Laurea in Ingegneria Gestionale

Corso di Fondamenti di Informatica
A.A. 2017/2018

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

liste

Le seguenti slide sono tratte principalmente dal lavoro di Giorgio Fumera (Univ. degli Studi di Cagliari) e riadattate da Raffele Nicolussi (Sapienza) e Domenico Lembo (Sapienza).

Tipi di dato strutturati

Tipi di dato nei linguaggi di alto livello

I valori che possono essere rappresentati ed elaborati nei programmi scritti in linguaggi di alto livello vengono classificati in diversi tipi di dato.

Per ***tipo di dato*** s'intende:

- un insieme di **valori**
- un insieme di **operazioni** eseguibili su tali valori

Abbiamo già incontrato alcuni tra i principali tipi di dato del linguaggio Python: numeri interi, numeri frazionari, stringhe, valori logici.

Tipi di dato Python: esempi

Il tipo di dato "numero intero" (associato al simbolo `int`) è definito come:

- l'insieme dei numeri interi che possono essere codificati in complemento a due con `n` bit (dove `n` dipende dall'architettura dello specifico calcolatore):
  `{-2n-1, . . . , +2n-1 - 1}`
- un insieme di operatori che include gli operatori aritmetici `+ - * / % **` e gli operatori di confronto `== != < <= => >`

Il tipo di dato "stringa" (`str`) è definito come:

- l'insieme delle sequenze di zero o più caratteri (senza un limite superiore predefinito sulla lunghezza), rappresentati tra apici singoli o doppi
- un insieme di operatori che include la concatenazione (`+`) e gli operatori di confronto `== != < <= => >`

Tipi semplici e strutturati

I tipi di dato vengono classificati a loro volta in:

- tipi semplici
- tipi strutturati

Un tipo **semplice** è costituito da valori che non possono essere "scomposti" in valori più semplici ai quali sia possibile accedere attraverso operatori o funzioni del linguaggio.
Esempi di tipi semplici del linguaggio Python (e di altri linguaggi) sono i **numeri interi, i numeri frazionari e i valori Booleani**.

Un tipo **strutturato** è invece costituito da valori che sono a loro volta collezioni o sequenze di valori più semplici.
Le stringhe sono un esempio di tipo strutturato: sono infatti composte da **sequenze ordinate** di caratteri a ciascuno dei quali è possibile accedere **individualmente**.

Tipi di dato struttuati di Python

Oltre alle stringhe, due dei principali tipi strutturati del linguaggio Python che vedremo in questo corso sono:

- le liste, che consentono di rappresentare **sequenze ordinate** di valori qualsiasi
- i dizionari, che consentono di rappresentare **collezioni** (non ordinate) di valori qualsiasi

Il tipo di dato *lista*

In molte applicazioni i dati da elaborare sono costituiti da **sequenze ordinate** di valori più semplici.

Per esempio, **le coordinate di un punto o gli elementi di un vettore in un dato sistema di riferimento possono essere rappresentati come una sequenza ordinata di numeri reali.**

Nei programmi è conveniente poter rappresentare dati di questa natura come singoli valori, per esempio associando le coordinate di un punto in uno spazio a tre dimensioni a una **singola** variabile invece che a tre variabili distinte.

Come si è già visto, nel caso particolare delle sequenze di caratteri questo è reso possibile dal tipo di dato *stringa*.

Il tipo *lista* fornisce questa possibilità per sequenze ordinate di valori che possono appartenente a tipi **qualsiasi**, anche diversi tra loro.

Il tipo di dato *lista*

Una lista si rappresenta nei programmi Python come:

- una sequenza ordinata di valori
- scritti tra parentesi **quadre**
- separati da virgole

Esempi:

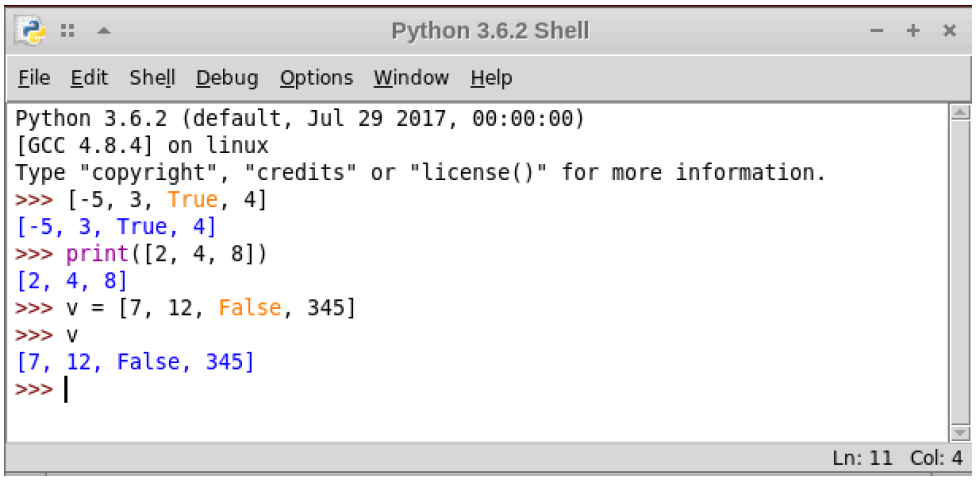
- `[7, -2, 4]`
  una lista composta da tre numeri interi
- `[-5.3, 6, True]`
  una lista composta da un numero reale, un numero intero e un valore logico
- `[]`
  una lista vuota

