tmp/prova', 'r')
  >>> file.mode
  'r'
  ```

- *name* proprità che restituisce il nome del file (lo stesso utilizzato nella istruzione *open()*)

  Esempio:
  ```
  >>> file = open('/tmp/prova', 'r')
  >>> file.name
  '/tmp/prova'
  ```

- *read([bytes])* legge da un file aperto un numero di *bytes* e lo restituisce in formato stringa. Se non viene specificato un numero di *bytes* viene letto tutto il file! La lettura avviene sequenzialmente.

Esempio:

```
>>> file = open('/tmp/prova', 'w')
>>> file.write('prova di lettura/scrittura')
>>> file.close()
>>> file = open('/tmp/prova', 'r')
>>> file.read(3)
'pro'
>>> file.read(13)
'va di lettura'
>>> file.read(5)
'/scri'
>>> file.read(15) # Legge fino a EOF (End Of File)
'ttura'
>>> file.read(5) # EOF
''
```

- *readline([bytes])* legge un numero massimo di *bytes* da una riga del file e la restituisce in formato testo. Se non viene specificato un numero di *bytes* viene restituita l'intera riga. Se si è raggiunta la fine del file *readline()* restituisce una stringa vuota.
  Una riga vuota all'interno del file non è una stringa vuota, in quanto contiene il carattere $\backslash n$ di fine riga.

  Esempio:

```
>>> file = open('/tmp/prova', 'w')
>>> file.write('prima riga\n')
>>> file.write('seconda riga\n')
>>> file.write('\n')
>>> file.close()
>>> file = open('/tmp/prova', 'r')
>>> file.readline()
'prima riga\n'
>>> file.readline(5)
'secon'
>>> file.readline()
'da riga\n'
>>> file.readline()
'\n'
>>> file.readline() # EOF
''
```

- *readlines([bytes])* legge una o più righe fino al raggiungimento del numero di *bytes* e le restituisce in una lista. Restituisce solo righe complete e quindi il numero di *bytes* restituito dal metodo potrebbe essere maggiore di quello impostato. Se non viene definito un numero di *bytes* viene letto tutto il file.

Esempio:

```
>>> file = open('/tmp/prova', 'w')
>>> file.write('prima riga\n')
>>> file.write('seconda riga\n')
>>> file.write('terza riga\n')
>>> file.write('\n')
>>> file.close()
>>> file = open('/tmp/prova', 'r')
>>> file.readlines()
['prima riga\n', 'seconda riga\n', 'terza riga\n', '\n']
```

- *seek(bytes [, posizione_iniziale])* posiziona il puntatore in un certo punto del file. Per calcolare il punto di arrivo viene aggiunto un numero di *bytes* ad una *posizione_iniziale*. La *posizione_iniziale* può avere i valori: 0 (che è il valore di default) per indicare l'inizio del file, 1 per indicare la posizione corrente, 2 per indicare la fine del file. *seek(0)* posiziona il puntatore all'inizio del file.

  Esempio:

```
>>> file = open('/tmp/prova', 'r+')
>>> file.write('1234567890abcdefghijklmnopqrstuvz')
>>> file.seek(5) # Si posiziona al sesto carattere
>>> file.read(1) # Legge il sesto e si posiziona sul settimo.
'6'
>>> file.seek(1,1) # Avanza di un carattere (8)
>>> file.read(1)
'8'
>>> file.seek(-2,1) # Torna indietro di due caratteri.
>>> file.read(1)
'7'
>>> file.seek(-3,2) # Si posiziona al terzultimo carattere.
>>> file.read(1)
'u'
```

- *tell()* restituisce un valore numerico rappresentante la posizione attuale all'interno di un file aperto.

  Esempio:

```
>>> file = open('/tmp/prova', 'r+')
>>> file.write('1234567890abcdefghijklmnopqrstuvz')
>>> file.seek(5) # Si posiziona al sesto carattere
>>> file.tell()
5L
```

- *truncate([bytes])* riduce le dimensioni di un file ad un numero specificagto di *bytes*. Se non viene specificato un numero di *bytes* viene utilizzata la posizione corrente nel file (lo stesso numero restituito dal metodo *tell()*).

Esempio:

```
>>> file = open('/tmp/prova', 'r+')
>>> file.write('1234567890abcdefghijklmnopqrstuvz')
>>> file.truncate(5)
>>> file.read()
'12345'
```

- *write(stringa)* scrive una *stringa* in un file aperto. Prima che il file venga effettivamente scritto su disco potrebbe essere necessario eseguire il metodo *flush()* oppure *close()*.

  Esempio:

  ```
  >>> file = open('/tmp/prova', 'r+')
  >>> file.write('1234567890abcdefghijklmnopqrstuvz')
  >>> file.close()
  ```

- *writelines(sequenza_di_stringhe)* scrive tutti gli elementi contenuti nella *sequenza_di_stringhe*. Ottiene lo stesso effetto che richiamare il metodo *write()* per ogni elemento della *sequenza_di_stringhe*. Da notare che non vengono aggiunti automaticamente il carattere di fine riga $\backslash n$.

  Esempio:

