Università degli Studi di Roma “La Sapienza”

2003/10

Impadronirsi si Python

Sonia D’Ambrosio, Patrizia Mentrasti

DIPARTIMENTO DI MATEMATICA
“GUIDO CASTELNUOVO”

IMPADRONIRSI DI PYTHON

Accenniamo in questo breve scritto alle caratteristiche principali del linguaggio. Per tutto quello che qui non compare si faccia riferimento a [1].

Un po' di storia

Il linguaggio Python è stato creato ad Amsterdam nel 1989 dal ricercatore Guido Van Rossum. Il nome del linguaggio non ha niente a che spartire con il rettile, l'autore ne trovò ispirazione da una commedia trasmessa in televisione di nome "The monty Python's Flying Circus".

Nei suoi dieci anni di vita, si è diffuso in tutto il mondo. In Italia, la 'comunità Python' era veramente ristretta, tuttavia dal 1999 sembra che l'interesse verso questo linguaggio di programmazione stia crescendo sempre più.

Python è innanzitutto un linguaggio di script pseudocompilato. Questo significa che ogni programma sorgente deve essere pseudocompilato da un interprete.

Il principale vantaggio di questo sistema è la portabilità: lo stesso programma potrà girare su una piattaforma Linux, Mac o Windows purché vi sia installato l'interprete.

Python è un linguaggio orientato agli oggetti. Supporta le classi, l'ereditarietà e si caratterizza per il binding dinamico o meglio potremmo affermare che tutte le funzioni sono virtuali. La memoria viene gestita automaticamente e non esistono specifici costruttori o distruttori; inoltre esistono diversi costruttori per la gestione delle eccezioni.

L'interprete interattivo

Come nella migliore tradizione Unix, Python si presenta sotto forma di interprete interattivo a riga di comando. Quindi una volta digitata la parola "python" appare uno specifico prompt, nel caso di Python si tratta di tre caratteri di maggiore (>>>).

Basta digitare un comando e si ottiene subito una risposta. Per uscire dall'interprete e tornare al sistema operativo ospite basta premere **CTRL+D**.

Tipi di variabili

In Python non è necessario definire le variabili prima di utilizzarle, non è necessario nemmeno assegnare ad esse un tipo. Il tutto avviene implicitamente mediante l'istruzione di assegnamento (=).

Per visualizzare in output il valore di una variabile è sufficiente utilizzare il comando "print".

1 Numeri

In Python ci sono quattro tipi di numeri: interi, interi 'long', numeri in virgola mobile e numeri complessi. Per chiarezza riassumo i tipi di dati nella seguente tabella:

Tipo di dato	**Rappresentazione interna**	**Esempi**
Intero	32 bit (tipo long del C)	1200, -56, 0
Intero lungo	Oltre 32 bit con crescita in base alle esigenze	999999999L, -323232323L
Reale	32 bit (tipo double del C)	1.23, 3.1e-10, 4.0E210
Complesso	Coppia di numeri reali	3+4j, 3j, 5.0+4.1j

Le operazioni ammesse fra questi elementi sono:

Addizione	+
Sottrazione	–
Moltiplicazione	*
Divisione	/

Esponenziale	**
Modulo	%

Notiamo che le operazioni che coinvolgono due interi producono sempre un intero; le operazioni che interessano un numero in virgola mobile producono sempre un numero in virgola mobile. Le operazioni che coinvolgono un intero e un intero grande producono un intero grande.

Interi

Gli interi sono basati sui 'long' in C.

```
>>> x = 5 + 2 - 3 * 2
>>> x
⇒ 1
>>> 5 / 2
⇒ 2
```

La divisione tra interi risulta troncata.

```
>>> 5 % 2
⇒ 1
>>> 2**8
⇒ 256
```

Interi lunghi

Gli interi 'long' sono illimitati.

```
>>> 1000000001L ** 3
⇒ 1000000003000000030000000001L
```