Il tipo di dato *lista*

Le liste, come i valori di qualsiasi tipo di dato Python, sono **espressioni**, e possono quindi essere scritte in qualsiasi punto di un programma nel quale possa comparire un'espressione.
È quindi possibile:

- scrivere una lista nella *shell* : dopo la pressione del tasto `INVIO`, la lista verrà stampata nella stessa *shell*, così come accade per i valori degl altri tipi (numeri, ecc.)
- stampare una lista con l'istruzione `print`, per esempio:
  `print ([7, -2, 4])`
- assegnare una lista a una variabile, per esempio:
  `v = [7, -2, 4]`

Esempi



Il tipo di dato *lista*

I valori degli elementi di una lista possono a loro volta essere indicati attraverso espressioni.

Per esempio, dopo l'esecuzione della seguente sequenza di istruzioni (scritte nella *shell* o in un *file*):

```
x = 2
y = -5
z = [x, y ** 2 + 1, x == 3]
```

la variabile `z` sarà associata alla lista `[2, 26, False]`

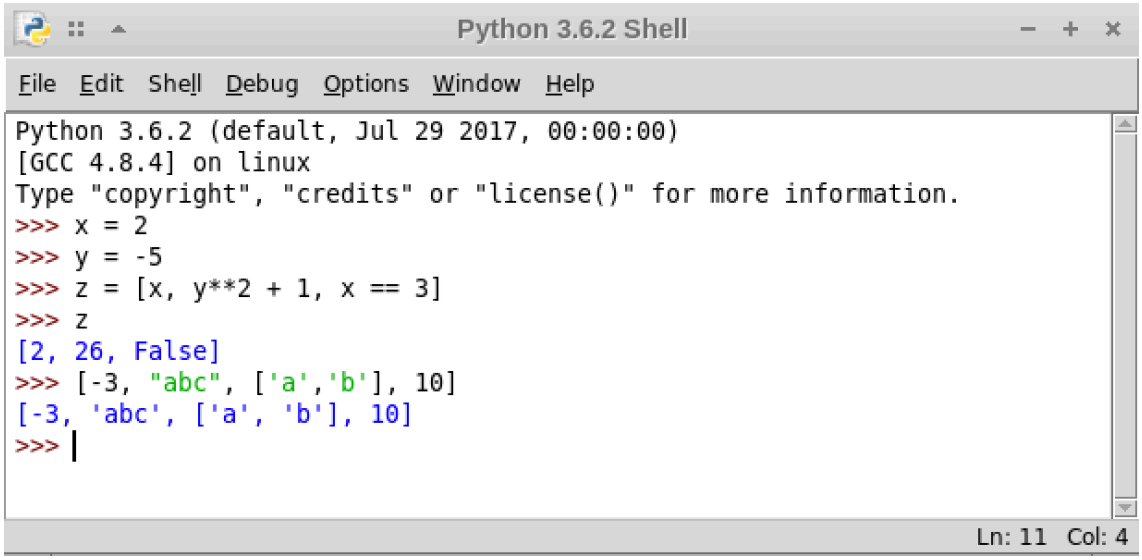
Il tipo di dato *lista*

Poiché gli elementi di una lista possono essere valori di un tipo qualsiasi, possono a loro volta essere liste: si parla in questo caso di liste **nidificate**, esattamente come nel caso di istruzioni composte (condizionale e iterativa) che contengano al loro interno altre istruzioni composte.

Per esempio, la seguente espressione produce come risultato una lista di **quattro** elementi: un numero intero, una stringa, una lista di due elementi, e un altro numero intero:

```
[-3, "abcd", ["a", 'b'], 10]
```


Esempi



Il tipo di dato *lista:* accesso ai singoli elementi

Come per gli altri tipi di dato, **anche per le liste il linguaggio Python mette a disposizione dei programmatori diversi operatori e funzioni predefinite.**

In particolare, essendo le liste un tipo strutturato, alcuni operatori consentono l'accesso ai **singoli** elementi di una lista.

Il meccanismo di accesso si basa sul fatto che ogni elemento è identificato univocamente dalla sua **posizione** all'interno di una lista (si ricordi che una lista è una sequenza **ordinata** di valori).

La posizione di ciascun elemento di una lista è rappresentata in linguaggio Python attraverso un numero intero, detto indice. Per convenzione, l'indice del primo elemento di una lista è 0, l'indice del secondo elemento è 1, e così via.