  ```
  >>> file = open('/tmp/prova', 'r+')
  >>> file.writelines(['a','b','c','d'])
  >>> file.seek(0)
  >>> file.read()
  'abcd'
  ```

5.3 Il modulo *pickle*

I metodi per leggere e scrivere file finora descritti funzionano perfettamente se si cerca di leggere o scrivere dati di tipo stringa. Se però dovessimo leggere o salvare dati differenti (come liste, dizionari, ...) la cosa potrebbe essere più complicata.
Per risolvere questo problema esiste il modulo *pickle* che è studiato appositamente per convertire qualsiasi oggetto Python in una sua rappresentazione in formato stringa. Il modulo permette anche l'operazione inversa, ovviamente.

Le funzione del modulo *pickle* per scrivere un oggetto su file è:

```
dump(oggetto, file [, protocollo])
```

I parametri della funzione sono:

- *oggetto* indica l'oggetto (lista, dizionario, ...) che si vuole scrivere su *file.*
- *file* oggetto file dove si intende scrivere.

- *protocollo* è un parametro opzionale (il suo default è 0) che indica quale protocollo utilizzare per scrivere il file. I valori possibili sono: 0 protocollo ASCII Originale di Python (utile per la retrocompatibilità), 1 vecchio protocollo binario mantenuto per retrocompatibilità, 2 nuovo protocollo introdotto in Python 2.3

La funzione del modulo *pickle* per leggere un oggetto da un file è:

```
load(file)
```

La funzione accetta solo un oggetto *file* come parametro che indica il file dal quale leggere. La funzione legge un oggetto e lo restituisce.

Esempio:

```
>>> import pickle
>>> file = open('/tmp/prova', 'w')
>>> a = [1,2,3,4]
>>> b = {'a':1, 'b':2}
>>> pickle.dump(a, file)
>>> pickle.dump(b, file)
>>> file.close()
>>> file = open('/tmp/prova', 'r')
>>> pickle.load(file)
[1, 2, 3, 4]
>>> pickle.load(file)
{'a':  1, 'b':  2}
```

Per ottenere la rappresentazione di tipo stringa di un oggeto senza scriverla in un file esistono due funzioni simili alle precedenti:

```
dumps(oggetto [, protocollo])
```

e

```
loads(stringa)
```

Esempio:

```
>>> a = [1,2,3,4]
>>> b = {'a':1, 'b':2}
>>> pickle.dumps(a)
'(lp0\nI1\naI2\naI3\naI4\na.'
>>> pickle.dumps(b)
"(dp0\nS'a'\np1\nI1\nsS'b'\np2\nI2\ns."
>>> pickle.loads('(lp0\nI1\naI2\naI3\naI4\na.')
[1, 2, 3, 4]
>>> pickle.loads("(dp0\nS'a'\np1\nI1\nsS'b'\np2\nI2\ns.")
```

```
{'a': 1, 'b': 2}
```

Capitolo 6

Le funzioni

6.1 Definizione di una funzione

Una funzione è un blocco di codice al quale viene assegnato un nome, utilizzato per far riferimento ad esso. Una funzione può restituire un valore (ed in questo caso è una funzione a tutti gli effetti), oppure no (in questo caso si potrebbe parlare più di procedura che di funzione).

Una funzione può ricevere dei parametri nel momento in cui viene richiamata, oppure non avere alcun parametro.
La sintassi per definire una funzione è:

```
def nome_funzione([lista parametri]):
 """docstring"""
  [istruzioni]
```

La *docstring* (documentation string) non è altro che un commento scritto in testa alla funzione, racchiuso da tre doppi apici (") , che ne descrive brevemente l'uso. La *docstring* non è obbligatoria dal punto di vista sintattico, ma è buona abitudine inserirla in ogni funzione. Alcuni programmi di documentazione leggono le *docstring* per aiutare il programmatore nella stesura dei documenti.
La *docstring* può essere visualizzata digitando *nome_funzione.__doc__*

Esempio:

```
>>> def numeri(n):
...     """Stampa i numeri da uno a n"""
...     for a in range(1,n+1,1):
...       print a
...
>>> numeri(5)
1
2
```

```
3
4
5
>>> numeri.__doc__
'Stampa i numeri da uno a n'
```

La funzione sopra riportata non restituisce alcun valore, ma accetta un parametro in ingresso (n).
Il parametro viene utilizzato all'interno della funzione per stampare i numeri compresi tra uno ed il parametro stesso.

Una volta definita una funzione, il suo nome viene riconosciuto da Python come il nome di una funzione-utente e può essere utilizzato per dichiarare un altro nome per la funzione.

Esempio:

```
>>> num = numeri
>>> num(3)
1
2
3
```

Nell'esempio sopra riportato viene assegnato il nome *num* alla funzione *numeri*. Dal momento della definizio in poi richiamare *num* significa fare una chiamata alla funzione *numeri*.

6.2 I parametri nelle funzioni

Quando si richiama una funzione passandole uno o più parametri, questi vengono utilizzati dalla funzione *per valore*.
Questo significa che modificando il valore di un parametro all'interno di una funzione, la modifica sarà visibile solo all'interno della funzione. Una volta usciti dalla funzione il parametro avrà il valore che aveva quando la funzione è stata richiamata.

Esempio:

```
>>> def prova(p1, p2):
...     """Prova di modifica parametri in una funzione"""

