3.1 L'enunciato `if`

L'enunciato `if` consente a un programma di compiere azioni diverse in relazione alla natura dei dati che vengono elaborati.

Per prendere una decisione all'interno di un programma si usa l'enunciato `if`: quando una determinata condizione è verificata, viene eseguito un certo insieme di enunciati, altrimenti ne viene eseguito un altro.

Vediamo un esempio di utilizzo dell'enunciato `if`. In molti Paesi, il numero 13 è considerato sfortunato, per cui, per rispetto nei confronti delle persone superstiziose, spesso i progettisti di edifici fanno in modo che il tredicesimo piano non sia presente: il dodicesimo piano è seguito dal quattordicesimo. Ovviamente, il tredicesimo piano non viene lasciato vuoto, né, come credono i teorici della cospirazione, viene destinato a custodire uffici e laboratori di ricerca segreti: banalmente, viene chiamato "quattordicesimo piano". Di conseguenza, i computer che controllano gli ascensori devono tener conto di questa fobia e modificare in modo coerente i numeri di tutti i piani superiori al 13.

Cerchiamo di simulare questa procedura in Python. Chiederemo all'utente di digitare il numero del piano desiderato e, poi, calcoleremo il numero effettivo corrispondente: quando il dato in ingresso è superiore a 13, per ottenere il numero di piano effettivo dobbiamo decrementare di un'unità il valore acquisito.

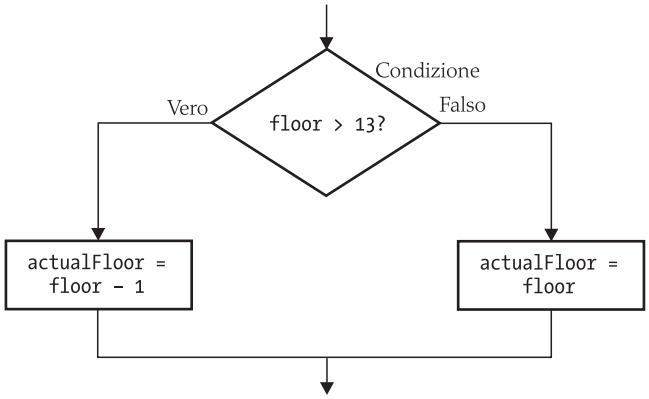
Se, ad esempio, l'utente digita il numero 20, il programma calcola che il numero di piano effettivo è 19, mentre, se il numero digitato è inferiore a 13, il programma visualizza semplicemente tale numero.

```
actualFloor = 0

if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
```

Il diagramma di flusso della Figura 1 illustra il comportamento della diramazione.

Figura 1
Diagramma di flusso per l'enunciato if



Nel nostro esempio entrambi i rami dell'enunciato `if` contengono un unico enunciato, ma in ciascun ramo si possono inserire anche più enunciati. Inoltre, a volte non c'è alcuna azione da compiere nel ramo `else` (cioè "altrimenti") dell'enunciato, come in questo esempio:

Sintassi 3.1 Enunciato `if`

Sintassi

```
if condizione :
   enunciati
```

```
if condizione :
   enunciati1
else :
   enunciati2
```

Esempio

Una *condizione* che è vera o falsa. Spesso contiene operatori relazionali (`== != < <= > >=`).

Il carattere "due punti" identifica un enunciato composto.

```
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
```

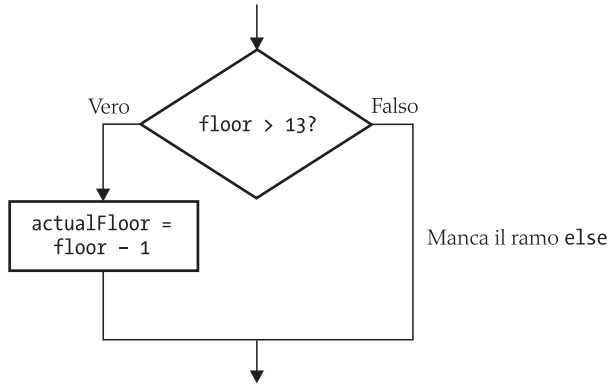
Se la *condizione* è vera, gli enunciati presenti in questo ramo vengono eseguiti in sequenza; se la *condizione* è falsa, vengono ignorati.