Il meccanismo è del tutto analogo a quanto abbiamo già visto per le stringhe.

Principali operatori sulle liste

La sintassi degli operatori principali è riassunta in questa tabella, ed è descritta di seguito in maggior dettaglio insieme alla semantica.
Si noti che ciascun operatore consente di scrivere una **espressione**.

sintassi	**descrizione**
$lista_1$ `==` $lista_2$	confronto ("uguale a")
$lista_1$ `!=` $lista_2$	confronto ("diverso da")
espressione `in` lista	verifica presenza di un valore in una lista
espressione `not in` lista	verifica assenza di un valore in una lista
$lista_1$ `+` $lista_2$	crea una nuova lista tramite attraverso la concatenazione degli elementi di $lista_1$ e $lista_2$
lista `*` num	crea una nuova lista replicando per num volte gli elementi di lista
lista `[indice]`	indicizzazione: accesso a un elemento
lista `[`$indice_1$`:`$indice_2$`]`	*slicing* (sezionamento): accesso a una sottosequenza di elementi

Operatori di confronto

Gli operatori `==` e `!=` consentono di scrivere espressioni **condizionali** (il cui valore sarà `True` o `False`) consistenti nel confronto tra due liste.

Sintassi:

- $lista_1$ `==` $lista_2$
- $lista_1$ `!=` $lista_2$

Dove $lista_1$ e $lista_2$ indicano **espressioni** che abbiano come valore una lista.

Semantica: due liste sono considerate identiche se sono composte dallo stesso numero di elementi, e se ogni elemento ha valore identico a quello dell'elemento che si trova nella **stessa posizione** nell'altra lista.

Gli operatori `in` e `not in`

Questi operatori consentono di scrivere espressioni **condizionali** che hanno lo scopo di verificare se un certo valore sia presente o meno all'interno di una lista.

Sintassi:

- espressione `in` lista
- espressione `not in` lista

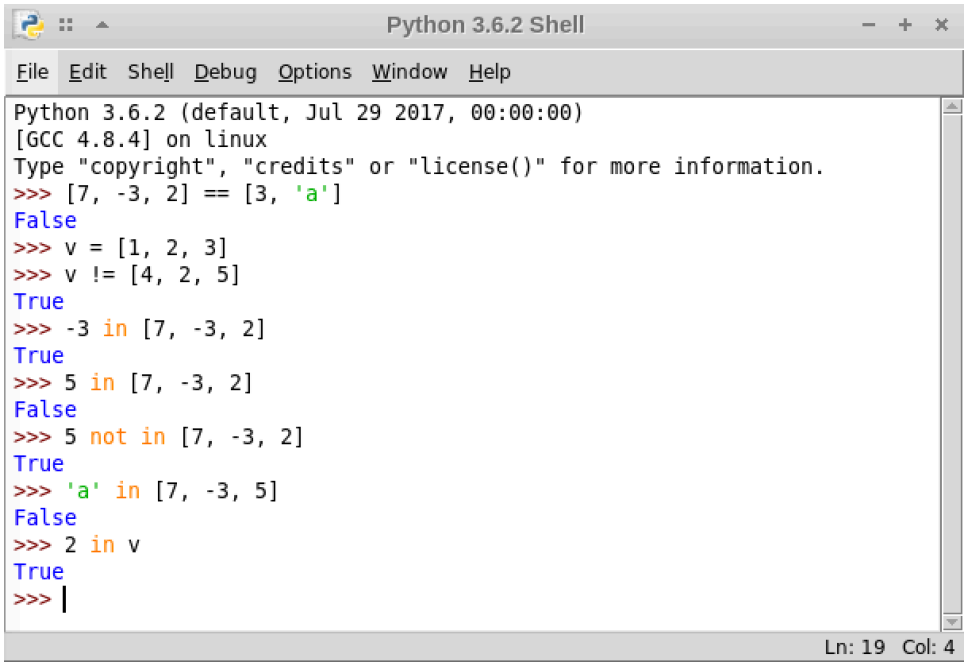
Dove espressione indica una qualsiasi espressione Python.

Semantica: se il **valore** di espressione è presente tra gli elementi di lista l'operatore `in` produce il valore `True`; in caso contrario produce `False`.

Il comportamento dell'operatore `not in` è quello opposto.

