Capitolo 8

Stringhe

Le stringhe non sono valori come gli interi, i float e i booleani. Una stringa è una **sequenza**, vale a dire un insieme ordinato di valori di altra natura. In questo capitolo vedrete come si accede ai caratteri che compongono una stringa e imparerete alcuni metodi che le stringhe espongono.

8.1 Una stringa è una sequenza

Una stringa è una sequenza di caratteri. Potete accedere ai singoli caratteri usando gli operatori parentesi quadre:

```
>>> frutto = 'banana'
>>> lettera = frutto[1]
```

La seconda istruzione seleziona il carattere numero 1 della variabile `frutto` e lo assegna alla variabile `lettera`.

L'espressione all'interno delle parentesi quadre è chiamato **indice**. L'indice è un numero intero che indica (di qui il nome) il carattere della sequenza che desiderate estrarre.

Ma il risultato potrebbe lasciarvi perplessi:

```
>>> lettera
a
```

Per la maggior parte delle persone, la prima lettera di `'banana'` è `b`, non `a`. Ma per gli informatici, premesso che l'indice è la posizione a partire dall'inizio della stringa, la posizione della prima lettera è considerata la numero zero, non uno.

```
>>> lettera = frutto[0]
>>> lettera
b
```

Quindi `b` è la "zero-esima" lettera di `'banana'`, `a` è la prima lettera ("1-esima"), e `n` è la seconda ("2-esima") lettera.

Potete usare come indice qualsiasi espressione, compresi variabili e operatori:

```
>>> i = 1
>>> frutto[i]
'a'
>>> frutto[i+1]
'n'
```

Tuttavia, il valore risultante deve essere un intero. Altrimenti succede questo:

```
>>> lettera = frutto[1.5]
TypeError: string indices must be integers
```

8.2 `len`

`len` è una funzione predefinita che restituisce il numero di caratteri contenuti in una stringa:

```
>>> frutto = 'banana'
>>> len(frutto)
6
```

Per estrarre l'ultimo carattere di una stringa, si potrebbe pensare di scrivere qualcosa del genere:

```
>>> lunghezza = len(frutto)
>>> ultimo = frutto[lunghezza]
IndexError: string index out of range
```

La ragione dell'`IndexError` è che non c'è nessuna lettera in `'banana'` con indice 6. Siccome partiamo a contare da zero, le sei lettere sono numerate da 0 a 5. Per estrarre l'ultimo carattere, dobbiamo perciò sottrarre 1 da `lunghezza`:

```
>>> ultimo = frutto[lunghezza-1]
>>> ultimo
'a'
```

Oppure, possiamo usare utilmente gli indici negativi, che contano a ritroso dalla fine della stringa: l'espressione `frutto[-1]` ricava l'ultimo carattere della stringa, `frutto[-2]` il penultimo carattere, e così via.

8.3 Attraversamento con un ciclo `for`

Parecchi tipi di calcolo comportano l'elaborazione di una stringa, un carattere per volta. Spesso iniziano dal primo carattere, selezionano un carattere per volta, eseguono una certa operazione e continuano fino alla fine della stringa. Questo tipo di elaborazione è detta **attraversamento**. Un modo per scrivere un attraversamento è quello di usare un ciclo `while`:

```
indice = 0
while indice < len(frutto):
    lettera = frutto[indice]
    print(lettera)
    indice = indice + 1
```

Questo ciclo attraversa tutta la stringa e ne mostra le singole lettere, ciascuna su una riga separata. La condizione del ciclo è `indice < len(frutto)`, per cui quando `indice` è uguale alla lunghezza della stringa, la condizione diventa falsa e il corpo del ciclo non viene più eseguito. L'ultimo carattere a cui si accede è quello di indice `len(frutto)-1`, cioè l'ultimo carattere della stringa.

Come esercizio, scrivete una funzione che riceva una stringa come argomento e ne stampi i singoli caratteri, uno per riga, partendo dall'ultimo a ritroso.

Un altro modo di scrivere un attraversamento è usare un ciclo `for`:

```
for lettera in frutto:
    print(lettera)
```

Ad ogni ciclo, il successivo carattere della stringa viene assegnato alla variabile `lettera`. Il ciclo continua finché non rimangono più caratteri da analizzare.