...     p1="modificato dalla funzione"
...     p2="modificato dalla funzione"
...     print p1
...     print p2 ...
>>> p1="Parametro 1"
>>> p2="Parametro 2"
```

```
>>> prova(p1, p2)
modificato dalla funzione
modificato dalla funzione
>>> print p1
Parametro 1
>>> print p2
Parametro 2
```

Nell'esempio sopra riportato il valore dei parametri *p1* e *p2* viene modificato all'interno della procedura. Ne viene stampato il contenuto per controllare l'avvenuto aggiornamento del valore.
Una volta terminata la funzione viene stampato il contenuto delle variabili utilizzate come parametri, ed il valore non è stato cambiato. Le due variabili hanno il valore iniziale.

Per ogni parametro di una funzione è possibile definire un valore di default. In questo modo la funzione potrà essere richiamata anche senza che le vengano passati tutti i parametri. Per quelli non passati viene utilizzato, infatti, il valore di fefault.

Esempio:

```
>>> def numeri(n=3):
...     """Stampa i numeri da uno a n (default=3)"""
...     for a in range(1,n+1,1):
...       print a ...
>>> numeri()
1
2
3
>>> numeri(5)
1
2
3
4
5
```

I parametri possono essere passati alle funzioni utilizzando anche una sintassi del tipo *parametro = valore*. Utilizzando questa tecnica i parametri possono anche essere passati senza seguire l'ordine nel quale sono definiti nella funzione. E' possibile utilizzare il metodo posizionale e quello *parametro = valore* nella stessa chiamata, ma un parametro *parametro = valore* non può precedere un parametro posizionale.

Esempio:

```
>>> numeri(n=4)
1
```

```
2
3
4
```

In Python è possibile anche definire una funzione con un numero di parametri variabile. Per ottenere questo risultato è necessario utilizzare come parametro una variabile **nome_variabile* che contiene una lista dei parametri posizionali passati alla funzione.

Esempio:

```
>>> def n(*name):
...     """Prova parametri variabili"""
...     for a in range(0, len(name)):
...       print name[a]
...
>>> n('prova', 2)
prova
2
>>> n('prova', 2, 'terzo parametro')
prova
2
terzo parametro
```

Nelle funzioni è possibile utilizzare anche un altro parametro speciale: ***nome_variabile*. Esso contiene i parametri passati col metodo *parametro = valore* che non hanno una corrispondenza coi parametri della funzione.

Esempio:

```
>>> def n(a, *pos, **keys):
...     """Parametri * e **"""
...     for a in pos:
...       print 'Param.  pos.  :', a
...     for a in keys:
...       print 'Param.  :', a, '-', keys[a]
...
>>> n(a=2,prova='2', prova2=3)
Param.  :  prova - 2
Param.  :  prova2 - 3
>>> n(2,prova='2', prova2=3)
Param.  :  prova - 2
Param.  :  prova2 - 3
>>> n(2,5,prova='2', prova2=3)
Param.  pos.  :  5
Param.  :  prova - 2
Param.  :  prova2 - 3
```

Da questo esempio si può evincere che:

- **pos* è una lista.
- **pos* contiene solo i parametri posizionali che eccedono quelli definiti nella funzione. Infatti nella prima e nella seconda chiamata alla funzione il valore di *a* non viene stampato.
- ***keys* è un dizionario (infatti ha un valore alfanumerico come chiave invece che un numerico intero).
- ***keys* contiene tutti i parametri non posizionali passati alla funzione che non corrispondodo a nessun parametro definito nella funzione. Infatti nella prima chiamata alla funzione il valore di *a* non viene stampato.
- Nell'indice di ***keys* si trova il nome del parametro non posizionale.
- Nel valore di ***keys* si trova il valore del parametro non posizionale.

6.3 Funzioni che restituiscono un valore (l' istruzione *return*)

Le funzioni scritte sino ad ora non restituiscono nessun valore. Questo è verificabile attraverso questa semplice prova:

Esempio:

```
>>> def funzione(n):
...     """Prova istruzione return"""
...     for a in range(1,n+1,1):
...      print a ...
>>> funzione(3)
1
2
3
>>> a = funzione(3)
1
2
3
>>> print a
None
```

Richiamando la funzione in modo normale il comportamento è quello visto sino ad ora.
Scrivendo il comando *a = funzione(3)* la funzione viene eseguita, e il valore restituito viene assegnato alla variabile *a*. In questo caso *a* contiene *None*. Questo significa che la funzione non ha restituito nessun valore.

Per restituire un valore la funzione deve utilizzare l'istruzione *return* (che permette di restituire un qualsiasi tipo di dati).
Modificando leggermente la funzione d'esempio si può fare in modo che essa restituisca la serie di numeri in una lista anziché stamparli a video.

Esempio:
```
>>> def funzione(n):
...     """Prova istruzione return"""
...     risultato = []
...     for a in range(1,n+1,1):
...       risultato = risultato + [a]
...     return risultato
...
>>> a = funzione(3)
>>> print a
[1, 2, 3]
```

In questo modo si può vedere come la funzione restituisca come valore una lista contenente i numeri da uno a tre.

Scrivendo *return* non seguito da una variabile la funzione restituirebbe *None*.
Lo stesso valore viene restituito se la funzione giunge all'ultima riga di codice senza incontrare una istruzione *return* (come succede nelle funzioni che non hanno bisogno di restituire valori).
Nel momento in cui una funzione incontra una istruzione *return* la funzione esce restituendo il valore che segue *return*. Le righe di codice che seguono il *return* non verranno eseguite!

6.4 Funzioni *lambda*

In Python è possibile definire delle piccole funzioni anonime tramite il comando *lambda*.

Esempio:
```
def prova(n):
...     n = n + 5
...     return lambda x:  x + n
>>> prova(5)
<function <lambda> at 0x403b1d14>
```

In questo modo si definisce una funzione che non restituisce un valore, ma una funzione. Le ultime due righe dell'output dimostrano che, richiamando la funzione *prova*, viene restituita una funzione e non un valore.
Così facendo è possibile attribuire alla funzione restituita un altro nome e poi utilizzarla normalmente.

Esempio:

```
>>> f = prova(5)
>>> f(1)
11
>>> f(4)
14
```

Quando viene eseguita la prima riga, la funzione *prova* aggiunge 5 al parametro in entrata (che ha valore 5 nell'esempio) ottenendo un valore di 10 per la variabile n.
Quyando si richiama f viene richiamata non la funzione *prova* ma la funzione *lambda* ed il valore di n è 10. La funzione *lambda* esegue la somma del parametro ricevuto in entrata e ne restituisce il risultato.

Capitolo 7

I moduli

Un modulo Python non è altro che un file di testo contenente del codice con definizioni e istruzioni.

L'uso dei moduli permette di segmentare un programma grande consentendo di utilizzare una o più funzioni in più programmi senza dover copiare il codice da un programma all'altro.
La funzione che si intende riutilizzare verrà scritta in un file di testo e salvata con un nome file che termini per *.py*
Il file con estensione *py* è un modulo Python.

Supponiamo di salvare in un file di testo (chiamato *funzioni.py*) il seguente codice:

```
descrizione="Modulo di prova"
def somma(a,b):
  """Funzione che somma due numeri"""
  return a + b
def sottrazione(a,b):
  """Funzione che fa la sottrazione di due numeri"""
  return a - b
```

7.1 Importazione di un modulo

Prima di potere usare le funzioni contenute in un modulo è necessario importarlo nel programma .
L'importazione di un modulo avviene tramite il comando *import*.
La sua sintassi è:

```
  import nome [as nome_interno], nome [as nome_interno],
...
```

Per importare solo una o più funzioni da un modulo esiste una sintassi differente del comando *import*:

```
from nome_modulo import nome_funzione [as nome_interno]
```

Quando viene eseguito un comando di *import* avvengono le seguenti cose:

- Il file del modulo che si cerca di caricare viene cercato prima nella directory corrente, e poi nel path indicato nella variabile di ambiente PYTHONPATH. Se PYTHONPATH non è valorizzato o il file non viene trovato, allora il modulo viene cercato in un path standard dipendente dal sistema operativo (in ambiente UNIX normalmente è *.:/usr/local/lib/python*). Python per cercare un modulo usa internamente la variabile *sys.path* che è valorizzata con la directory dove si trova lo script di avvio di python, PYTHONPATH e la directory di default del sistema operativo.

  Esempio:

  ```
  >>> import sys
  >>> sys.path
  ['', '/usr/lib/python23.zip', '/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2',

  '/usr/lib/python2.3/lib-tk', '/usr/lib/python2.3/lib-dynload',

  '/usr/lib/python2.3/site-packages', '/usr/lib/python2.3/site-packages/Numeric',

  '/usr/lib/python2.3/site-packages/gtk-2.0']
  ```

- Una volta trovato il file vengono eseguite le righe di codice del modulo che non fanno parte di nessuna funzione (nell'esempio *funzioni.py* viene eseguita solo la prima riga)
- Vengono importate le funzioni del modulo

Proseguendo l'esempio del modulo *funzioni.py* è possibile importare il modulo ed utilizzarne le funzioni nel seguente modo:

```
>>> import funzioni
>>> funzioni.sottrazione(5,3)
2
>>> funzioni.somma(5,3)
8
>>> import funzioni as prova
>>> prova.somma(6,4)
10
```

In questo modo le funzioni del modulo sono accessibili con la forma *nome_modulo.nome_funzione*.
Per rendere più semplice l'uso delle funzioni è possibile assegnare loro un nome locale:

```
>>> somma = funzioni.somma
>>> somma(6,23)
29
```

Le variabili globali del modulo non interferiscono con le variabili utente:

```
>>> descrizione = 'Prova'
>>> funzioni.descrizione
'Modulo di prova'
>>> descrizione
'Prova'
```

Per aiutare la leggibilità dei programmi tutte le *import* vengono normalmente fatte tutte in testa al programma.

Quando Python riesce a caricare un modulo correttamente genera un file con estensione *pyc* contenente la versione "compilata" del modulo.
Importando un modulo viene prima ricercato il file *pyc* (in quanto il suo caricamento in memoria è molto più veloce rispetto al tempo di caricamento di un file *py*) e solo se non viene trovato (oppure se il file *py* è stato modificato dal momento della generazione del file *pyc*) viene caricato il file non copilato *py*.

7.2 Il comando *dir()*

Il comando *dir()* permette di avere una lista dei nomi (proprietà, metodi, ...) definiti nella sessione corrente oppure all'interno di un modulo.
La sua sintassi è:

```
dir([nome_modulo])
```

Richiamando il comando *dir()* senza indicare un nome di modulo, verranno elencati i nomi dichiarati nella sessione corrente. Verranno elencate le funzioni definite ed i moduli importati.
Passando al comando *dir()* il nome di un modulo caricato si otterrà invece la lista dei nomi definiti all'interno del modulo.

Esempio:

```
>>> import funzioni
>>> dir()
['__builtins__', '__doc__', '__name__', 'funzioni']
>>> dir(funzioni)
['__builtins__', '__doc__', '__file__', '__name__', 'descrizione',
'somma', 'sottrazione']
```

7.3 Packages

I packages servono a raggruppare più moduli all'interno di una struttura gerarchica, rendendo possibile una migliore organizzazione dei moduli.
Per creare un package è sufficiente:

- Creare una sottodirectory all'interno di una delle directory dove Python cerca i moduli.
- Scrivere un file *__init__.py* (obbligatorio) dove è possibile inserire istruzioni di inizializzazione del package.
- Creare una struttura di directory gerarchiche dove posizionare i moduli veri e propri.
- Porre in ogni sottodirectory un file *__init__.py*

In questo modo è possibile importare i moduli o le funzioni con una sintassi *package.modulo.funzione*

Supponiamo di avere una struttura di directory simile a:

```
pack/
   __init__.py
   gruppo1
     __init__.py
     modulo1.py
   gruppo2
     __init__.py
     modulo2.py
```

In questo esempio, e ipotizzando che nel *modulo1* e nel *modulo2* sia presente una funzione chiamata *funzione* è possibile importare i moduli nei seguenti modi:

Esempio:
```
# Importazione di una funzione:
>>> import pack.gruppo1.funzione
```

Nel modo appena esemplificato, però, è necessario riferirsi alla funzione con tutto il percorso per raggiungerla (*pack.gruppo1.funzione*).
Per evitare questo, è possibile importare la funzione con il costrutto *from..import*

Esempio:
```
>>> from pack.gruppo1 import funzione
```

In questo modo sarà possibile far riferimento alla funzione attraverso il solo nome, senza dover indicare il suo percorso.

7.3.1 Importare * da un package

Importare tutti i sottomoduli e tutte le funzioni di un package non è normalmente una bella idea. Per rendere più leggibile un programma e per risparmiare tempo di avvio e memoria sarebbe bene importare solo i moduli e le funzioni che effettivamente vengono ustilizzate al suo interno.
É comunque possibile importare più oggetti contemporaneamente da un package.

Vista la natura multipiattaforma di Python e le diferenze nella gestione del filesystem dei diversi sistemi oprativi sui quali gira, non è semplice importare tutti gli elementi di un package (questo soprattutto perché sistemi operativi come Windows non gestiscono informazioni dettagliate sui nomi di files con lettere maiuscole e minuscole, rendendo difficile la ricerca automatica dei file che fanno parte di un package) .
Per poter ovviare a questo problema è sufficiente definire nel file *__init__.py* del package una variabile *__all__* contenente i nomi dei moduli da caricare quando viene dato un *import **

Una variabile *__all__* così definita:

```
__all__ = ["modulo1", "modulo2"]
```

in un file *__init__.py* di un package fa in modo che, inserendo il comando

```
from package import *
```

verranno importati i moduli, contenuti nel package, *modulo1* e *modulo2*.
Questo metodo può essere utile quando in molti programmi vengono importate tutte (o molte) funzioni di uno stesso package, però prevede la manutenzione della variabile *__all__* ogni volta che si aggiunge un modulo al package.

Capitolo 8

La gestione degli errori

Quando durante l'esecuzione di una istruzione Python si verifica un errore viene generato quello che viene definito *exception* il quale non sempre è fatale.
È possibile scrivere programmi in grado di gestire gli errori.

8.1 Intercettare gli errori

Per intercettare gli errori che si potrebbero verificare durante l'esecuzione di un programma si utilizza l'istruzione *try* la cui sintassi è riportata nella pagina 39. Nell'esempio seguente il programma continua a chiedere un input all'utente finché non viene inserito un numero intero.

Esempio:

```
>>> while True:
...     try:
...       x = int(raw_input("Inserire un numero:  "))
...     except ValueError:
...       print "Non hai inserito un numero valido."
...     else:
...       break
...
Inserire un numero:  dd
Non hai inserito un numero valido.
Inserire un numero:  1a
Non hai inserito un numero valido.
Inserire un numero:  12
```

L'istruzione *try* cerca di eseguire l'istruzione *int()*. Se ci riesce (se cioè l'utente ha inserito un numero) il programma prosegue e viene eseguita l'istruzione *breal* che esce dal ciclo *while* e termina il programma (in quanto dopo il ciclo *while* non c'è altro codice).

Se invece l'istruzione *int()* va in errore viene eseguito il codice definito per l'errore *ValueError* (che è quello che si verifica cercando di convertire in numero una stringa che non contiene un numero) ed il ciclo *while* ricomincia.

Quando si utilizza l'istruzione *try* è consigliabile scrivere nel blocco *try* solo l'istruzione che si intende controllare e scrivere il codice da eseguire in caso di successo dell'istruzione nel blocco *else*. Questo perché l'istruzione *try* intercetta gli errori di tutte le righe di codice scritte al suo interno, funzioni comprese. Questo significa che se in un blocco *try* viene richiamata una funzione esterna al blocco, anche questa sarà sotto la protezione dagli errori (e forse non era questo che si voleva ottenere).

8.2 Generare un errore standard

È possibile generare un errore programmativamente utilizzando l'istruzione *raise*. La sua sintassi è:

```
raise exception [, parametri]
```

In questo modo è possibile forzare il programma a generare un errore standard:

Esempio:

```
>>> raise ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError
>>> raise ValueError, "Testo di prova"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError:  Testo di prova
```

Se non viene passata nessuna *exception*, l'istruzione *raise* genera l'ultimo errore che si è verificato. Sfruttando questa caratteristica risulta semplice gestire un errore per stampare solo un messaggio e poi uscire dal programma nel seguente modo:

Esempio:

```
>>> try:
...     x = int("aa")
...  except ValueError:
...     print "Non si può fare!!"
...     raise
...
Non si può fare!!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
ValueError:  invalid literal for int():  aa
```

8.3 Definire un errore

È possibile creare dei nuovi errori da utilizzare all'interno di un programma con una istruzione *raise*. Definendo un nuovo errore spesso si utilizza un nome che termina per Error, per similitudine con gli errori standard.
Per fare questo si devono utilizzare le classi, che sonop analizzete in modo dettagliato nel capitolo 9 nella pagina 67.

Esempio:

```
>>> class MioErrore_01(Exception):
...     def __init__(self):
...       self.value = "E' un errore di prova"
...     def __str__(self):
...       return repr(self.value)
...  >>> raise MioErrore_01
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.Mio:  "E' un errore di prova"
```

In questo modo viene definito un errore dal nome *MioErrore_ 01* che può essere generato con una istruzione *raise*.

È possibile creare errori che ricevono dei parametri:

Esempio:

```
>>> class MioErrore_02(Exception):
...     def __init__(self, value):
...       self.value = value
...     def __str__(self):
...       return repr(self.value)
...
>>> raise MioErrore_02(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MioError:  2
```

Il parametro dell'errore è stato aggiunto al metodo $\_\_init\_\_$ della classe.

Le classi scritte per definire errori possono eseguire le operazioni che possono eseguire tutte le altre classi, ma normalmente la definizione di un errore è molto semplice in quanto serve a fornire informazioni sull'errore.

Quando si crea un modulo che definisce più errori è pratica comune definire una classe generica per gli errori di tutto il modulo e definire poi gli errori come derivati di questa classe.

Esempio:

```
>>> class Errore(Exception):
...     pass
...
>>> class MioErrore_01(Errore):
...     def __init__(self):
...       self.value = "E' un errore di prova"
...     def __str__(self):
...       return repr(self.value)
```

L'errore *MioErrore_ 01* funzionerà esattamente come prima.

Capitolo 9

Le classi

9.1 Definire una classe

Per definire una nuova classe la sintassi più semplice è la seguente:

```
class nome_classe:
 """docstring"""
  <istruzioni>
```

Quando viene definita una classe il suo nome viene aggiunto alla lista dei nomi attuale. Ne consegue che non è possibile far riferimento ad una classe prima che vengsa definita.

Esempio:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> class ProvaClasse:
...     """Descrizione"""
...     pass
...
>>> dir()
['ProvaClasse', '__builtins__', '__doc__', '__name__']
```

Anche nella definizione delle classi è presente la *docstring* utile per documentare il programma.
Tale descrizione è accessibile attraverso la proprietà $\_\_doc\_\_$

Esempio:

```
>>> ProvaClasse.__doc__
'Descrizione'
```

9.2 Attributi

Un attributo è un qualsiasi nome che viene definito all'interno di una classe.
Per far riferimento ad un attributo di una classe si utilizza la sintassi *nome_classe.nome_attributo*

Esempio:
```
>>> class Prova:
...     """Classe di prova"""
...     proprieta = 10
...     def funzione(self):
...       return "Metodo"
```

Nella classe appena dichiarata ci sono due attributi validi: *proprieta* e *metodo* che possono essere referenziati rispettivamente con *classe.proprieta* e *classe.funzione*. Anche *__doc__* è un attributo valido (anche se non è indicata alcuna descrizione), che restituisce la *docstring* della classe.

Esempio:
```
>>> a = Prova()
>>> a.proprieta
10
>>> a.funzione()
'Metodo'
```

Quando viene richiamato l'attributo *funzione* si nota che non viene passato nessun parametro alla funzione, che però ne mostra uno nella sua definizione. Questo avviene perché il parametro *self* viene aggiunto automaticamente da Python.

9.2.1 Attributi dati

Gli attributi dati (data attributes) sono valori che una istanza dell'oggetto valorizza e che possono essere utilizzati, appunto, come valori.
Nella classe del precedente esempio un attributo dati è *proprieta*.
Come le variabili, gli attributi dati esistono solo dopo che gli è stato assegnato un valore.
È normale attendersi quindi di trovare tutti gli attributi dati all'inizio della definizione di una classe.