La ricerca **non** viene estesa agli elementi di eventuali liste nidificate all'interno di lista (nel senso che la lista nidificata è trattata come un elemento "atomico")

Esempi



L'operatore di concatenazione

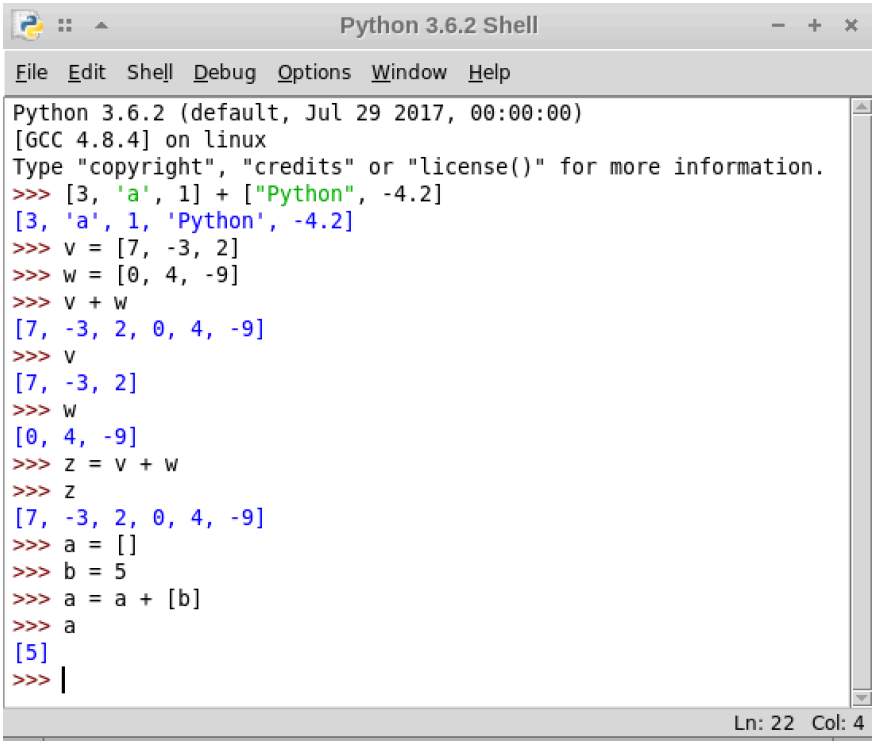
Questo operatore è analogo al corrispondente operatore del tipo di dato stringa.

Sintassi: $lista_1 + lista_2$

Semantica: la concatenazione produce una **nuova** lista composta dagli elementi di $lista_1$ seguiti da quelli di $lista_2$, disposti nello stesso ordine in cui si trovano nelle due liste.

Le liste originali **non** vengono modificate.

Esempi



L'operatore di indicizzazione

L'operatore di indicizzazione consente di accedere a ogni singolo elemento di una lista, per mezzo dell'indice corrispondente.

Sintassi: lista [indice]

Dove indice deve essere un'espressione il cui valore sia un **intero** compreso tra 0 e la lunghezza della lista **meno uno**.

Semantica: il risultato è il valore dell'elemento di lista il cui indice è pari al valore di indice. Se il valore di indice non corrisponde a una delle posizioni della lista si otterrà un messaggio d'errore.

L'operatore di indicizzazione

L'operatore di indicizzazione consente anche di **modificare** i singoli elementi di una lista, attraverso un'istruzione di assegnamento.

Sintassi: lista [indice] = espressione

- lista indica il nome di una variabile alla quale sia stata in precedenza assegnata una lista
- espressione indica una **qualsiasi** espressione Python

Semantica: l'elemento di lista nella posizione corrispondente a indice viene **sostituito** dal valore di espressione.

Esempi

Python 3.6.2 Shell

File Edit Shell Debug Options Window Help

```
Python 3.6.2 (default, Jul 29 2017, 00:00:00)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> v = [7, -3, 3, 0.4, -9]
>>> v[0]
7
>>> v[3]
0.4
>>> v[6]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    v[6]
IndexError: list index out of range
>>> v
[7, -3, 3, 0.4, -9]
>>> v[0] = -5
>>> v
[-5, -3, 3, 0.4, -9]
>>> v[3] = 10.5
>>> v
[-5, -3, 3, 10.5, -9]
>>> v[1.0]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    v[1.0]
TypeError: list indices must be integers or slices, not float
>>> v["0"]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    v["0"]
TypeError: list indices must be integers or slices, not str
>>>
```

Ln: 32 Col: 4

L'operatore di *slicing*

Questo operatore restituisce una lista composta da una sottosequenza di una data lista.