Numeri in virgola mobile

I numeri in virgola mobile sono basati sui tipi double del C. Si possono usare interi grandi per simulare alcune operazioni in virgola mobile quando è necessario precisare l'occorrenza. Comunque, i numeri in virgola mobile sono più efficienti e veloci.

```
>>> x = 4.3 ** 2.4
>>> x
```

```
⇒ 33.1378473777
>>> 3.5e30 * 2.77e45
⇒ 9.69e+075
>>> 1000000001.0 ** 3
⇒ 1.000000003e+027
```

Numeri complessi

I numeri complessi sono creati automaticamente attraverso un'espressione della forma *nj* dove n ha la stessa forma degli interi o numeri in virgola mobile. *j* è la notazione standard per i numeri immaginari uguale alla radice quadrata di –1. Per esempio:

```
>>> (3 + 2j) ** (2 + 3j)
⇒ (0.681766519089-2.12074577662j)
>>> x = (3 + 2j) * (4 + 9j)
>>> x
⇒ (-6 + 35j)
```

Per estrarre la parte reale e quella immaginaria da un numero complesso x si usano x.real e x.imag.

```
>>> x.real
⇒ -6
>>> x.imag
⇒ 35.0
```

2 Liste

Una lista in Python è rappresentata come un array in C o in altri linguaggi. È una collezione ordinata di oggetti creata ponendo tra parentesi quadre gli elementi che ne fanno parte. I suoi elementi non devono essere necessariamente dello stesso tipo. Questo può essere un aspetto interessante che può risultare utile quando si sviluppano applicazioni RAD (Rapid Application Development), dove non si ha il tempo di creare strutture complesse per contenere dati di natura diversa.

Di seguito elenco una serie di esempi che illustrano le caratteristiche delle liste e gli operatori ad esse associate:

```
>>> x = [ 2, "two", [1, 2, 3]]
```

Una delle più importanti funzioni legate alle liste è la funzione `len` che restituisce il numero di elementi in una lista:

```
>>> x = [ "Questa", "lista", "ha", "5", "elementi"]
>>> len (x)
⇒ 5
```

Esiste un elemento particolare che rappresenta la lista vuota ([]):

```
>>> lista1 = []
>>> len (x)
⇒ 0
```

Gli elementi possono essere estratti da una lista usando le stesse notazioni utilizzate in C. Notiamo inoltre che come in C ed altri linguaggi Python inizia a contare da 0, quindi se chiediamo l'elemento 0 ci sarà dato il primo elemento della lista. Vediamo un piccolo esempio:

```
>>> x = ["primo", "secondo", "terzo", "quarto"]
>>> x[0]
⇒ 'primo'
>>> x[2]
⇒ 'terzo'
```

Comunque in Python gli indici sono più flessibili che in C se gli indici sono negativi essi indicano le posizioni di conteggio partendo dalla fine della lista, dove –1 sarà l'ultima posizione , -2 la penultima e così via.

```
>>> x[-1]
⇒ 'quarto'
>>> x[-2]
⇒ 'terzo'
```

Si ottiene uno "slice" usando [m:n], dove *m* è il punto iniziale e *n* è il punto finale. Con [:n] si indica uno 'slice' che inizia dal primo elemento della lista e si ferma ad *n*, mentre con la notazione [m:] si indice uno 'slice' che inizia da *m* e termina con l'ultimo elemento della lista.

```
>>> x[1:-1]
⇒['secondo', 'terzo']
>>> x[0:3]
⇒ [ 'primo', 'secondo', 'terzo']
```

Il modo migliore di tenere a mente come lavorano gli slice è di pensare che gli indici puntano tra i caratteri, con l'indice 0 posto al margine sinistro del primo carattere. Quindi il margine destro dell'ultimo carattere di una stringa di *n* caratteri ha indice *n*, per esempio:

```
x =      [   "primo" ,  "secondo" , "terzo" , "quarto"    ]
             ↑         ↑           ↑         ↑          ↑
indice positivo  0         1           2         3          4
indice negativo -4        -3          -2        -1
```

Le liste possono anche essere modificate ponendo semplicemente l'indice sul lato sinistro di un operatore di assegnamento:

```
>>> x = [1, 2, 3, 4]
>>> x[1] = "due"
>>> x
⇒ [1, "due", 3, 4]
```

