Programmazione in Python

Indice generale

Fondamenti di programmazione

Questa breve guida si prefigge di chiarire alcuni semplici concetti relativi alla programmazione, in particolare nel linguaggio Python. Potete trovare una guida completa sulla programmazione in Python al seguente indirizzo:

https://www.programmareinpython.it/video-corso-python-base/introduzione-e-installazione-di-python/

Nelle lingue il significato dei termini può variare in base al contesto. Per esempio in italiano la parola casa può rappresentare un edificio, ma anche il paese di appartenenza, le persone a cui ci si sente legati. La parola terra assume un significato nella frase “la mia terra natale è l'Italia” ed un significato molto diverso in “mi sento a terra” o ancora “mi sono scrollato la terra di dosso”. La stessa parola “lingua” assume significasti completamente differenti: c'è la lingua parlata, la lingua intesa come organo o anche la lingua di terra per indicare un lembo di terra che si insinua in un lago, in un mare o nell'oceano. E' difficile spiegarsi in modo rigoroso perché il significato di molti vocaboli è ambiguo. Se riusciamo a capirci è perché disponiamo di esperienze comuni, perché facciamo riferimento al mondo reale e adottiamo di una certa dose di buon senso. Anche così tuttavia i fraintendimenti non sono rari.

Programmare significa implementare, (scrivere, codificare) un algoritmo (una procedura per ottenere un certo risultato in certo numero di passi, attingendo ad un insieme finito di istruzioni “semplici”) mediante un linguaggio di programmazione. Programmare un computer significa, a grandi linee, istruirlo ad eseguire determinati compiti mediante una sequenza di istruzioni che prendono il nome di programma. A differenza delle lingue, i linguaggi di programmazione sono linguaggi formali, dotati di regole ben precise e privi di ambiguità.

In principio i computer venivano programmati scrivendo nella memoria centrale codici operativi ed operandi di istruzioni in **linguaggio macchina**, (istruzioni aritmetiche, logiche, accessi alla memoria, salti etc.). Il programmatore inseriva in memoria lunghe sequenze di 0 e di 1 eseguibili direttamente dal processore.

Non doveva essere facile ricordarsi il significato di tutti i codici operativi delle istruzioni ed in effetti la maggioranza dei programmatori in linguaggio macchina ne usava solo alcune. I codici operativi venivano scritti su un foglio di carta affiancati da **codici mnemonici** (add, sub, mov, jmp, …) che ne spiegavano brevemente il significato. Al crescere della potenza degli elaboratori si iniziò a scrivere i programmi usando direttamente i codici mnemonici (magari con operandi in esadecimale o in decimale) ed a lasciare il compito della traduzione ad un programma detto **assembler** (**assemblatore** in italiano): nasceva **l'assembly**, linguaggio a basso livello (vicino al linguaggio macchina) utilizzato ancor oggi in molti settori ed insegnato tuttora in alcune scuole. Per

la prima volta si cominciò a distinguere tra **codice sorgente,** scritto dal programmatore, e **codice eseguibile**, tradotto dall'assemblatore a partire dal codice sorgente e comprensibile dal processore.

Anche se spesso si parla genericamente di linguaggio assembly, in realtà gli assembly sono almeno tanti quanto i modelli dei processori (per uno stesso processore possono anche diverse varianti di assembly) perché ogni codice mnemonico fa riferimento ad una istruzione ben precisa di un certo linguaggio macchina. A complicare le cose, spesso il programmatore assembly non programma più "sulla nuda macchina" ma basandosi su un determinato sistema operativo. Un programma scritto in assembly per ARM (famiglia di processori molto usati in smartphone, tablet, console portatili, ...) non può essere utilizzato su un computer dotato di processore x86-64. Come conseguenza di ciò, i programmi scritti in assembly sono scarsamente **portabili**: tradurli significa riscriverli in buona parte. Inoltre, scrivere programmi in assembly è difficile e richiede tempo. Il vantaggio di questo linguaggio è che permette ai programmatori bravi di ottenere codice molto efficiente che sfrutta appieno la potenza della macchina su cui gira.