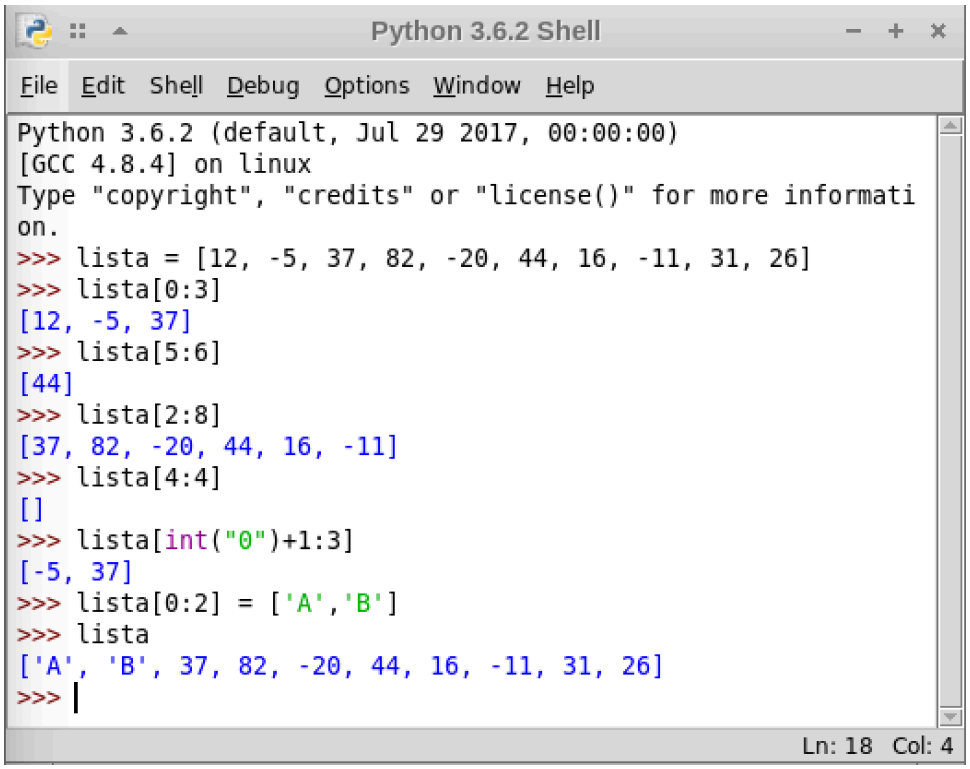
Sintassi: lista `[indice1:indice2]`

Dove indice $_1$ e indice $_2$ sono espressioni i cui valori devono essere numeri interi compresi tra 0 e la lunghezza di lista.

Semantica: il risultato è una lista composta dagli elementi di lista aventi indici da indice $_1$ a indice $_2$ − 1 (si noti che l'elemento di indice indice $_2$ **non** viene incluso nel risultato).

Anche in questo caso la lista originale **non** viene modificata.

Esempi



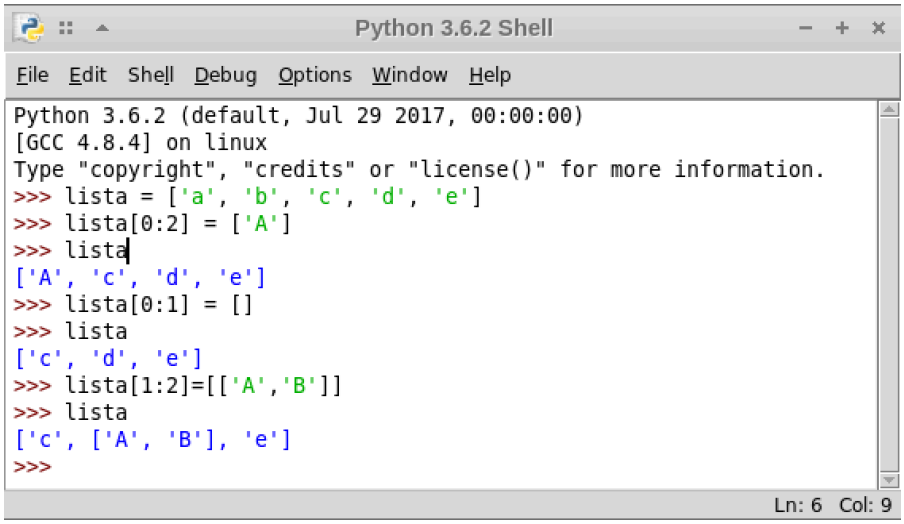
Nota

Nell'istruzione

```
lista[0:2] = ['A','B']
```

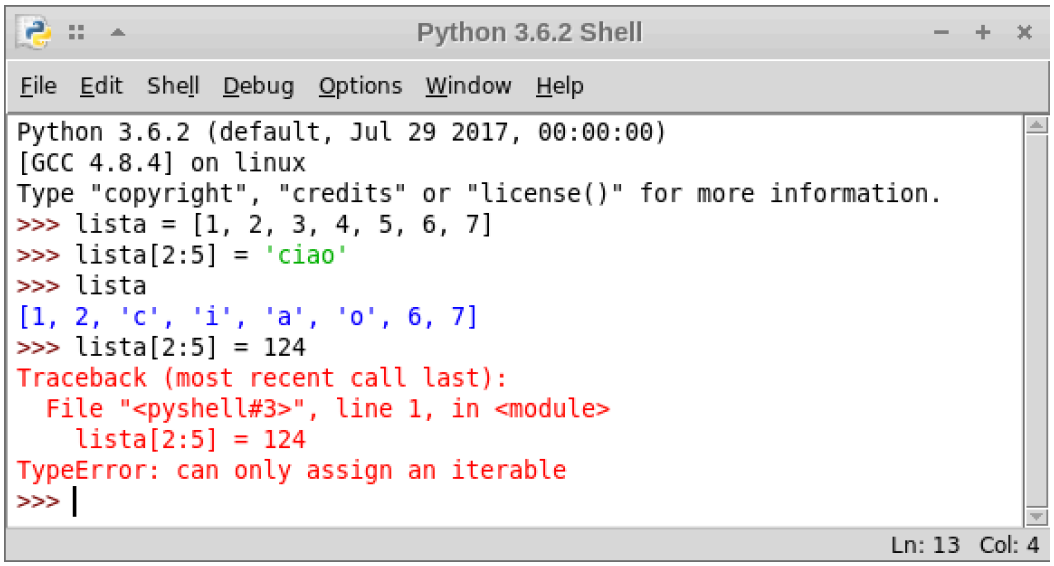
non è l'opertore di slicing ad aver modificato la variabile lista, ma è l'assegnazione (che riguarda solo i primi due elementi della lista).