L'esempio che segue illustra come usare il concatenamento (addizione di stringhe) e un ciclo `for` per generare una serie alfabetica (cioè, disposta in ordine alfabetico). Nel libro *Make Way for Ducklings* di Robert McCloskey, ci sono degli anatroccoli che si chiamano Jack, Kack, Lack, Mack, Nack, Ouack, Pack, e Quack. Questo ciclo restituisce i nomi in ordine:

```
prefissi = 'JKLMNOPQ'
suffisso = 'ack'

for lettera in prefissi:
    print(lettera + suffisso)
```

Il risultato del programma è:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

È evidente che non è del tutto giusto, dato che "Ouack" e "Quack" sono scritti in modo errato.

Provate a modificare il programma per correggere questo errore.

8.4 Slicing

Un segmento o porzione di stringa è chiamato **slice**. L'operazione di selezione di una porzione di stringa è simile alla selezione di un carattere, ed è detta **slicing**:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

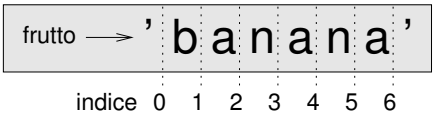

Figura 8.1: Indici di slicing.

L'operatore `[n:m]` restituisce la porzione di stringa nell'intervallo compreso tra l'"n-esimo" carattere incluso fino all'"m-esimo" escluso. Questo comportamento è poco intuitivo, e per tenerlo a mente può essere d'aiuto immaginare gli indici puntare *tra* i caratteri, come spiegato nella Figura 8.1.

Se non è specificato il primo indice (quello prima dei due punti :), la porzione parte dall'inizio della stringa. Se manca il secondo indice, la porzione arriva fino in fondo alla stringa:

```
>>> frutto = 'banana'
>>> frutto[:3]
'ban'
>>> frutto[3:]
'ana'
```

Se il primo indice è maggiore o uguale al secondo, il risultato è una **stringa vuota**, rappresentata da due apici consecutivi.

```
>>> frutto = 'banana'
>>> frutto[3:3]
''
```

Una stringa vuota non contiene caratteri e ha lunghezza 0, ma a parte questo è a tutti gli effetti una stringa come le altre.

Proseguendo con l'esempio, data una stringa di nome `frutto`, secondo voi che cosa significa `frutto[:]`? Provate a vedere.

8.5 Le stringhe sono immutabili

Per sostituire un carattere all'interno di una stringa, potreste pensare di utilizzare l'operatore `[]` sul lato sinistro di un'assegnazione, per esempio così:

```
>>> saluto = 'Ciao, mondo!'
>>> saluto[0] = 'M'
TypeError: 'str' object does not support item assignment
```

L'"oggetto" (*object*) in questo caso è la stringa, e l'"elemento" (*item*) è il carattere che avete tentato di assegnare. Per ora, consideriamo un oggetto come la stessa cosa di un valore, ma più avanti (Paragrafo 10.10) puntualizzeremo meglio questa definizione.

La ragione dell'errore è che le stringhe sono **immutabili**, in altre parole, non è consentito cambiare una stringa esistente. La cosa migliore da fare è creare una nuova stringa, variante dell'originale:

```
>>> saluto = 'Ciao, mondo!'
>>> nuovo_saluto = 'M' + saluto[1:]
>>> nuovo_saluto
'Miao, mondo!'
```

Questo esempio concatena una nuova prima lettera con la restante porzione di `saluto`. Non ha alcun effetto sulla stringa di origine, che resta invariata.

8.6 Ricerca

Cosa fa la funzione seguente?

```
def trova(parola, lettera):
    indice = 0
    while indice < len(parola):
        if parola[indice] == lettera:
            return indice
        indice = indice + 1
    return -1
```

In un certo senso, `trova` è l'inverso dell'operatore `[]`. Anziché prendere un indice ed estrarre il carattere corrispondente, prende un carattere e trova l'indice in corrispondenza del quale appare il carattere. Se non trova il carattere indicato nella parola data, la funzione restituisce `-1`.

Per la prima volta incontriamo l'istruzione `return` all'interno di un ciclo. Se `parola[indice] == lettera`, la funzione interrompe il ciclo e ritorna immediatamente, restituendo `indice`.

Se il carattere non compare nella stringa data, il programma termina il ciclo normalmente e restituisce `-1`.

Questo schema di calcolo—attraversare una sequenza e ritornare quando si trova ciò che si sta cercando—è chiamato **ricerca**.