Esiste un operatore assai originale che permette di concatenare in modo ripetitivo una lista (operatore "*"):

```
>>> Lista =[1, 2]
>>> lista_tot = Lista * 3
>>> lista_tot
⇒ [1, 2, 1, 2, 1, 2]
```

Per aggiungere un singolo elemento ad una lista si può usare la funzione `append`:

```
>>> x = [ 1, 2, 3]
>>> x.append ("quattro")
>>> x
⇒ [1, 2, 3, 'quattro']
```

Per modificare una lista può anche essere usato il metodo speciale `insert` che inserisce un nuovo elemento tra due elementi esistenti nella lista o all'inizio della lista. Tale metodo prende due argomenti addizionali; il primo è l'indice di posizione nella lista dove il nuovo elemento dovrebbe essere inserito, e il secondo è l'elemento che desideriamo inserire.

```
>>> x = [ 1, 2, 3]
>>> x.insert (2, "hello")
```

```
>>> print x
⇒ [1, 2, ”hello”, 3]
>>> x.insert (0, “start”)
>>> print x
⇒ [‘start’, 1, 2, ”hello”, 3]
```

Il metodo `remove (x)` rimuove un elemento della lista il cui valore è x. L'assenza di tale elemento produce un errore.

```
>>> x = [ 1, 2, 3, 4, 3, 5]
>>> x.remove(3)
>>> x
⇒ [1, 2, 4, 3, 5]
```

Il metodo `reverse ()` inverte gli elementi della lista

```
>>> x = [ 1, 3, 5, 6, 7]
>>> x.reverse ()
>>> x
⇒ [7, 6, 5, 3, 1]
```

Una lista può anche essere ordinata usando il metodo `sort ()`

```
>>> x = [ 3, 8, 4, 0, 2, 1]
>>> x.sort()
>>> x
⇒ [0, 1, 2, 3, 4, 8]
```

Infine per cancellare un elemento:

```
>>> x = [ 1, 2, 3, 4, 3, 5]
>>> del x[2]
>>> x
⇒ [1, 3, 4, 3, 5]
```

3 Tuple

Le tuple sono simili alle liste ma sono immutabili, non possono essere modificate una volta che sono state create. Le tuple sono utilizzate quando si deve essere certi che nessuno possa aggiungere o togliere elementi. Per il resto le tuple hanno il medesimo funzionamento delle liste. C'è solo una piccola differenza sintattica: sono racchiuse tra parentesi tonde. Gli operatori sono gli stessi delle liste. Vediamo qualche esempio:

```
>>> t = (12345, 54321, ‘ciao’)
>>> t[0]
```

```
⇒ 12345
```

Una lista può essere convertita in una tupla usando la funzione `tuple`:

```
>>> x = [ 1, 2, 3, 4]
>>> tuple(x)
⇒ (1, 2, 3, 4)
```

Viceversa, una tupla può essere convertita in una lista usando la funzione `list`:

```
>>> x = ( 1, 2, 3, 4)
>>> list(x)
⇒ [1, 2, 3, 4]
```

4 Stringhe

Oltre ai numeri, Python può anche manipolare stringhe, che possono essere espresse in diversi modi. Esse possono essere delimitate tra virgolette singole, doppie o triple e possono contenere l'indicazione per una nuova linea nel seguente modo \n.

Le stringhe sono immutabili. Gli operatori (in, + e *) e le funzioni (len, max e min) operano sulle stringhe come sulle liste e sulle tuple. La notazione degli indici e degli slice anche qui è la stessa ma non può essere usata per aggiungere rimuovere, o sostituire elementi.

5 Dizionari

Un dizionario rappresenta una collezione non ordinata di oggetti. Gli oggetti sono identificati univocamente mediante una chiave (generalmente una stringa) invece che mediante un indice numerico, come avviene nelle liste.

Ogni elemento del dizionario è rappresentato da una coppia (chiave:valore), la chiave serve per accedere all'elemento e recuperare il valore.

In analogia alle liste, anche per i dizionari Python lascia la possibilità di inserire oggetti eterogenei nello stesso dizionario. Vediamo ora alcuni esempi per illustrare le caratteristiche dei dizionari.