Se la *condizione* è falsa, gli enunciati presenti in questo ramo vengono eseguiti in sequenza; se la *condizione* è vera, vengono ignorati.

Le clausole `if` e `else` devono essere incolonnate correttamente.

Se il ramo `else` non contiene azioni, la clausola va omessa.

Figura 2
Diagramma di flusso per l'enunciato if senza il ramo else




```
actualFloor = floor
if floor > 13 :
   actualFloor = actualFloor - 1
```

Il diagramma di flusso corrispondente è riportato nella Figura 2.

Il programma seguente usa enunciati `if`: chiede all'utente qual è il numero del piano desiderato e visualizza il corrispondente numero effettivo del piano.

File ch03/sec01/elevatorsim.py

```
1  ##
2  #  Questo programma simula il pannello di comando di un
3  #  ascensore che salta il tredicesimo piano.
4
```

```
# Acquisisce il numero del piano come numero intero.
floor = int(input("Floor: "))

# Corregge il numero del piano, se necessario.
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor

# Visualizza il risultato.
print("The elevator will travel to the actual floor", actualFloor)
```

Esecuzione del programma

```
Floor: 20
The elevator will travel to the actual floor 19
```

Gli enunciati composti sono costituiti da una intestazione e da un blocco di enunciati.

Le istruzioni Python usate fino ad ora sono enunciati semplici, che sono contenuti in un'unica riga (o che proseguono esplicitamente nella riga successiva, come visto nella sezione Argomenti avanzati 2.3). Alcuni costrutti sintattici di Python, invece, sono **enunciati composti**, che possono estendersi su più righe e sono costituiti da una **intestazione** (*header*) e da un **blocco di enunciati** (*statement block*). L'enunciato `if` è un esempio di enunciato composto.

```
if totalSales > 100.0 :  # L'intestazione termina con "due punti"
   discount = totalSales * 0.05  # Le righe di un blocco sono incolonnate
   totalSales = totalSales - discount
   print("You received a discount of", discount)
```

Gli enunciati di un blocco devono iniziare con lo stesso livello di rientro verso destra.

Al termine dell'intestazione di un enunciato composto è necessaria la presenza di un carattere "due punti", mentre il blocco di enunciati è un gruppo di uno o più enunciati incolonnati a sinistra allo stesso modo e posizionati più a destra rispetto all'intestazione. Un blocco di enunciati inizia nella riga che segue l'intestazione e termina in corrispondenza del primo enunciato che inizia più a sinistra della colonna in cui iniziano tutti gli enunciati del blocco stesso. Come spostamento verso destra degli enunciati di un blocco si può usare qualsiasi numero di spazi, ma tutti gli enunciati del blocco devono iniziare nella stessa colonna. Si noti come i commenti non siano enunciati e, quindi, possano iniziare in qualsiasi posizione.

I blocchi di enunciati, che possono comparire anche uno dentro l'altro (e in tal caso si chiamano "blocchi annidati", dall'inglese *nested*), evidenziano dal punto di vista sintattico che uno o più enunciati fanno parte di un determinato enunciato composto. Nel caso del costrutto `if`, il blocco di enunciati specifica le istruzioni che verranno eseguite se la condizione è vera (e che verranno ignorate se la condizione è falsa).

Errori comuni 3.1

Caratteri di tabulazione

Il codice strutturato a blocchi richiede che enunciati annidati vengano spostati verso destra di uno o più livelli:

```
if totalSales > 100.0 :
¦  discount = totalSales * 0.05
¦  totalSales = totalSales - discount
¦  print("You received a discount of $%.2f" % discount)
else :
¦  diff = 100.0 - totalSales
¦  if diff < 10.0 :
¦  ¦  print("If you purchase our item of the day you get a 5% discount.")
¦  else :
¦  ¦  print("You need to spend $%.2f more to receive a 5% discount." % diff)
¦  ¦  ¦
¦  ¦  ¦
¦  ¦  ¦
0  1  2  Livelli di rientro
```

In Python, la struttura a blocchi del codice è parte della sintassi: l'incolonnamento specifica quali enunciati fanno parte di un determinato blocco.

