Capitolo 2

Variabili, espressioni ed istruzioni

Una delle caratteristiche più potenti di un linguaggio di programmazione è la capacità di elaborare delle **variabili**. Una variabile è un nome che fa riferimento ad un valore.

2.1 Istruzioni di assegnazione

Un'**istruzione di assegnazione** serve a creare una nuova variabile, specificandone il nome, e ad assegnarle un valore:

```
>>> messaggio = 'E ora qualcosa di completamente diverso'
>>> n = 17
>>> pi = 3.141592653589793
```

Questo esempio effettua tre assegnazioni. La prima assegna una stringa ad una nuova variabile di nome `messaggio`; la seconda assegna il numero intero `17` alla variabile `n`; la terza assegna il valore decimale approssimato di π alla variabile `pi`.

Un modo frequente di raffigurare le variabili è quello di scriverne il nome e disegnare una freccia che punta al loro valore. Questa illustrazione è chiamata **diagramma di stato** perché mostra lo stato in cui si trova la variabile. La Figura 2.1 contiene i risultati delle istruzioni di assegnazione dell'esempio precedente.

2.2 Nomi delle variabili

Generalmente, i programmatori chiamano le variabili con dei nomi significativi, in modo da documentare a cosa servono.

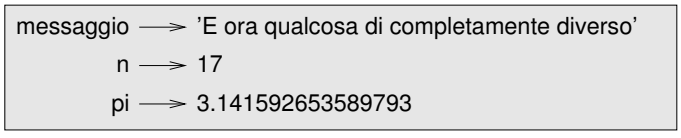

Figura 2.1: Diagramma di stato.

I nomi possono essere lunghi a piacere e possono contenere sia lettere che numeri, ma non possono iniziare con un numero. È possibile usare anche le lettere maiuscole, ma per i nomi di variabile è convenzione utilizzare solo lettere minuscole. In ogni caso, tenete conto che, per l'interprete, maiuscole e minuscole sono diverse, pertanto `spam`, `Spam` e `SPAM` sono variabili diverse.

Il trattino basso o *underscore*, `_`, può far parte di un nome: è usato spesso in nomi di variabile composti da più parole (per esempio `il_tuo_nome` o `monty_python`).

Se assegnate un nome non valido alla variabile, otterrete un errore di sintassi:

```
>>> 76tromboni = 'grande banda'
SyntaxError: invalid syntax
>>> altro@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Zymurgy Teorico Avanzato'
SyntaxError: invalid syntax
```

`76tromboni` non è valido perché non inizia con una lettera. `altro@` non è valido perché contiene un carattere non ammesso (la chiocciola `@`). Ma cosa c'è di sbagliato in `class`?

Succede che `class` è una delle **parole chiave riservate** di Python. L'interprete utilizza queste parole per riconoscere la struttura del programma, pertanto non è consentito usarle come nomi di variabile.

Python 3 ha queste parole chiave:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Non occorre imparare a memoria questo elenco. Nella maggior parte degli ambienti di sviluppo, le parole chiave vengono evidenziate con un diverso colore; se cercate di usarne una come nome di variabile, ve ne accorgerete subito.

2.3 Espressioni e istruzioni

Un'**espressione** è una combinazione di valori, variabili e operatori. Un valore è considerato già di per sé un'espressione, come pure una variabile, per cui quelle che seguono sono tutte delle espressioni valide (supponendo che alla variabile `n` sia già stato assegnato un valore):

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

Quando scrivete un'espressione al prompt dei comandi, l'interprete la **valuta**, cioè trova il valore dell'espressione. Nell'esempio di prima, `n` ha valore 17 e `n + 25` ha valore 42.

Un'**istruzione** è una porzione di codice che l'interprete Python può eseguire e che ha un qualche effetto, come creare una variabile o mostrare un valore.

```
>>> n = 17
>>> print(n)
```

La prima riga è un'istruzione di assegnazione che dà un valore alla variabile `n`. La seconda è un'istruzione di stampa che mostra a video il valore di `n`.

Quando scrivete un'istruzione, l'interprete la **esegue**, cioè fa quello che l'istruzione dice di fare. In linea generale, le istruzioni, a differenza delle espressioni, non contengono valori.