Come esercizio, modificate la funzione `trova` in modo che richieda un terzo parametro, che rappresenta la posizione da cui si deve cominciare la ricerca all'interno della stringa `parola`.

8.7 Cicli e contatori

Il programma seguente conta il numero di volte in cui la lettera `a` compare in una stringa:

```
parola = 'banana'
conta = 0
for lettera in parola:
    if lettera == 'a':
        conta = conta + 1
print(conta)
```

Si tratta di un altro schema di calcolo chiamato **contatore**. La variabile `conta` è inizializzata a 0, quindi incrementata di uno per ogni volta che viene trovata una `a`. Al termine del ciclo, `conta` contiene il risultato: il numero totale di lettere `a` nella stringa.

Come esercizio, incapsulate questo codice in una funzione di nome `conta`, e generalizzatela in modo che accetti come argomenti sia la stringa che la lettera da cercare. Quindi, riscrivete questa funzione in modo che, invece di attraversare completamente la stringa, faccia uso della versione a tre parametri di `trova`, vista nel precedente paragrafo.

8.8 Metodi delle stringhe

Le stringhe espongono dei metodi che permettono di effettuare molte utili operazioni. Un **metodo** è simile a una funzione—riceve argomenti e restituisce un valore—ma la sintassi è diversa. Prendiamo ad esempio il metodo `upper`, che prende una stringa e crea una nuova stringa di tutte lettere maiuscole.

Al posto della sintassi delle funzioni, `upper(parola)`, si usa la sintassi dei metodi, `parola.upper()`.

```
>>> parola = 'banana'
>>> nuova_parola = parola.upper()
>>> nuova_parola
BANANA
```

Questa forma di notazione a punto, in inglese *dot notation*, specifica il nome del metodo, `upper`, preceduto dal nome della stringa a cui va applicato il metodo, `parola`. Le parentesi vuote indicano che il metodo non ha argomenti.

La chiamata di un metodo è detta **invocazione**; nel nostro caso, diciamo che stiamo invocando `upper` su `parola`.

Visto che ci siamo, esiste un metodo delle stringhe chiamato `find` che è molto simile alla funzione che abbiamo scritto prima:

```
>>> parola = 'banana'
>>> indice = parola.find('a')
>>> indice
1
```

In questo esempio, abbiamo invocato `find` su `parola` e abbiamo passato come parametro la lettera che stiamo cercando.

In realtà, il metodo `find` è più generale della nostra funzione: può ricercare anche sottostringhe e non solo singoli caratteri:

```
>>> parola.find('na')
2
```

Di default, `find` parte dall'inizio della stringa, ma può ricevere come secondo argomento l'indice da cui partire:

```
>>> parola.find('na', 3)
4
```

Questo è un esempio di **argomento opzionale**; `find` può anche avere un terzo argomento opzionale, l'indice in corrispondenza del quale fermarsi:

```
>>> nome = 'bob'
>>> nome.find('b', 1, 2)
-1
```

In quest'ultimo caso la ricerca fallisce, perché `b` non è compreso nell'intervallo da `1` a `2`, in quanto `2` si considera escluso. Questo comportamento rende `find` coerente con l'operatore di slicing.

8.9 L'operatore in

La parola in è un operatore booleano che confronta due stringhe e restituisce True se la prima è una sottostringa della seconda:

```
>>> 'a' in 'banana'
True
>>> 'seme' in 'banana'
False
```

Ad esempio, la funzione che segue stampa tutte le lettere di parola1 che compaiono anche in parola2:

```
def in_entrambe(parola1, parola2):
    for lettera in parola1:
        if lettera in parola2:
            print(lettera)
```

Con qualche nome di variabile scelto bene, Python a volte si legge quasi come fosse un misto di inglese e italiano: "per (ogni) lettera in parola1, se (la) lettera (è) in parola2, stampa (la) lettera."

Ecco cosa succede se paragonate carote e patate:

```
>>> in_entrambe('carote', 'patate')
a
t
e
```

8.10 Confronto di stringhe

Gli operatori di confronto funzionano anche sulle stringhe. Per controllare se due stringhe sono uguali:

```
if parola == 'banana':
    print('Tutto ok, banane.')
```

Altri operatori di confronto sono utili per mettere le parole in ordine alfabetico:

```
if parola < 'banana':
    print('La tua parola,' + parola + ', viene prima di banana.')
elif parola > 'banana':
    print('La tua parola,' + parola + ', viene dopo banana.')
else:
    print('Tutto ok, banane.')
```

Attenzione che Python non gestisce le lettere maiuscole e minuscole come siamo abituati: in un confronto, le lettere maiuscole vengono sempre prima di tutte le minuscole, così che:

```
La tua parola, Papaya, viene prima di banana.
```

Questo problema si risolve facilmente convertendo le stringhe in un formato standard, ad esempio tutte lettere minuscole, prima di effettuare il confronto.