Ulteriori esempi di assegnazioni:



Nota (cont)

Come già detto, l'operatore di slicing restituisce una sotto-lista della lista a cui è applicato, per cui in una assegnazione in cui lo slicing compare a sinistra del `=` si richiede che anche a destra sia presente una espressione che restituisce una lista. Essendo una stringa di fatto una lista di caratteri, questa espressione può anche essere una stringa.

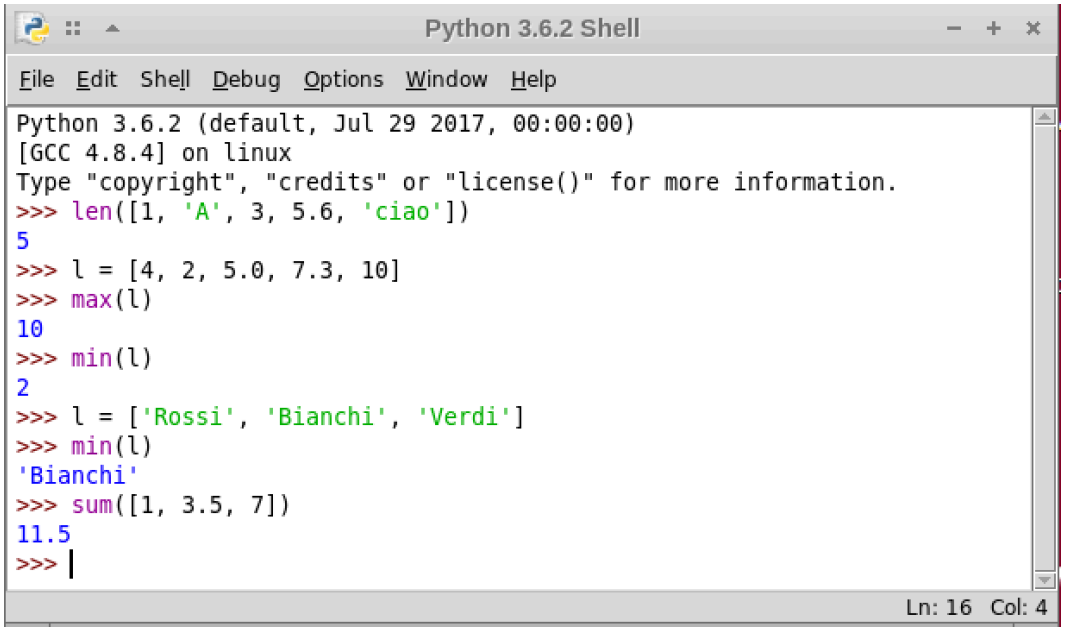


Funzioni predefinite che operano sulle liste

Le seguenti funzioni predefinite sono immediatamente disponibili nella *shell* e nei propri programmi:

nome	descrizione
`len(`lista`)`	restituisce la lunghezza di una lista
`min(`lista`)`	restituisce l'elemento più piccolo in una lista di elementi su cui è definito un ordinamento (solo numeri, solo stringhe, ecc.). Se l'ordinamento non è definito si ottiene un errore
`max(`lista`)`	restituisce l'elemento più grande in una lista di elementi su cui è definito un ordinamento. Se l'ordinamento non è definito si ottiene un errore
`sum(`lista`)`	data una lista di numeri (int o float) restituisce la somma degli elementi della lista (se la lista in input non è di numeri, si ottiene un errore)

Esempi



Esempio: costruzione di una lista

In molti casi pratici è necessario costruire una lista composta da valori che devono essere acquisiti durante l'esecuzione di un programma.
Questo si può ottenere attraverso l'operatore di concatenazione, a partire da una lista vuota.