2.4 Modalità script

Finora abbiamo avviato Python in **modalità interattiva**, detta anche "a riga di comando", che vuol dire interagire direttamente con l'interprete. La modalità interattiva è un buon modo per iniziare e fare esperimenti, ma se si deve lavorare con più di qualche riga di codice, può diventare in breve tempo un impiccio.

In alternativa alla riga di comando, si può scrivere e salvare un programma in un file di testo semplice, chiamato **script**, ed usare poi l'interprete in **modalità script** per eseguirlo. Per convenzione, i file contenenti programmi Python hanno nomi che terminano con l'estensione `.py` .

Se già sapete come creare e avviare uno script nel vostro computer, siete a cavallo. Altrimenti vi consiglio di nuovo di usare PythonAnywhere. Le istruzioni per l'avvio in modalità script sono pubblicate all'indirizzo `http://tinyurl.com/thinkpython2e`.

Poiché Python consente entrambe queste modalità, potete provare dei pezzi di codice in modalità interattiva prima di inserirli in uno script. Ma tra le due modalità, ci sono delle differenze che possono disorientare.

Per esempio, usando Python come una calcolatrice, si può scrivere:

```
>>> miglia = 26.2
>>> miglia * 1.61
42.182
```

La prima riga assegna un valore a `miglia`, e non ha alcun effetto visibile. La seconda riga è un'espressione, e l'interprete la valuta e ne mostra il risultato. Vediamo così che una maratona misura circa 42 chilometri.

Ma se scrivete lo stesso codice in uno script e lo avviate, non otterrete alcun riscontro. In modalità script, un'espressione, di per sé, non ha effetti visibili. In realtà Python valuta l'espressione, ma non ne mostra il risultato finché non gli dite esplicitamente di farlo:

```
miglia = 26.2
print(miglia * 1.61)
```

Questo comportamento inizialmente può confondere.

Uno script di solito contiene una serie di istruzioni. Se ci sono più istruzioni, i risultati compaiono uno alla volta, man mano che le istruzioni vengono eseguite.

Per esempio lo script:

```
print(1)
x = 2
print(x)
```

visualizza questo:

```
1
2
```

mentre l'istruzione di assegnazione non produce alcun output sullo schermo.

Per controllare se avete capito tutto, scrivete le seguenti istruzioni nell'interprete Python per vedere quali effetti producono:

```
5
x = 5
x + 1
```

Ora scrivete le stesse istruzioni in uno script ed avviatelo. Qual è il risultato? Modificate lo script trasformando ciascuna espressione in un'istruzione di stampa, ed avviatelo nuovamente.

2.5 Ordine delle operazioni

Quando un'espressione contiene più operatori, la successione con cui viene eseguito il calcolo dipende dall'**ordine delle operazioni**. Per quelle matematiche, Python segue le stesse regole di precedenza comunemente usate in matematica. L'acronimo **PEMDAS** è un modo utile per ricordare le regole:

- **P**arentesi: hanno il livello di precedenza più elevato e possono essere usate per forzare la valutazione di un'espressione secondo qualsiasi ordine si desideri. Dato che le espressioni tra parentesi sono valutate per prime, `2 * (3-1)` fa 4, e `(1+1)**(5-2)` fa 8. Si possono usare le parentesi anche solo per rendere più leggibile un'espressione, come in `(minuti * 100) / 60`, e in questo caso non influiscono sul risultato.
- **E**levamento a potenza: ha la priorità successiva, così `1 + 2**3` fa 9, e non 27, e `2 * 3**2` fa 18, e non 36.
- **M**oltiplicazione e **D**ivisione hanno priorità superiore ad **A**ddizione e **S**ottrazione. Per cui `2*3-1` fa 5, e non 4, e `6+4/2` fa 8, e non 5.
- Gli operatori con la stessa priorità vengono valutati da sinistra verso destra (eccetto la potenza), per cui nell'espressione `gradi / 2 * pi`, la divisione viene calcolata per prima e il risultato viene moltiplicato per `pi`. Per dividere `gradi` per 2π, dovete usare le parentesi o scrivere `gradi / 2 / pi`.

Personalmente, non mi sforzo molto di ricordare la precedenza degli operatori. Se non ne sono certo guardando un'espressione, inserisco le parentesi per fugare ogni dubbio.