8.11 Debug

Quando usate gli indici per l'attraversamento dei valori di una sequenza, non è facile determinare bene l'inizio e la fine. Ecco una funzione che dovrebbe confrontare due parole e restituire `True` quando una parola è scritta al contrario dell'altra, ma contiene due errori:

```
def al_contrario(parola1, parola2):
    if len(parola1) != len(parola2):
        return False

    i = 0
    j = len(parola2)

    while j > 0:
        if parola1[i] != parola2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

La prima istruzione `if` controlla se le parole sono della stessa lunghezza. Se non è così, possiamo restituire immediatamente `False`. Altrimenti, per il resto della funzione, possiamo presupporre che le parole abbiano pari lunghezza. È un altro esempio di condizione di guardia, vista nel Paragrafo 6.8.

`i` e `j` sono indici: `i` attraversa `parola1` in avanti, mentre `j` attraversa `parola2` a ritroso. Se troviamo due lettere che non coincidono, possiamo restituire subito `False` Se continuiamo per tutto il ciclo e tutte le lettere coincidono, il valore di ritorno è `True`.

Se proviamo la funzione con i valori "pots" e "stop", ci aspetteremmo di ricevere di ritorno `True`, invece risulta un IndexError:

```
>>> al_contrario('pots', 'stop')
...
  File "reverse.py", line 15, in al_contrario
    if parola1[i] != parola2[j]:
IndexError: string index out of range
```

Per fare il debug, la mia prima mossa è di stampare il valore degli indici appena prima della riga dove è comparso l'errore.

```
    while j > 0:
        print(i, j)       # stampare qui

        if parola1[i] != parola2[j]:
            return False
        i = i+1
        j = j-1
```

Ora, eseguendo di nuovo il programma, ho qualche informazione in più:

```
>>> al_contrario('pots', 'stop')
0 4
...
IndexError: string index out of range
```

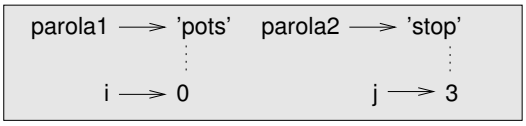

Figura 8.2: Diagramma di Stato.

Alla prima esecuzione del ciclo, il valore di `j` è 4, che è fuori intervallo della stringa `'pots'`. Infatti l'indice dell'ultimo carattere è 3, e il valore iniziale di `j` va corretto in `len(parola2)-1`.

Se correggo l'errore e rieseguo ancora il programma:

```
>>> al_contrario('pots', 'stop')
0 3
1 2
2 1
True
```

Stavolta il risultato è giusto, ma pare che il ciclo sia stato eseguito solo per tre volte, il che è sospetto. Per avere un'idea di cosa stia succedendo, è utile disegnare un diagramma di stato. Durante la prima iterazione, il frame di `al_contrario` è illustrato in Figura 8.2.

Mi sono preso la libertà di disporre le variabili nel frame e di aggiungere delle linee tratteggiate per evidenziare che i valori di `i` e `j` indicano i caratteri in `parola1` e `parola2`.

Partendo da questo diagramma, sviluppate il programma su carta cambiando i valori di `i` e `j` ad ogni iterazione. Trovate e correggete il secondo errore in questa funzione.

8.12 Glossario

oggetto: Qualcosa a cui una variabile può fare riferimento. Per ora, potete utilizzare "oggetto" e "valore" indifferentemente.

sequenza: Una raccolta ordinata di valori, in cui ciascun valore è identificato da un numero intero.

elemento: Uno dei valori di una sequenza.

indice: Un valore intero usato per selezionare un elemento di una sequenza, come un carattere in una stringa. In Python gli indici partono da 0.

slice: Porzione di una stringa identificata tramite un intervallo di indici.

stringa vuota: Una stringa priva di caratteri e di lunghezza 0, rappresentata da due apici o virgolette successivi.

immutabile: Detto di una sequenza i cui elementi non possono essere cambiati.

attraversare: Iterare attraverso gli elementi di una sequenza, effettuando su ciascuno un'operazione simile.

ricerca: Schema di attraversamento che si ferma quando trova ciò che si sta cercando.

contatore: Variabile utilizzata per contare qualcosa, solitamente inizializzata a zero e poi incrementata.

invocazione: Istruzione che chiama un metodo.

argomento opzionale: Un argomento di una funzione o di un metodo che non è obbligatorio.

8.13 Esercizi

Esercizio 8.1. *Leggete la documentazione dei metodi delle stringhe sul sito* `http://docs.python.org/3/library/stdtypes.html#string-methods`. *Fate degli esperimenti con alcuni metodi per assicurarvi di avere capito come funzionano.* `strip` *e* `replace` *sono particolarmente utili.*

La documentazione utilizza una sintassi che può risultare poco chiara. Per esempio, in `find(sub[, start[, end]])`*, le parentesi quadre indicano dei parametri opzionali (non vanno digitate). Quindi* `sub` *è obbligatorio, ma* `start` *è opzionale, e se indicate* `start`*, allora* `end` *è a sua volta opzionale.*

Esercizio 8.2. *Esiste un metodo delle stringhe di nome* `count` *che è simile alla funzione del Paragrafo 8.7. Leggete la documentazione del metodo e scrivete un'invocazione che conti il numero di* `a` *in* `'banana'`.

Esercizio 8.3. *Nello slicing, si può specificare un terzo indice che stabilisce lo* step *o "passo", cioè il numero di elementi da saltare tra un carattere estratto e il successivo. Uno step di 2 significa estrarre un carattere ogni 2 (uno sì, uno no), 3 significa uno ogni 3 (uno sì, due no), ecc.*

```
>>> frutto = 'banana'
>>> frutto[0:5:2]
'bnn'
```

Uno step di -1 fa scorrere all'indietro nella parola, per cui lo slicing `[::-1]` *genera una stringa scritta al contrario.*

Usate questo costrutto per scrivere una variante di una sola riga della funzione `palindromo` *dell'Esercizio 6.3.*

Esercizio 8.4. *Tutte le funzioni che seguono* dovrebbero *controllare se una stringa contiene almeno una lettera minuscola, ma qualcuna di esse è sbagliata. Per ogni funzione, descrivete cosa fa in realtà (supponendo che il parametro sia una stringa).*

```
def una_minuscola1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def una_minuscola2(s):
    for c in s:
        if 'c'.islower():
```

```
            return 'True'
        else:
            return 'False'

def una_minuscola3(s):
    for c in s:
        flag = c.islower()
    return flag

def una_minuscola4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def una_minuscola5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Esercizio 8.5. *Un cifrario di Cesare è un metodo di criptazione debole che consiste nel "ruotare" ogni lettera di una parola di un dato numero di posti seguendo la sequenza alfabetica, ricominciando da capo quando necessario. Ad esempio 'A' ruotata di 3 posti diventa 'D', 'Z' ruotata di 1 posto diventa 'A'.*

Per ruotare una parola, si ruota ciascuna delle sue lettere dello stesso numero di posti prefissato. Per esempio, "cheer" ruotata di 7 dà "jolly" e "melon" ruotata di -10 dà "cubed". Nel film 2001: Odissea nello Spazio*, il computer di bordo si chiama HAL, che non è altro che IBM ruotato di -1.*

Scrivete una funzione di nome `ruota_parola` *che richieda una stringa e un intero come parametri, e che restituisca una nuova stringa che contiene le lettere della stringa di partenza ruotate della quantità indicata.*

Potete usare le funzioni predefinite `ord`*, che converte un carattere in un codice numerico, e* `chr`*, che converte i codici numerici in caratteri. Le lettere sono codificate con il loro numero di ordine alfabetico, per esempio:*

```
>>> ord('c') - ord('a')
2
```

Dato che `'c'` *è la "2-esima" lettera dell'alfabeto. Ma attenzione: i codici numerici delle lettere maiuscole sono diversi.*

Su Internet, talvolta, vengono codificate in ROT13 (un cifrario di Cesare con rotazione 13) delle barzellette potenzialmente offensive. Se non siete suscettibili, cercatene qualcuna e decodificatela. Soluzione: `http://thinkpython2.com/code/rotate.py`.