Istruzioni condizionali

L'argomento principale di queste note è l'istruzione `if`, che permette di eseguire codice diverso a seconda dello stato del programma.

Espressioni booleane

Un'**espressione booleana** è un'espressione che può essere o vera o falsa. Gli esempi che seguono usano l'operatore ==, confrontano due valori e restituiscono `True` (vero) se sono uguali, `False` (falso) altrimenti:

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

`True` e `False` sono valori speciali che sono di tipo `bool`; non sono delle stringhe:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

L'operatore == è uno degli **operatori di confronto** (chiamati anche operatori relazionali); gli altri sono:

```
x != y               # x è diverso da y
x > y                # x è maggiore di y
x < y                # x è minore di y
x >= y               # x è maggiore o uguale a y
x <= y               # x è minore o uguale a y
```

Queste operazioni vi saranno familiari, tuttavia i simboli usati in Python non sono del tutto uguali a quelli matematici. Un errore frequente è usare il simbolo di uguale(=) anziché il doppio uguale (==). Ricordatevi la differenza: = è un operatore di assegnazione, mentre == è un operatore di confronto. Inoltre in Python non esistono simboli come =< o =>.

Operatori logici

Ci sono tre **operatori logici**: `and`, `or`, e `not`. Il significato di questi operatori è simile a quello comune (e, o, non): per esempio, l'espressione `x > 0 and x < 10` è vera solo se sono vere *entrambe* le condizioni, cioè `x` è più grande di 0 *e* più piccolo di 10.

L'espressione `n > 2 or n > 3` invece è vera se è verificata *almeno una* delle due condizioni, cioè se il numero è maggiore di 2 *o* di 3 (o di entrambi).

Infine, l'operatore `not` nega il valore di un'espressione booleana, per cui `not (x > y)` è vera se `x > y` è falsa, cioè se `x` è minore o uguale a `y`.

In senso stretto, gli operandi degli operatori logici dovrebbero essere delle espressioni booleane, ma qui Python non è rigido: ogni numero diverso da zero viene accettato ed interpretato come `True`.

Note provenienti dal Capitolo 5 di "Pensare in Python" di A. Downey, traduzione di A. Zanella. https://github.com/AllenDowney/ThinkPythonItalian

```
>>> 42 and True
True
```

Questa flessibilità può essere utile, ma ci sono alcune sottigliezze che potrebbero confondere. È preferibile evitarla (a meno che non sappiate esattamente quello che state facendo).

Esecuzione condizionale

Se volete scrivere programmi utili, vi capiterà spesso di dover controllare se si verificano determinate condizioni, e di variare di conseguenza il comportamento del programma. Le **istruzioni condizionali** servono proprio a questo. La forma più semplice di istruzione condizionale è l'istruzione `if` ("se" in inglese):

```
if x > 0:
    print('x è positivo')
```

L'espressione booleana dopo l'istruzione `if` è chiamata **condizione**. Se risulta vera, viene eseguita l'istruzione indentata che segue sulla riga successiva. Altrimenti, non succede nulla.

L'istruzione `if` ha la stessa struttura che abbiamo già visto nel caso delle definizioni di funzione: un'intestazione seguita da un corpo indentato. Le istruzioni come questa vengono chiamate **istruzioni composte**.

Non c'è limite al numero di istruzioni che possono essere scritte nel corpo, ma deve sempre essercene almeno una. Talvolta può servire che il corpo sia privo di istruzioni (di solito quando c'è del codice ancora da scrivere); in questo caso potete usare l'istruzione `pass`, che serve solo da segnaposto temporaneo e nulla più:

```
if x < 0:
    pass          # scrivere cosa fare con i valori negativi!
```

Esecuzione alternativa

Una seconda forma di istruzione `if` è l'**esecuzione alternativa**, dove esistono due possibili azioni, e il valore della condizione determina quale delle due azioni debba essere eseguita e quale no. La sintassi è:

```
if x > 0:
    print('x è positivo')
else:
    print('x è zero o negativo')
```

Se `x` è maggiore di zero, significa che `x` è un numero positivo, e il programma mostra il messaggio appropriato. Altrimenti (`else`), se la condizione è falsa, viene eseguito il secondo blocco di istruzioni. Dato che la condizione deve essere necessariamente o vera o falsa, sarà sempre eseguita una sola delle due alternative. Queste sono dette **ramificazioni**, perché rappresentano dei bivi nel flusso di esecuzione del programma.