Come si può spostare il cursore dalla colonna più a sinistra al livello di rientro appropriato? È assolutamente ragionevole premere la barra spaziatrice il numero di volte sufficiente, ma molti programmatori preferiscono usare il tasto di tabulazione (*Tab*), che sposta il cursore alla successiva posizione di tabulazione definita nel programma di scrittura (*editor*), che a volte ha anche un'opzione che consente di inserire i caratteri di tabulazione in modo automatico (*autoindent*).

Nonostante i caratteri di tabulazione siano comodi, alcuni editor li usano anche per allineare bene il testo e questa non è una buona idea. Python è piuttosto esigente in merito all'incolonnamento degli enunciati all'interno di un blocco: tutti devono essere incolonnati mediante spazi o caratteri di tabulazione, ma non usando una combinazione dei due. Inoltre, i caratteri di tabulazione possono costituire un problema quando si invia un file che contiene tabulazioni a un'altra persona o a una stampante. Non esiste uno standard per l'ampiezza di un carattere di tabulazione e alcuni software li ignorano completamente. Quindi, per concludere, è preferibile archiviare i propri file in modo che contengano spazi al posto dei caratteri di tabulazione: per questo motivo, la maggior parte degli editor può convertire le tabulazioni in spazi in modo automatico.

Cercate questa utile opzione nella documentazione del vostro ambiente di sviluppo e attivatela.

Suggerimenti per la programmazione 3.1

Evitare duplicazioni nelle diramazioni

Controllate sempre se le due diramazioni di un enunciato `if` contengono, come nell'esempio seguente, *codice duplicato*. In tal caso, spostatelo al di fuori dell'enunciato `if`.

```
if floor > 13 :
   actualFloor = floor - 1
   print("Actual floor:", actualFloor)
else :
   actualFloor = floor
   print("Actual floor:", actualFloor)
```

L'enunciato di visualizzazione è esattamente il medesimo in entrambe le diramazioni. Non si tratta di un errore: il programma viene eseguito correttamente. Tuttavia, si può semplificare il codice spostando l'enunciato duplicato, in questo modo:

```
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
print("Actual floor:", actualFloor)
```

L'eliminazione delle duplicazioni è particolarmente importante quando i programmi devono essere manutenuti per lungo tempo: quando due insiemi di enunciati producono il medesimo effetto, può facilmente accadere che un programmatore modifichi un insieme senza modificare l'altro.

.py

Argomenti avanzati 3.1

Espressioni condizionali

Python dispone di un operatore condizionale, con questa forma:

valore$_1$ `if` *condizione* `else` *valore*$_2$

Il valore di tale espressione è: *valore*$_1$ se la *condizione* è vera, *valore*$_2$ se la *condizione* è falsa. Ad esempio, possiamo calcolare il numero del piano effettivo in questo modo:

```
actualFloor = floor - 1 if floor > 13 else floor
```

Tale codice è equivalente al seguente:

```
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
```

Si noti che un'espressione condizionale è un singolo enunciato, che deve, quindi, svilupparsi su un'unica riga, oppure essere esplicitamente proseguito sulla riga successiva (come detto nella sezione Argomenti avanzati 2.3). Ancora, si noti che non serve un carattere "due punti", perché un'espressione condizionale non è un enunciato composto.

Un'espressione condizionale può essere utilizzata in qualsiasi punto in cui sia prevista la presenza di un valore, come in questo esempio:

```
print("Actual floor:", floor - 1 if floor > 13 else floor)
```

In questo libro non useremo espressioni condizionali, ma si tratta di un costrutto sintattico utile, che troverete in alcuni programmi Python.

3.2 Operatori relazionali

In questo paragrafo imparerete a confrontare numeri e stringhe in Python.

Per confrontare numeri e stringhe si usano gli operatori relazionali (`<` `<=` `>` `>=` `==` `!=`).

Ogni enunciato `if` contiene una condizione che, in molti casi, richiede un confronto tra due valori, come negli esempi precedenti, dove abbiamo verificato se `floor > 13`. Tale confronto viene effettuato tramite l'operatore `>`, che viene chiamato **operatore relazionale**. Il linguaggio Python dispone di sei operatori relazionali, riportati nella Tabella 1.