Il programma seguente (disponibile nel *file* `costruzioneLista.py`) mostra un esempio di questo procedimento, con la costruzione di una lista di cinque numeri interi i cui valori vengono acquisiti attraverso la tastiera:

```
print ("Inserire cinque interi." )
lista = [ ]
k = 1
while k <= 5 :
    elemento = int(input("Prossimo valore: "))
    lista = lista + [elemento]
    k = k + 1
print ("La lista e':", lista)
```


Esempio: costruzione di una lista

Versione con l'uso del ciclo `for`

```
print ("Inserire cinque interi." )
lista = [ ]
for k in range(5) :
    elemento = int(input ("Prossimo valore: "))
    lista = lista + [elemento]
print ("La lista e':", lista)
```

Scansione di una lista

Si è detto in precedenza che nell'accesso agli elementi di una lista tramite l'operatore di indicizzazione, con l'espressione lista `[`indice`]`, il valore di indice può essere un'espressione qualsiasi. Questo implica che al suo interno possono comparire delle variabili.

Per esempio, se alle variabili `k` e `v` sono stati assegnati rispettivamente i valori `3` e `[4, -3, 6, 2, 9, -12]`, l'espressione `v[k]` produrrà il valore `2`, e l'espressione `v[k+1]` produrrà il valore `9`.

L'uso di variabili per rappresentare gli indici di una lista consente di esprimere in modo conciso, attraverso un'istruzione iterativa, la ripetizione di una stessa sequenza di istruzioni su tutti gli elementi di una lista, come mostrato di seguito.

Scansione di una lista

Lo schema seguente mostra come accedere in sequenza a tutti gli elementi di una lista, dal primo all'ultimo, per eseguire una stessa operazione su ciascuno di essi, usando una variabile nel ruolo di indice:

```
k = 0
while k < len(lista):
    istruzioni che coinvolgono lista[k]
    k = k + 1
```

Si noti l'uso della funzione `len`, che consente di applicare questo schema anche a liste delle quali non sia nota la lunghezza nel momento in cui si scrive il programma.

Notare anche la condizione `k < len(`lista`)` nell'istruzione `while`: il valore dell'indice dell'ultimo elemento è infatti pari alla lunghezza della lista **meno uno**.

Scansione di una lista

Come già fatto in precedenza, possiamo sostituire il ciclo `while` con un ciclo `for`, dato che il numero di iterazioni è noto all'inizio del ciclo:

```
for k in range(len(lista)):
    istruzioni che coinvolgono lista[k]
```

L'uso della funzione range consente alla variabile `k` di iterare su tutti gli elementi della lista, partendo da `0` fino a `len(lista)- 1`

Poiché una lista è essa stessa un contenitore di elementi ordinati, possiamo anche riscrivere il ciclo precedente in modo ancora più compatto, iterando direttamente sugli elementi della lista, invece che sugli indici. Questo però se non serve elaborare direttamente gli indici.

```
for elem in lista:
    istruzioni che coinvolgono elem
```

Nota: Come detto sopra, a volte è necessario iterare sull'indice e non sulla lista (ad esempio per acquisire gli elementi della lista come in `costruzioneLista.py`).

Confronto tra `while` e `for` tra per la scansione di una lista

Non è possibile usare l'istruzione `for`, con lo schema visto in precedenza, per eseguire operazioni sugli elementi di una sequenza che richiedono l'uso esplicito degli indici, come per esempio:

- la **modifica** di un elemento di una lista, attraverso un'istruzione di assegnamento `lista[k] = valore`

- l'accesso a **più di un elemento** di una lista (o di una stringa), per esempio per confrontare i valori di due elementi adiacenti con un'espressione condizionale come la seguente:
  `lista[k] != lista[k + 1]`

Scansione degli elementi di una lista: confronto tra iterazione su indici ed iterazione su elementi

Le istruzioni seguenti assegnano il valore 0 a tutti gli elementi di una lista precedentemente memorizzata in una variabile di nome `lista`:

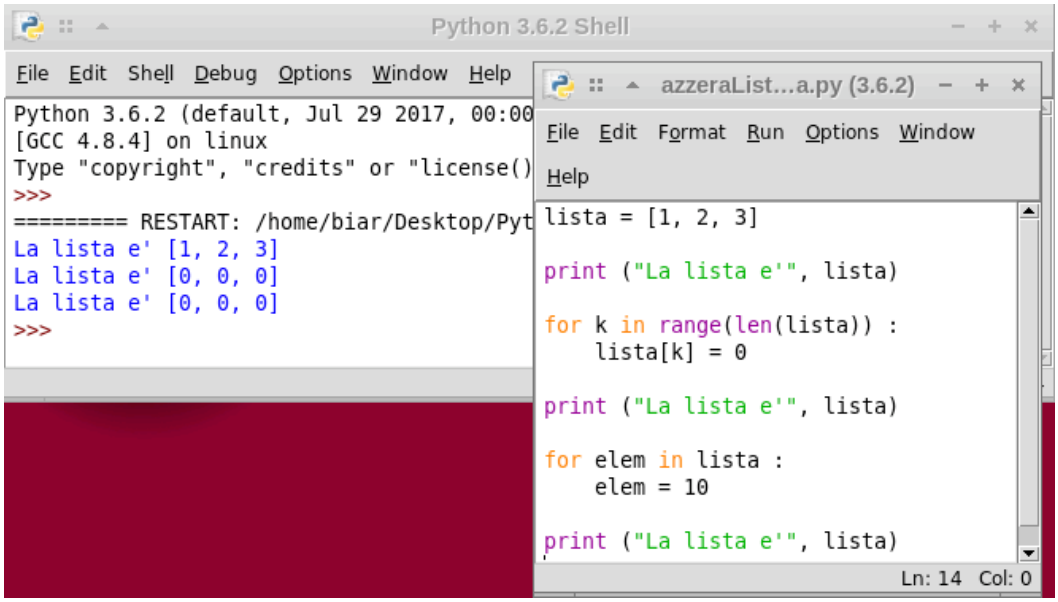
```
for k <= range(len(lista)) :
  lista[k] = 0
```