Gli attributi dati di una classe possono essere creati anche dopo che la classe è stata istanziata. Per fare questo è sufficiente impostare l'attributo. È necessario porre molta attenzione a questa caratteristica.

Esempio:
```
>>> class prova:
...     a = 0
...     b = 'aa'
```

```
...
>>> a = prova()
>>> dir(a)
['__doc__', '__module__', 'a', 'b']
>>> a.a
0
>>> a.c
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError:  prova instance has no attribute 'c'
>>> a.c = 'testo'
>>> a.c
'testo'
```

Se da un lato questa caratterisgtica può essere pericolosa (basta sbagliare a digitare il nome di un attributo e questo viene creato, se si sta scrivendo una valorizzazione) può anche tornare utile in alcune circostanze.
Capita, per esempio, di avere la necessità di utilizzare una struttura a *record* è possibile sfruttare una classe vuota nel seguente modo:

Esempio:

```
>>> class vuota:
...    pass
...
>>> a = vuota
>>> a.campo_1 = 10
>>> a.campo_2 = 'testo'
>>> a.campo_3 = False
>>> a.campo_1
10
>>> a.campo_2
'testo'
>>> a.campo_3
False
```

9.2.2 Metodi

I metodi di una classe sono funzioni interne alla classe stessa.
Nella classe del precedente esempio un metodo è *funzione*.

Un metodo di una classe è anche un oggetto, e quindi può essere memotirzzato in una varfiabile per poter essere utilizzato senza dover scrivere il nome della classe.

Esempio:

```
>>> a.funzione()
```

```
'Metodo'
>>> prova = a.funzione
>>> prova()
'Metodo'
```

Un metodo può essere definito anche appoggiandosi ad una funzione già esistente e definita:

Esempio:

```
>>> def funzione(self,a,b):
...     return a + b
...
>>> class prova:
...     somma = funzione
...
>>> a = prova()
>>> a.somma(5,4)
9
```

All'interno di un metodo si può far riferimento agli altri metodi della classe attraverso la variabile *self*:

Esempio:

```
>>> class prova:
...     valore = 0
...     def somma(self, a):
...       self.valore = self.valore + a
...     def sommaduevolte(self, a):
...       self.somma(a)
...       self.somma(a)
...
>>> a = prova()
>>> a.somma(3)
>>> a.valore
3
>>> a.sommaduevolte(2)
>>> a.valore
7
```

9.2.2.1 Creare una classe col metodo *next()*

Il metodo *next()*, peculiare degli iteratori (vedere a pagina 42), può essere implementato in una classe in modo semplice.

- Definire una funzione $\_\_iter\_\_()$ nella classe

9.2.3 I nomi degli attributi

Scegliendo i nomi degli attributi (dati o metodi che siano) bisogna fare attenzione in quanto definendo due attributi con lo stesso nome solo l'ultimo attributo che è stato definito sarà visibile e valido.

Esempio:

```
>>> class prova:
...     def attributo(self):
...       print "Metodo"
...     attributo = 10
...
>>> a = prova()
>>> a.attributo
10
>>> a.attributo()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError:  'int' object is not callable
```

Nell'esempio riportato sopra l'attributo dati *attributo* sostituisce il metodo con lo stesso nome dichiarato precedentemente. Istanziando la classe sarà visibile solo l'attributo dati.

Esempio:

```
>>> class prova:
...     attributo = 10
...     def attributo(self):
...       print "Metodo"
...
>>> a = prova()
>>> a.attributo()
Metodo
>>> a.attributo
<bound method prova.attributo of <__main__.prova instance
at 0x403b7b8c>>
```

In questo esempio è invece il metodo *attributo* a sostituire l'attributo dati con lo stesso nome. Anche in questo caso solo l'ultimo attributo definito (cioè il metodo) sara valido.

Scoperto questo si rende necessario (ma soprattutto comodo) definire uno standard per scegliere i nomi degli attributi dati e dei metodi.
Si potrebbe decidere di scrivere i nomi degli attributi dati tutti in lettere masiuscole e lasciare le minuscole per i metodi, oppure utilizzare nomi per gli attributi dati e verbi per i metodi. L'importante è studiare uno standard che eviti di avere nomi uguali tra attributi dati e metodi.

9.3 Instanziare una classe

Per instanziare una classe si utilizza la sintassi che si utilizza per le funzioni (basta pensare alla classe come ad una funzione senza parametri che restituisce una istanza della classe) e cioè:

```
nome_classe()
```

Quindi per creare una nuova istanza della classe definita nell'esempio precedente:

```
>>> a = Prova()
>>> a.proprieta
10
>>> a.funzione()
'Metodo'
```

La variabile *a* è una istanza della classe *Prova* e quindi dispone dei suoi attributi.

9.3.1 Il metodo __init__

Quando si crea una nuova istanza di una classe, la classe cerca al suo interno un metodo chiamato *__init__* e, se lo trova, lo esegue.
Il metodo *__init__* viene eseguito ogni volta che si crea una istanza della classe, ed è quindi il posto migliore per eseguire tutte le operazioni di inizializzazione che una nuova istanza della classe richiede.