Tabella 1
Operatori relazionali

Operatore Python	Notazione matematica	Descrizione
>	>	Maggiore
>=	≥	Maggiore o uguale
<	<	Minore
<=	≤	Minore o uguale
==	=	Uguale
!=	≠	Diverso

Come si può notare, soltanto due operatori relazionali di Python hanno una notazione identica a quella degli analoghi operatori matematici. Le tastiere dei computer non hanno i simboli ≥, ≤ o ≠, ma gli operatori `>=`, `<=` e `!=` sono facili da ricordare, perché hanno un aspetto simile, mentre l'operatore di uguaglianza, `==`, è fonte di confusione per chi si avvicina per la prima volta al linguaggio Python.

In Python, il simbolo `=` ha già un significato: rappresenta l'assegnazione. Quindi, per indicare una verifica di uguaglianza si usa l'operatore `==`:

```
floor = 13  # Assegna 13 a floor
if floor == 13 :  # Verifica se floor è uguale a 13
```

Dovete ricordarvi di usare `==` nelle verifiche e `=` al di fuori di esse.

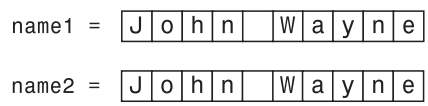
In Python, anche le stringhe possono essere confrontate tra loro usando i medesimi operatori relazionali. Ad esempio, per verificare se due stringhe sono uguali si usa l'operatore `==`:

```
if name1 == name2 :
   print("The strings are identical.")
```

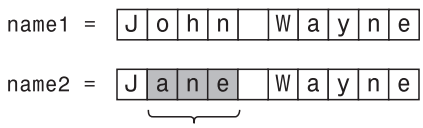
mentre per verificare se sono diverse si usa l'operatore `!=`:

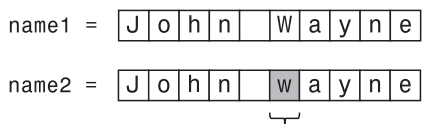
```
if name1 != name2 :
   print("The strings are not identical.")
```

Perché due stringhe siano uguali, devono avere la stessa lunghezza e contenere la stessa sequenza di caratteri:



Se anche un solo carattere è diverso, le due stringhe non sono uguali:





Gli operatori relazionali presentati nella Tabella 1 hanno una precedenza inferiore a quella degli operatori aritmetici. Questo significa che è possibile scrivere espressioni aritmetiche ai lati degli operatori relazionali senza che ci sia bisogno di parentesi. Per esempio, nell'espressione:

```
floor - 1 < 13
```

vengono prima valutate entrambe le espressioni ai lati dell'operatore `<` (cioè `floor - 1` e `13`), poi vengono confrontati i risultati. L'Appendice A riporta una tabella con gli operatori Python e le loro precedenze.

La Tabella 2 riassume le modalità di confronto di valori in Python, mentre il programma seguente esegue confronti usando espressioni logiche.

File ch03/sec02/compare.py

```
##
#  Questo programma effettua confronti tra numeri
#  e tra stringhe.

from math import sqrt

# Confronti tra numeri interi
m = 2
n = 4

if m * m == n :
   print("2 times 2 is four.")

# Confronti tra numeri in virgola mobile
x = sqrt(2)
y = 2.0

if x * x == y :
   print("sqrt(2) times sqrt(2) is 2")
else :
   print("sqrt(2) times sqrt(2) is not two but %.18f" % (x * x))

EPSILON = 1E-14
if abs(x * x - y) < EPSILON :
   print("sqrt(2) times sqrt(2) is approximately 2")

# Confronti tra stringhe
s = "120"
t = "20"

if s == t :
   comparison = "is the same as"
else :
   comparison = "is not the same as"

print("The string '%s' %s the string '%s'." % (s, comparison, t))