Condizioni in serie

A volte è necessario considerare più di due possibili sviluppi, e occorre che nel programma ci siano più di due ramificazioni. Un modo per esprimere questo tipo di calcolo sono le **condizioni in serie**:

```
if x < y:
    print('x è minore di y')
elif x > y:
    print('x è maggiore di y')
else:
    print('x e y sono uguali')
```

`elif` è l'abbreviazione di *else if*, che in inglese significa "altrimenti se". Anche stavolta verrà eseguito solo uno dei tre rami, a seconda dell'esito del confronto tra `x` e `y`. Non c'è alcun limite al numero di istruzioni `elif`. Se esiste una clausola `else`, deve essere scritta per ultima, ma non è obbligatoria; il ramo corrispondente viene eseguito solo quando tutte le condizioni precedenti sono false.

```
if scelta == 'a':
    disegna_a()
elif scelta == 'b':
    disegna_b()
elif scelta == 'c':
    disegna_c()
```

Le condizioni vengono controllate nell'ordine dall'alto al basso: se la prima è falsa, viene controllata la seconda e così via. Non appena una condizione risulta vera, viene eseguito il ramo corrispondente e l'istruzione termina. Anche se risultassero vere altre condizioni successive, viene eseguita sempre e soltanto la prima che risulta vera.

Condizioni nidificate

Si può anche inserire un'istruzione condizionale nel corpo di un'altra istruzione condizionale. Possiamo dunque scrivere l'esempio del paragrafo precedente anche in questo modo:

```
if x == y:
    print('x e y sono uguali')
else:
    if x < y:
        print('x è minore di y')
    else:
        print('x è maggiore di y')
```

La condizione più esterna contiene due rami: il primo contiene un'istruzione semplice, il secondo un'altra istruzione `if` che a sua volta ha due ramificazioni. Entrambi i rami del secondo `if` sono istruzioni di stampa, ma potrebbero anche contenere a loro volta ulteriori istruzioni condizionali.

Anche se l'indentazione delle istruzioni aiuta ad evidenziare la struttura, le **condizioni nidificate** diventano rapidamente difficili da leggere, quindi è meglio usarle con moderazione.

Gli operatori logici permettono spesso di semplificare le istruzioni condizionali nidificate. Il codice seguente può essere riscritto usando un'unica condizione:

```
if 0 < x:
    if x < 10:
        print('x è un numero positivo a una cifra.')
```

Infatti, dato che l'istruzione di stampa è eseguita solo se si verificano entrambe le condizioni, possiamo ottenere lo stesso risultato usando l'operatore `and`:

```
if 0 < x and x < 10:
    print('x è un numero positivo a una cifra.')
```

Per una condizione di questo tipo, Python consente anche un'opzione sintattica più concisa:

```
if 0 < x < 10:
    print('x è un numero positivo a una cifra.')
```

Debug

Quando si verifica un errore di sintassi o di runtime, il messaggio d'errore contiene molte informazioni, ma può essere sovrabbondante. Di solito le parti più utili sono:

- Che tipo di errore era, e
- Dove si è verificato.

Gli errori di sintassi di solito sono facili da trovare, con qualche eccezione. Gli spaziatori possono essere insidiosi, perché spazi e tabulazioni non sono visibili e non siamo abituati a tenerne conto.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In questo esempio, il problema è che la seconda riga è erroneamente indentata di uno spazio, mentre dovrebbe stare al margine sinistro. Ma il messaggio di errore punta su `y`, portando fuori strada. In genere, i messaggi di errore indicano dove il problema è venuto a galla, ma il vero errore potrebbe essere in un punto precedente del codice, a volte anche nella riga precedente.

Lo stesso vale per gli errori di runtime.

Supponiamo di voler calcolare un rapporto segnale/rumore in decibel. La formula è $SNR_{db} = 10 \log_{10}(P_{segnale}/P_{rumore})$. In Python si può scrivere:

```
import math
potenza_segnale = 9
potenza_rumore = 10
rapporto = potenza_segnale // potenza_rumore
decibel = 10 * math.log10(rapporto)
print(decibel)
```

Se avviate questo programma, compare un messaggio di errore.

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibel = 10 * math.log10(rapporto)
ValueError: math domain error
```

Il messaggio punta alla riga 5, ma lì non c'è niente di sbagliato. Per trovare il vero errore, può essere utile stampare il valore di `rapporto`, che risulta essere 0. Il problema sta nella riga 4, perché calcola una divisione intera anziché una normale divisione.

Prendetevi la briga di leggere attentamente i messaggi di errore, ma non date per scontato che tutto quello che dicono sia esatto.

Glossario

espressione booleana: Espressione il cui valore è o vero (`True`) o falso (`False`).

operatore di confronto: Operatore che confronta due valori detti operandi: `==`, `!=`, `>`, `<`, `>=`, e `<=`.

operatore logico: Operatore che unisce delle espressioni booleane: `and`, `or`, e `not`.

istruzione condizionale: Istruzione che controlla il flusso di esecuzione del programma, variandolo a seconda di determinate condizioni.

condizione: Espressione booleana in un'istruzione condizionale che determina quale ramificazione sarà eseguita.

istruzione composta: Istruzione che consiste di un'intestazione e di un corpo. L'intestazione deve terminare con i due punti (:) e il corpo deve essere indentato rispetto ad essa.

ramificazione: Uno dei blocchi di istruzioni alternative presenti in un'istruzione condizionale.

condizioni in serie: Istruzione condizionale con una serie di ramificazioni alternative.

condizione nidificata (o annidata): Un'istruzione condizionale inserita in una ramificazione di un'altra istruzione condizionale.