Esiste un elemento particolare che rappresenta il dizionario vuoto ():

```
>>> diz1 = {}
>>> len (diz)
⇒ 0.
```

È possibile conoscere il numero di elementi di un dizionario mediante la funzione len (x). É possibile controllare l'esistenza di una chiave nel dizionario mediante la funzione `has_key (x)`. Il risultato di tale funzione è booleano (1=vero, 0=falso).

```
>>> diz2 = 'stefano': 23, 'elena':19, 'enrico':25,
'simone':30}
>>> diz2.has_key('stefano')
⇒ 1
>>> diz2.has_key('luca')
⇒ 0
```

Esistono due metodi che permettono di estrarre dal dizionario la lista delle chiavi e la lista dei valori rispettivamente `keys ()` e `values ()`:

```
>>> diz2.keys ()
⇒ ['stefano', 'elena', enrico', 'simone']
>>> diz2.values ()
⇒ [23, 19, 25, 30]
```

Potremmo usare anche qui le funzioni `sort` per ordinare gli elementi del dizionario in ordine alfabetico e `del` per eliminare un qualche elemento dal dizionario.

Questa è sicuramente una delle migliori strutture del Python. I dizionari sono semplici da utilizzare, efficienti e molto flessibili, soprattutto sono molto utili per chi gestisce dati in strutture relazionali. Infatti combinando opportunamente liste e dizionari è possibile creare un vero e proprio database relazionale.

Strumenti di controllo del flusso

1 Espressioni e valori booleani

Python non ha un tipo di dato specifico booleano. Al posto di 0 userà il valore `None` e il valore vuoto come per esempio la stringa vuota assumerà valore falso. Qualsiasi altra cosa sarà considerata vera.

Espressioni di comparazione possono essere create usando gli operatori di comparazione (`<, <=, ==, >, >=, !=, <>, is, is not, in, not in`) e gli operatori logici (`and, not, or`) che ci daranno tutti come risultato 1 se il valore è vero e 0 se il risultato è falso.

2 L'istruzione if

In Python la forma più generale del costrutto `if-then-else` è:

```
if condizione1:
   istruzioni1
elif condizione2:
   istruzioni2
elif condizione3:
   istruzioni3
.
.
.
elif condizione (n-1):
   istruzioni (n-1)
else:
   istruzioni (n)
```

Questo dice: **`if`** `condizione1` è vera esegui `istruzioni1`; altrimenti, **`if`** `condizione2` è vera esegui `istruzioni2`; altrimenti..., e così via, sino a quando o si trova una condizione che è stata valutata vera oppure si giunge alla clausola `else` in quel caso viene eseguita l'ultima istruzione.

Ricordiamo che i gruppi di istruzione devono essere indentati nello stesso modo in quanto è solo l'indentazione che permette di strutturare l'istruzione if-else. Notiamo inoltre che a differenza del C o del Pascal non esistono parole che delimitano l'inizio e le fine del blocco.

3 Il ciclo while

Il ciclo `while` è così strutturato:

```
while condizione:
   istruzini
else:
   istruzioni
```

`Condizione` è un espressione che può assumere valore vero o falso. Sino a quando questo è vero le istruzioni vengo eseguite ripetutamente. Se il valore è falso il ciclo eseguirà la parte riferita all'`else` e il ciclo sarà così terminato. Notiamo che la parte `else` è opzionale infatti non è molto usata.

4 L'istruzione for

L'istruzione `for` di Python differisce un po' da quella a cui si è abituati in C o Pascal. Piuttosto che iterare sempre su una progressione aritmetica o dare all'utente la possibilità di definire sia il passo iterativo sia la condizione di arresto, in Python l'istruzione `for` compie un'iterazione sugli elementi di una sequenza di valori che può essere una lista, una tupla o una stringa. La forma generale è:

```
for variabile in sequenza:
   istruzioni
else:
   istruzioni
```

le istruzioni saranno eseguite una volta per ciascun elemento nella sequenza. La parte `else` è opzionale infatti non è molto usata.