2.6 Operazioni sulle stringhe

In linea generale non è possibile effettuare operazioni matematiche sulle stringhe, anche se la stringa sembra un numero; quindi gli esempi che seguono non sono validi.

```
'2'-'1'    'uova'/'facili'    'terzo'*'una magia'
```

Ma ci sono due eccezioni: + e *.

L'operatore + esegue il **concatenamento**, cioè unisce le stringhe collegandole ai due estremi. Per esempio:

```
>>> primo = 'bagno'
>>> secondo = 'schiuma'
>>> primo + secondo
bagnoschiuma
```

Anche l'operatore * funziona sulle stringhe: ne esegue la ripetizione. Per esempio, `'Spam'*3` dà `'SpamSpamSpam'`. Uno degli operandi deve essere una stringa, l'altro un numero intero.

Questo utilizzo di + e * è coerente per analogia con l'addizione e la moltiplicazione. Dato che `4*3` è equivalente a `4+4+4`, ci possiamo aspettare che `'Spam'*3` sia la stessa cosa di `'Spam'+'Spam'+'Spam'`, ed infatti è così. Tuttavia, concatenamento e ripetizione di stringhe differiscono da addizione e moltiplicazione di interi per un particolare importante. Riuscite a pensare ad una proprietà che ha l'addizione ma che non vale per il concatenamento di stringhe?

2.7 Commenti

Al crescere delle sue dimensioni e della sua complessità, un programma diventa anche sempre più difficile da leggere. I linguaggi formali sono densi di significato, e spesso non è facile guardare un segmento di codice scritto da altri e capire immediatamente che cosa fa, o perché.

Per questo motivo, è buona abitudine aggiungere ai vostri programmi delle annotazioni che spiegano in linguaggio naturale ciò che il programma sta facendo. Queste annotazioni si chiamano **commenti**, contrassegnati dal simbolo #:

```
# calcola la percentuale di ora trascorsa
percentuale = (minuti * 100) / 60
```

In questo caso, il commento si trova su una riga a sé stante, ma potete anche inserire un commento in coda a una riga:

```
percentuale = (minuti * 100) / 60     # percentuale di un'ora
```

Tutto ciò che viene scritto dopo il simbolo # e fino alla fine della riga, viene trascurato e non influisce in alcun modo sull'esecuzione del programma.

I commenti più utili sono quelli che documentano caratteristiche del codice di non immediata comprensione. È ragionevole supporre che chi legge il codice possa capire *cosa* esso faccia; è più utile spiegare *perché*.

Questo commento è ridondante e inutile:

```
v = 5     # assegna 5 a v
```

Questo commento contiene invece un'informazione utile che non è contenuta nel codice:

```
v = 5     # velocità in metri/secondo
```

Dei buoni nomi di variabile possono ridurre la necessità di commenti, ma nomi lunghi possono complicare la lettura, pertanto va trovato un giusto compromesso.

2.8 Debug

In un programma si possono verificare tre tipi di errori: gli errori di sintassi, gli errori in esecuzione e gli errori di semantica. È utile analizzarli singolarmente per facilitarne l'individuazione.

2.8.1 Errori di sintassi

Il termine **sintassi** si riferisce alla struttura di un programma e alle regole che la governano. Ad esempio, le parentesi devono essere sempre presenti a coppie corrispondenti, così `(1 + 2)` è corretto, ma `8)` è un **errore di sintassi**.

Se c'è anche un solo errore di sintassi in qualche parte del programma, non sarete in grado di eseguirlo: Python visualizzerà subito un messaggio d'errore e lo terminerà. Nelle prime settimane della vostra carriera di programmatori, dovrete probabilmente dedicare molto tempo a correggere errori di sintassi. Ma con l'esperienza, ne commetterete meno e li troverete più velocemente.

2.8.2 Errori in esecuzione

Il secondo tipo di errore è l'**errore in esecuzione** (o di *runtime*), così chiamato perché non compare fino a quando il programma non viene eseguito. Questi errori sono anche detti **eccezioni** perché indicano che è accaduto qualcosa di eccezionale (e di spiacevole) durante l'esecuzione.