Il passo successivo all'assembly fu quello di utilizzare linguaggi più simili a quelli umani e sufficientemente slegati dalla macchina da poter essere portati su **architetture** (processore, computer, sistema operativo) molto diverse. Tali linguaggi vengono detti **ad alto livello**, in contrapposizione a quelli **a basso livello**, come gli assembly o i linguaggi macchina. Tra i più famosi ricordiamo C, Basic, Pascal, Java, Ruby, Python, PHP, Javascript, ... ma ne esistono centinaia, se non addirittura migliaia!

Come accade per l'assembly, anche nei linguaggi a basso livello la traduzione del codice sorgente in linguaggio macchina viene affidata ad un programma. In alcuni linguaggi (C, Pascal, …) tale programma traduttore genera un eseguibile in linguaggio macchina che può essere utilizzato tutte le volte che si desidera, anche in mancanza dei sorgenti, similmente a quanto avviene con l'assemblatore. Tale programma traduttore prende il nome di **compilatore** e i programmi che ne fanno uso **compilati**. Per altri linguaggi (il vecchio Basic, Ruby, Python, PHP, Javascript …) la traduzione viene effettuata "al volo" ogni volta che si esegue il programma. In questo caso il programma che effettua la traduzione prende il nome di **interprete** e i programmi che ne fanno uso **interpretati**.

L'esempio che segue serve a chiarire le idee. Immaginate di dover parlare spesso ad un pubblico di madrelingua tedesca (linguaggio macchina). Potete scrivere il vostro discorso direttamente in tedesco e leggerlo ogniqualvolta desideriate (scrittura del programma direttamente in linguaggio macchina), potete scriverlo in italiano (linguaggio ad alto livello), farvelo tradurre da un amico che sa il tedesco e poi leggerlo ogni volta che serve (approccio compilato) oppure potete scriverlo in italiano (linguaggio ad alto livello) e il vostro amico ve lo traduce "al volo" di fronte alla teutonica platea (approccio interpretato).

A prima vista l'approccio compilato sembra il migliore perché:

- Non necessita della conoscenza del linguaggio macchina;
- Avvenuta la compilazione i sorgenti non sono più necessari (posso anche evitare di diffonderli se non voglio);
- Il codice eseguibile è direttamente comprensibile dal processore senza traduzioni, quindi è estremamente veloce.

Anche l'approccio interpretato, tuttavia, ha alcuni vantaggi:

- Anche qui, non è necessaria la conoscenza del linguaggio macchina;
- La portabilità dei linguaggi interpretati è maggiore, infatti a partire dal sorgente si può eseguire il programma su qualunque architettura, a patto che sia presente l'interprete, senza la necessità di ricompilarlo;
- Essendo eseguiti "al volo", spesso i linguaggi interpretati possono adottare un approccio più

amichevole nei confronti del programmatore rispetto a quelli compilati. Generalmente infatti questi ultimi risultano sempre un po' legati all'hardware.

Esistono anche approcci “ibridi”, come quello adottato da Java (che non è JavaScript!!!). Il sorgente di un programma Java viene prima compilato per creare un file intermedio detto “bytecode” che poi viene interpretato dall’interprete, la Java Virtual Machine.

La maggiore portabilità dei linguaggi interpretati in alcune circostanze è fondamentale e per essa si sacrifica volentieri la velocità. Immaginatevi di scrivere un programma JavaScript in una pagina web su Internet che viene visualizzata dentro un browser. Non potendo prevedere se gli utenti si collegheranno mediante un cellulare ARM con Android (Linux), un iPad (ARM) con iOS, un netbook x86-64 con Windows 7, un iMac G5 con Mac OS X, un portatile x86-64 con Linux Mint, ... è più semplice fornire i sorgenti e lasciare la traduzione all'interprete presente sul dispositivo dell'utente.