5 La funzione range ()

Il comando `range` può essere usato insieme con il comando `len` su una lista, per generare una sequenza di indici che poi vengono usati dal ciclo `for`.

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

quindi dato un numero *n* , `range (n)` ritorna una sequenza 0, 1, 2..., n-1. È possibile far partire l'intervallo da un altro numero, o specificare un incremento diverso:

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

Le funzioni

Le funzioni sono un raggruppamento di istruzioni che prendono in ingresso un insieme di valori detti parametri e restituiscono un risultato come elaborazione dei parametri stessi.

La sintassi di base per una funzione o procedura in Python è la seguente:

```
def nome (parametro1, parametro2, ...):
   istruzioni
```

una volta definita una funzione in questo modo è possibile richiamarla semplicemente invocando il suo nome, seguito dalla lista di valori che si intende passare come parametri.

Come per le strutture di controllo Python usa indentazioni per delimitare il corpo di una funzione. L'esempio seguente è una semplice funzione scritta in linguaggio Python attraverso la quale possiamo calcolarici il fattoriale di un numero:

```
def fact(n):
     """Calcola il fattoriale di un dato numero."""
     r = 1
     while n > 0:
         r = r * n
         n = n - 1
     return r
```

la seconda linea è una stringa di documentazione opzionale. Il suo valore può essere ottenuto scrivendo `fact.__doc__`. Lo scopo della stringa di documentazione è di descrivere il comportamento interno di una funzione. I

parametri di una funzione possono essere valori per default che sono dichiarati assegnando un valore di default nella prima linea della funzione, nel seguente modo:

```
def fun (arg1, arg2=default, arg3=default,...):
```

un numero qualsiasi di parametri può essere assegnato come valore per default. I parametri con valore di default devono essere definiti come gli ultimi parametri della lista.

L'esempio che segue è una funzione in cui viene calcolata la potenza y di x. Se y non viene assegnato nel richiamare la funzione ad esso sarà assegnato il suo valore di default e quindi nel seguente caso la funzione calcolerà il quadrato di x.

```
def power(x, y=2):
    r = 1
    while y > 0:
         r = r * x
         y = y - 1
return r
>>> power (3, 3)
⇒ 27
>>> power (3)
⇒ 9
```

Un argomento può anche essere trasmesso in una funzione usando il nome del parametro piuttosto che la posizione. Quindi riprendendo l'esempio di prima avremo:

```
>>> power (2, 3)
⇒ 8
>>> power (3, 2)
⇒ 9
>>> power(y=2, x=3)
⇒ 9
```

Non è importante se un parametro formale ha un valore di default o meno. Nessun argomento deve ricevere un valore più di una volta, in una medesima invocazione non possono essere usati come parole chiave nomi di parametri formali corrispondenti ad argomenti posizionali.

Le funzioni Python possono anche essere chiamate usando argomenti a parola chiave nella forma "`parolachiave = valore`". Infatti ci sono due modi differenti nei quali ciò può essere fatto. Dove si desidera collegare un numero indefinito di argomenti alla fine di una lista in una struttura. L'altro metodo può collegare un numero arbitrario di argomenti trasmessi come parole chiave, che non hanno un parametro corrispondente elencato nella lista dei parametri della funzione, in un dizionario.

Dunque prefissando il nome del parametro finale della funzione con '*' ciò causa la creazione di un numero qualsiasi di argomenti senza parole chiave in una chiamata della funzione che sono assegnati come una tupla ad un dato parametro. Vediamo un semplice esempio di una funzione che calcoli il massimo elemento in una lista di numeri.

```
def maximum(*numbers):
    if len (numbers) == 0:
          return (None)
    else:
         max = numbers [0]
         for n in number[1: ]:
           if n > max: max = n
         return max
```

il comportamento della funzione sarà il seguente:

```
>>> maximum (3, 2, 8)
⇒ 8
>>> maximum (1, 5, 9, -2, 2)
⇒ 9
```