Gli errori di runtime sono molto rari nei programmi semplici, come quelli che vedrete nei primi capitoli di questo libro, e potrebbe passare un po' di tempo prima di incontrarne uno.

2.8.3 Errori di semantica

Il terzo tipo di errore è l'**errore di semantica** (o di logica), che è correlato al significato del programma. In presenza di un errore di semantica, il programma verrà eseguito senza che compaia alcun messaggio di errore, ma non farà la cosa giusta: farà qualcosa di diverso. Nello specifico, farà esattamente ciò che voi gli avete detto di fare, esprimendovi in modo sbagliato.

Gli errori di semantica sono insidiosi e identificarli può essere complicato, perché occorre lavorare a ritroso, partendo dai risultati dell'esecuzione e cercando di risalire a che cosa non sia andato per il verso giusto.

2.9 Glossario

variabile: Un nome che fa riferimento ad un valore.

assegnazione: Istruzione che assegna un valore ad una variabile.

diagramma di stato: Rappresentazione grafica di una serie di variabili e dei valori ai quali esse si riferiscono.

parola chiave riservata: Parola chiave destinata esclusivamente all'analisi del programma e che non può essere usata come nome di variabile o di funzione, come `if`, `def`, e `while`.

operando: Uno dei valori sui quali si applica un operatore.

espressione: Combinazione di variabili, operatori e valori che rappresentano un unico valore risultante.

valutare: Semplificare un'espressione eseguendo una serie di operazioni che producono un unico valore.

istruzione: Porzione di codice che rappresenta un comando o un'azione, come le istruzioni di assegnazione e di stampa che abbiamo visto finora.

eseguire: Dare efficacia a un'istruzione e fare ciò che dice.

modalità interattiva: Un modo di usare l'interprete Python, scrivendo del codice al prompt.

modalità script: Un modo di usare l'interprete Python, leggendo del codice da uno script ed eseguendolo.

script: Un programma scritto e memorizzato in un file di testo.

ordine delle operazioni: Regole che stabiliscono l'ordine in cui vengono valutate le espressioni che contengono più operandi ed operatori.

concatenare: Unire due stringhe accodando la seconda alla prima.

commento: Annotazione in un programma, rivolta ad altri programmatori (o a chi legge il codice sorgente), che non ha effetti sull'esecuzione del programma.

errore di sintassi: Errore in un programma che ne rende impossibile l'analisi (il programma non è interpretabile).

eccezione: Errore (detto anche di *runtime*) che si verifica mentre il programma è in esecuzione.

semantica: Il significato logico di un programma.

errore di semantica: Errore nel programma tale da produrre risultati diversi da quelli che il programmatore si aspettava.

2.10 Esercizi

Esercizio 2.1. *Rinnovo la raccomandazione del capitolo precedente: ogni volta che apprendete qualcosa di nuovo, provatelo in modalità interattiva e fate degli errori di proposito per vedere cosa non funziona.*

- *Abbiamo visto che* `n = 42` *è valido. E* `42 = n`*?*
- *E se scrivete* `x = y = 1`*?*

- *In alcuni linguaggi, ogni istruzione termina con un punto e virgola,* `;`*. Cosa succede se mettete un punto e virgola alla fine di un'istruzione in Python?*
- *E se mettete un punto alla fine di un'istruzione?*
- *Nella notazione matematica potete indicare la moltiplicazione di x per y scrivendo: xy. Cosa succede scrivendo questo in Python?*

Esercizio 2.2. *Fate un po' di pratica con l'interprete Python usandolo come calcolatrice:*

1. *Il volume di una sfera di raggio r è $\frac{4}{3}\pi r^3$. Che volume ha una sfera di raggio 5?*
2. *Il prezzo di copertina di un libro è € 24,95, ma una libreria ottiene il 40% di sconto. I costi di spedizione sono € 3 per la prima copia e 75 centesimi per ogni copia aggiuntiva. Qual è il costo totale di 60 copie?*
3. *Se uscite di casa alle 6:52 di mattina e correte 1 miglio a ritmo blando (8:15 al miglio), poi 3 miglia a ritmo moderato (7:12 al miglio), quindi 1 altro miglio a ritmo blando, a che ora tornate a casa per colazione?*