Ci sono anche altri ambiti in cui i linguaggi interpretati sono preferibili. Immaginiamo per esempio di possedere un server web (inteso come il computer che ospita siti ed applicazioni web) che affittiamo a numerosi clienti. Ad esso accedono numerosi webmaster che, ovviamente, lavorano da remoto. Se usassero dei linguaggi compilati dovrebbero collegarsi via SSH per compilare i sorgenti. L’accesso SSH potrebbe rappresentare un problema per la sicurezza del nostro server e degli altri clienti. Invece, per modificare un programma interpretato è sufficiente che modificare e salvare i sorgenti, per esempio via SFTP.

L'esecuzione di un programma interpretato è anche più sicura rispetto a quella di un linguaggio compilato perché un linguaggio interpretato è limitato strettamente da ciò che gli permette di fare il suo interprete.

Linguaggio di programmazione Python

Python è un linguaggio di programmazione ad alto livello interpretato. E' stato scelto per la sua semplicità e diffusione. Con gli appunti che seguono non si intende fornire un manuale di Python ma solo alcuni concetti base trasferibili anche ad altri linguaggi. Python va visto come mezzo per imparare a programmare, non come fine.

Per eseguire un programma scritto in Python è sufficiente digitare

python nomeprogramma.py

dove nomeprogramma.py è un file di testo contenente codice Python oppure utilizzare il comando python, senza il nome del file, una shell interattiva che esegue il codice Python alla pressione del tasto invio.

Espressioni booleane

Un’**espressione booleana** è un’espressione che è o vera o falsa. In Python un’espressione vera ha valore **True**, un’espressione falsa ha valore **False**.

L’operatore == confronta due valori e produce un risultato di tipo True o False:

```
>>> 5==5
True
>>> 5==6
False
```

Nella prima riga i due operandi sono uguali, così l’espressione vale True (vero); nella seconda riga 5 e 6 non sono uguali, così otteniamo False (falso).

L’operatore == è uno degli operatori di confronto; gli altri sono **!=** (diverso), < (minore), > (maggiore), <= (minore o uguale), >= (maggiore o uguale). Un errore comune è quello di scambiare il simbolo di uguale (=) con il doppio uguale (==). Ricorda che = è un operatore di assegnazione e == un operatore di confronto.

Operatori logici

Gli operatori logici più usati sono **and**, **or** e **not**. Il significato di questi operatori è simile al loro significato in italiano: per esempio, (x>1) and (x<8) è vera solo se x è più grande di 1 e meno di 8.

(n%2==0) or (n%3==0) è vera se si verifica almeno una delle due condizioni e cioè se il numero è divisibile per 2 o per 3 (% indica il resto della divisione intera).

Infine, l’operatore not nega il valore di un’espressione booleana, trasformando in falsa un’espressione vera e viceversa. Così se x>y è vera (x è maggiore di y), not(x>y) è falsa.

Tipi di dato

In Python esistono alcuni “tipi di dato” che, a differenza di quanto accade in altri linguaggi, come il C, non dipendono dall'architettura. I più comuni sono i numeri **interi** {0,1,-1,2,-2, …}, i numeri **razionali** {0.0, 1.0, -1.0, 0.5, -0.5, …}, le **stringhe**, {'0', 'a', '2BSA', '2CSA', ‘2FSA', ‘2GSA’, 'H2O', 'ciao mondo', …}, i **booleani**, detti anche valori di verità {False, True}. Chiaramente, tranne i booleani, che sono due, gli altri sono infiniti (numerabili), quindi dobbiamo accontentarci di utilizzare solo un loro sottoinsieme (finito).