Esempio:

```
>>> class movimento:
...     "Classe che descrive un movimento"
...     velocita=0
...     direzione=None
...     def __init__(self):
...       self.direzione = "Nord"
...       self.velocita = 5
...     def CambiaVelocita(self, vel):
...       """Cambia la velocita di movimento"""
...       self.velovita = vel
...     def CambiaDirezione(self, dir):
...       """Cambia la direzione di movimento"""
...       self.direzione = dir
...     def Status(self):
...       """Stampa lo stato del movimento"""
...       print "Ti stai muovendo "
...       print "in direzione ", self.direzione
...       print "alla velocita' di ", self.velocita
```

La classe appena definita ha i seguenti attributi: *velocita*, *direzione*, *CambiaVelocita*, *CambiaDirezione*, *Status*, *__init__*.
Il metodo *__init__* inizializza la *velocita* e la direzione a dei valori di default. Ogni nuova istanza della classe avrà alla sua creazione i valori che vengono qui impostati.
I metodi *CambiaVelocita* e *CambiaDirezione* non fanno altro che impostare gli attributi dell'istanza stessa, facendo riferimento ad essa tramite il parametro *self*.

Esempio:

```
>>> mov = movimento()
>>> mov.velocita
5
>>> mov.direzione
'Nord'
>>> mov.CambiaVelocita(10)
>>> mov.CambiaDirezione("Sud")
>>> mov.Status()
Ti stai muovendo
in direzione Sud
alla velocita' di 10
```

Come tutte le funzioni, anche *__init__* può accettare dei parametri in entrata. Modificando il metodo *__init__* dellla classe *movimento* è possibile parametrizzare i valori iniziale di *velocita* e *direzione*:

Esempio:

```
...  def __init__(self, dir, vel):
...    self.direzione = dir
...    self.velocita = vel
```

Con questa semplice modifica è possibile passare alla classe *movimento* i valori di *direzione* e *velocita* nel momento in cui si istanzia la classe:

Esempio:

```
>>> mov = movimento("Est", 3)
>>> mov.velocita
3
>>> mov.direzione
'Est'
```

Il primo parametro, *self*, non viene mai passato esplicitamente. Viene aggiunto automaticamente in testa alla lista dei parametri di ogni funzione facente parte di una classe.

9.3.2 Il metodo __del__

Quando una istanza di una classe viene distrutta tramite il comando `del` viene automaticamente richiamato (se esiste) il metodo `__del__` della classe.
In questo metodo è quindi possibile eseguire tutte le operazioni che è necessario fare quando una istanza della classe viene distrutta.

9.4 Ereditarietà

L'eriditarietà permette di definire una classe facendola derivare da un'altra ereditandone gli attributi.
La sintassi per definire una classe derivata è:

```
class nome_classe(classe_base):
  <istruzioni>
```

La *classe_base* deve essere il nome valido di una classe nel momento in cui *nome_classe* viene definita.
Per definire *classe_base* è possibile anche utilizzare una espressione del tipo *modulo.classe_base*. Questo si rende necessario quando la *classe_base* si trova in un modulo differente.

Una classe derivata avrà a disposzione i propri attributi dati e metodi più quelli della sua classe base.

Esempio:

```
>>> class quadrupede:
...     nome = ""
...     zampe = 0
...     def __init__(self):
...       self.nome = "il quadrupede"
...       self.zampe = 4
...     def corre(self):
...       print self.nome, " corre."
...
>>> class cane(quadrupede):
...     def __init__(self):
...       self.nome = "il cane"
...     def ringhia(self):
...       print self.nome, " ringhia"
...
>>> c = cane()
>>> c.nome
'il cane'
>>> c.zampe
4
```

```
>>> c.corre()
il cane corre.
>>> c.ringhia()
il cane ringhia
```

Nell'esempio precedente la classe *cane* ha a disposizione l'attributo *zampe* anche se non è dichiarato (e nemmeno usato) al suo interno. L'attributo viene ereditato dalla classe base *quadrupede*. La classe cane aggiunge il metodo *ringhia* non presente nella classe base (non tutti i quadrupedi ringhiano!).
Quando si fa riferimento ad un attributo di una classe derivata, l'attributo viene prima cercato nella classe derivata, quindi nelle sua classe base (ed in modo ricorsivo nella successivsa classe base). Se l'attributo non viene trovato nemmeno nelle classi base allora viene generato un errore.

In Python è possibile anche utilizzare l'ereditarietà multipla. Questo significa che una classe derivsata può avere più di una classe base.
La sintassi è la seguente:

```
class nome_classe(classe_base1, classe_base2, ...)
 <istruzioni>
```

Definendo in questo modo una classe derivata, essa erediterò le proprietà ed i metodi di tutte le classi base indicate.
Quando viene richiamato un attributo della classe derivata, questo viene prima cercato nella classe derivata, successivamente nella *classe_base1* e nelle sue classi basi, la ricerca passa poi alla *classe_base2* e poi alla *classe_base3*.
Come è facile capire, abusare dell'ereditarietà multipla rende un programma molto meno leggibile. È consigliabile utilizzarla con cautela e solo quando si rende realmente necessario.