Se il parametro finale della lista è preceduto da '**', esso riceve un dizionario contenente tutti gli argomenti a parola chiave la cui chiave non corrisponde a un parametro formale. Ecco un esempio:

```
 def exampleFun(x, y, **other):
        print "x:", x, "y:", y, ", keys in 'other':",
other.keys()
     otherTotal = 0
     for k in other.keys():
          otherTotal = otherTotal + other[k]
       print "il numero totale dei valori in 'other' è",
otherTotal
```

```
>>> exampleFun (2, y="1", foo=3, bar=4)
⇒ x:2 y:1, Keys in 'other': [foo, bar]
il numero totale dei valori in 'other' è 7
```

Introducendo le funzioni si devono distinguere le "variabili locali alle funzioni" (quindi utilizzabili solo da esse) e le "variabili globali", ossia appartenenti al namespace del modulo (quindi utilizzabili al di fuori della funzione).

Quando si richiama una funzione vengono trasferiti i valori delle variabili ai parametri delle funzioni. Questo permette alla funzione di venire a conoscenza di informazioni provenienti dal blocco chiamante.

In generale nei linguaggi di programmazione esistono due modalità di passaggio:

i) passaggio per valore: viene trasferito alla funzione solo una copia della variabile, quindi la funzione non può alterare il valore di quella variabile;
ii) passaggio per riferimento: in questo caso viene trasferita la variabile vera e propria e quindi la funzione può alterare il valore della variabile.

In Python, a differenza di altri linguaggi, i parametri passati alle funzioni sono sempre trasferiti per valore.

Infatti quando si utilizza una variabile, Python cerca prima il nome di quella variabile nel namespace locale. Se la ricerca non da esito positivo, si prosegue con il namespace globale e solo successivamente si va a cercare il nome tra le funzioni builtin. Questo meccanismo permette di utilizzare il valore delle variabili globali, ma di non poterle mai modificare, in quanto un nuovo assegnamento alla variabile provoca la creazione dello stesso nome in namespace nuovo.

I moduli

Quando si inizia a diventare programmatori esperti, i programmi crescono a vista d'occhio e diventa estremamente difficile gestire l'intero programma in un unico file di script. Per questo è opportuno suddividere il programma

in diversi file di script. Questa metodologia permette di creare delle raccolte di funzioni che possono essere utilizzate in progetti diversi.

I moduli quindi sono dei files di script che possono essere richiamati da altri programmi Python per riutilizzare le funzioni contenute in essi.

I moduli possono essere creati facilmente dal programmatore, ma ne esistono moltissimi già precostituiti e contenuti dentro all'istallazione di Python.

Per scrivere un modulo, basta semplicemente creare un file con estensione PY e riempirlo con tutte le funzioni necessarie al modulo stesso. Per utilizzare il modulo appena creato è sufficiente importarlo con il comando "import". Ad esempio, supponendo di aver creato il modulo "libreria.py", basta scrivere in cima al programma:

```
>>> import libreria
```

Dopo aver importato il modulo possiamo semplicemente richiamare le funzioni contenute in esso utilizzando la **dot notation.** Digitando il nome del modulo, un punto e il nome della funzione riusciamo ad entrare nel modulo e richiamare la funzione desiderata. Ad esempio, supponendo che "libreria.py" contenga le funzioni apri (), chiudi () e sposta (oggetto), posso richiamare le funzioni nel seguente modo:

```
>>> import libreria
>>> libreria.apri ()
>>> libreria.sposta (oggetto)
>>> libreria.chiudi ()
```

Bisogna considerare un altro vantaggio offerto dai moduli: quando si interpreta un programma Python, esso crea per ogni modulo una versione "semicompilata" in un linguaggio intermedio più veloce (bytecode). Dopo un'esecuzione, è possibile notare la presenza di tanti files con estensione PYC (Python compiled). Ogni file ha lo stesso nome del sorgente PY, ma con estensione PYC. In questo modo Python non deve interpretare il codice tutte le volte, ma solamente quando viene effettuata una modifica. L'interprete confronta ogni volta la data e l'ora del file PY con il file PYC, se sono diverse interpreta il codice, altrimenti esegue direttamente il bytecode.

Bibliografia

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

[2] *Python Tutorial* disponibile sul sito www.python.org