Ogni tipo ha le sue operazioni e in caso di omonimia, l'interprete di Python capisce di quale operazione si tratta in base al tipo degli operandi. Per esempio:

Operazione	**Risultato**	**Tipi operandi**	**Tipo Risultato**	**Operazione**
3+4	7	intero, intero	intero	Somma tra interi
'1'+'Ls'	'1Ls'	stringa,stringa	stringa	Concatenazione tra stringhe
3<4	True	intero, intero	booleano	Confronto tra interi
'pino'<'arturo'	False	stringa,stringa	booleano	Confronto tra stringhe
'2'>'11'	True	stringa, stringa	booleano	Confronto tra stringhe
True and False	False	booleano, booleano	booleano	Operatore logico
3==3	True	intero, inter	booleano	Confronto tra interi
3*'a'	'aaa'	intero, stringa	stringa	Ripetizione di stringa
3+4.2	7.2	intero, razionale	razionale	Somma tra intero e razionale

L'interprete capisce che nel primo caso deve effettuare una somma tra interi mentre nel secondo una **concatenazione** tra stringhe anche se l'operazione è espressa dallo stesso simbolo. Nel terzo esempio c'è un confronto tra interi (minore) e nel quarto tra stringhe (precede in ordine alfabetico): viene usato lo stesso simbolo ma il significato è diverso. Notate come il tipo degli operandi non necessariamente è uguale e così pure il tipo del risultato.

Talvolta è possibile cambiare il tipo di un dato. Per esempio: int('3') restituisce 3, str(3) restituisce '3'. Nel primo caso una stringa viene trasformata in un intero, nel secondo un intero viene

trasformato in una stringa.

Variabili

Le **variabili** sono dei “contenitori” realizzati mediante porzioni di memoria, delle “scatole con un nome scritto sopra” che le distingue tra loro. Il contenuto delle variabili può variare (da qui il nome) durante l'esecuzione del programma. Normalmente una variabile va **dichiarata** (esiste una variabile con un certo nome). Dopo la dichiarazione è possibile assegnarvi un valore mediante l'operatore `=`. A differenza di altri linguaggi, come il C per esempio, in Python la dichiarazione della variabile avviene la prima volta che le viene assegnato un valore.

```
pippo=10
```

significa che esiste una variabile che si chiama pippo e contiene il valore 10 (più o meno). Se scriviamo ancora:

```
pippo=8
```

la variabile è sempre la stessa di prima.

Input ed output

Esistono numerose istruzioni di input e di output. In Python le più usate sono **print**, per scrivere una stringa (e nel caso l'oggetto non sia una stringa viene automaticamente convertito), **input** per leggere una stringa. Se vogliamo leggere un intero, dunque, dobbiamo scrivere int(input()).

Legge una stringa di testo e la assegna alla variabile x:

```
x=input()
```

Legge una stringa, la trasforma in intero e la assegna alla variabile x:

```
x=int(input())
```

Legge una stringa, la trasforma in un decimale e lo assegna alla variabile x:

```
x=float(input())
```

Stampa ciao mondo:

```
print('ciao mondo')
```

Stampa il contenuto della variabile x:

```
print(x)
```

Istruzioni di controllo

La **selezione** (**if**) è traducibile in italiano col termine se, l'**iterazione** (**while**) con mentre. Entrambi si aspettano una condizione che può essere vera o falsa. Per implementare le strutture di controllo è necessario **indentare** (allineare) il codice appartenente ad esse.

Esempi:

```
if True:
  print('ciao')
```

scrive ciao

```
if False:
  print('ciao')
```

non scrive nulla

```
while True:
  print('ciao')
```

scrive ciao all'infinito (o finché qualcuno non termina il programma con CTRL-C)

```
while False:
  print('ciao')
```

non scrive niente.

Chiaramente, tranne il “while true”, che in certi casi potrebbe anche tornare utile, gli altri non servono assolutamente a nulla. Più interessanti sono i casi in cui la condizione non è una costante ma il risultato di una espressione booleana contenente una variabile. In quel caso il valore della condizione varia al variare del valore contenuto nella variabile.

```
if input()==input():
  print('i due input sono uguali')
```

Notare come il valore di verità del confronto dipende dai valori inseriti.

```
pensato=input()
while input()!=pensato:
  print('spiacente, hai sbagliato, riprova')
print('bravo, hai indovinato')
```

Spesso il ciclo while viene usato per ripetere delle istruzioni un certo numero di volte. Ecco un classico esempio:

```
i=0
while i<10:
  print('ciao')
  i=i+1
```