```

	Espressione	Valore	Commento
	`3 <= 4`	Vero	3 è minore di 4; <= verifica se "è minore di o uguale a".
	`3 =< 4`	Errore	L'operatore "minore di o uguale a" è <=, non =<. Il simbolo "minore di" figura per primo.
	`3 > 4`	Falso	> è il contrario di <=.
	`4 < 4`	Falso	La parte sinistra deve essere strettamente minore della parte destra perché il risultato sia vero.
	`4 <= 4`	Vero	Le due espressioni confrontate sono uguali; **<=** verifica se "è minore di o uguale a".
	`3 == 5 - 2`	Vero	== verifica l'uguaglianza.
	`3 != 5 - 1`	Vero	!= verifica l'ineguaglianza, ed è vero che 3 è diverso da 5 – 1.
	`3 = 6 / 2`	Errore	Per le verifiche di uguaglianza bisogna usare ==.
	`1.0 / 3.0 == 0.333333333`	Falso	Anche se i valori sono molto simili, non sono esattamente uguali (Errori comuni 3.2).
	`"10" > 5`	Errore	Non si può confrontare una stringa con un numero.

Tabella 2
Esempi con operatori relazionali

```
38 u = "1" + t
39 if s != u :
40    comparison = "not "
41 else :
42    comparison = ""
43
44 print("The strings '%s' and '%s' are %sidentical." % (s, u, comparison))
```

Esecuzione del programma

```
2 times 2 is four.
sqrt(2) times sqrt(2) is not two but 2.000000000000000444
sqrt(2) times sqrt(2) is approximately 2
The string '120' is not the same as the string '20'.
The strings '120' and '120' are identical.
```

Errori comuni 3.2

Confronto esatto di numeri in virgola mobile

I numeri in virgola mobile hanno una precisione limitata e i calcoli possono inevitabilmente introdurre errori di arrotondamento, che vanno tenuti in considerazione quando si confrontano due numeri in virgola mobile. Ad esempio, il codice seguente moltiplica la radice quadrata di 2 per se stessa e, in teoria, il risultato dovrebbe essere 2:

```
from math import sqrt

r = sqrt(2.0)
if r * r == 2.0 :
   print("sqrt(2.0) squared is 2.0")
else :
   print("sqrt(2.0) squared is not 2.0 but", r * r)
```

Questo programma visualizza:

```
sqrt(2.0) squared is not 2.0 but 2.0000000000000004
```

Nella maggior parte dei casi non ha senso confrontare numeri in virgola mobile per verificare se sono esattamente uguali: si dovrebbe verificare se sono *abbastanza simili*, cioè se il valore assoluto della loro differenza è inferiore a una determinata soglia. In termini matematici, scriveremo che x e y sono abbastanza simili se

$$|x - y| < \varepsilon$$

dove ε è un numero molto piccolo (ε è la lettera greca "epsilon", che viene solitamente utilizzata in matematica per indicare una quantità molto piccola). Nel confrontare numeri in virgola mobile, solitamente si usa $\varepsilon = 10^{-14}$:

```
from math import sqrt

EPSILON = 1E-14
r = sqrt(2.0)
if abs(r * r - 2.0) < EPSILON :
   print("sqrt(2.0) squared is approximately 2.0")
```

.py

Argomenti avanzati 3.2

Ordinamento lessicografico di stringhe

Se due stringhe non sono identiche, potreste comunque voler conoscere la relazione in cui si pongono: gli operatori relazionali di Python confrontano le stringhe secondo l'ordinamento "lessicografico", un ordinamento molto simile a quello con cui le parole vengono disposte in un dizionario. Se:

```
string1 < string2
```

allora nel dizionario la stringa `string1` precede la stringa `string2`. Ad esempio, questo è il caso che accade quando `string1` è `"Harry"` e `string2` è `"Hello"`. Se, invece:

```
string1 > string2
```

allora nel dizionario la stringa `string1` segue la stringa `string2`.

Come detto nel paragrafo precedente, se:

```
string1 == string2
```

allora le stringhe `string1` e `string2` sono uguali.

Ci sono, però, alcune differenze tecniche tra l'ordinamento delle parole in un dizionario e l'ordinamento lessicografico delle stringhe in Python.

In Python:

- tutte le lettere maiuscole precedono tutte le lettere minuscole (ad esempio, `"Z"` precede `"a"`);
- il carattere "spazio" precede tutti i caratteri visualizzabili;
- le cifre precedono le lettere;
- per l'ordinamento dei segni di punteggiatura, si veda l'Appendice D.