Se invece si usasse una sequenza di istruzioni come questa:

```
for elem in lista :
   elem = 0
```

la lista **non** verrebbe modificata, poiché ad una certa *iterazione i*, l'assegnazione `elem = 0` modifica solo il valore della variabile `elem` (cioè, dopo l'assegnazione, `elem` si riferisce ad una nuova locazione di memoria il cui contenuto è `0`), e non modifica il contenuto della locazione di memoria a cui si riferisce `lista[i-1]` (ricordiamoci che all'iterazione i, `elem` "seleziona" l'elemento in posizione i-1).

Esempio



Esempio

Il seguente programma (disponibile nel *file* `stampaLista.py`) stampa tutti gli elementi di una lista acquisita attraverso la tastiera, dal primo all'ultimo:

```
lista = input("Inserire una lista \
    (separando gli elementi con la ',' ): ").split(',')

print ("Gli elementi della lista sono:" )
for k in range(len(lista)) :
    print(lista[k])
```

Si noti l'uso del metodo `split` per suddividere la stringa in input *s* in sottostringhe per ogni `','` contenuta in *s*. Ad esempio, se l'utente inserisce da tastiera la sequenza `1,2,3`, la `input` restituisce al programma la stringa `'1,2,3'` ed il metodo `'1,2,3'.split(',')` restituisce la lista `['1', '2', '3']`

Nota: il codice presentato qui assume che la lista da leggere in input non contenga elementi di tipo stringa che contengono una virgola

Esempio

Versione alternativa che itera sugli elementi della lista:

```
lista = input("Inserire una lista \
    (separando gli elementi con la ',' ): ").split(',')

print ("Gli elementi della lista sono:" )
for elem in lista :
    print(elem)
```

Digressione: il metodo predefinito `split`

Il metodo `split` è molto utile per l'elaborazione di stringhe (per esempio, si veda più avanti il suo uso per l'elaborazione di dati acquisiti da *file* di testo).

La sintassi della chiamata è:

stringa`.split()`

Dove stringa indica una variabile avente per valore una stringa.

Questa funzione suddivide una stringa in corrispondenza dei caratteri di spaziatura (incluso il *newline*) e restituisce le corrispondenti sottostringhe all'interno di una **lista**, senza includere i caratteri di spaziatura (si noti che la stringa originale **non viene modificata**).

```
>>> s="ciao come va?"
>>> print(s.split())
['ciao', 'come', 'va?']
```

Digressione: il metodo predefinito `split`

Per esempio, se una stringa contiene una sequenza di parole separate da caratteri di spaziatura, `split` consente di "estrarre" le singole parole.

Un esempio: se la variabile `t` contiene la stringa:

```
"Questa e' una frase."
```

la chiamata:

```
t.split()
```

restituirà la lista:

```
["Questa", "e'", "una", "frase."]
```

Digressione: il metodo predefinito `split`

Se si desidera suddividere una stringa in corrispondenza di una sequenza di uno o più caratteri qualsiasi, tale sequenza dovrà essere indicata (sotto forma di **una** stringa) come argomento di `split`, con la seguente sintassi:

stringa`.split(`caratteri`)`

Per esempio, se la variabile `t` contenesse una stringa composta da una sequenza di numeri separati dal punto e virgola (senza spazi), come la seguente:

```
"15;1;25;9;6;21"
```

la chiamata:

```
t.split(';')
```

restituirebbe la lista:

```
["15", "1", "25", "9", "6", "21"]
```

(notare che i caratteri di separazione vengono eliminati).

Il metodo predefinito `split`: esempi

Dopo i seguenti assegnamenti:

```
a = "Una frase di cinque parole"
b = "PrimaRiga\nsecondaRiga."
c = "Uno...due...tre"
```

- `a.split()` restituisce `["Una", "frase", "di", "cinque", "parole"]`
- `b.split()` restituisce `["PrimaRiga", "secondaRiga."]`
- `c.split()` restituisce `["Uno...due...tre"]` (la stringa non viene suddivisa, poiché non contiene caratteri di spaziatura)
- `c.split("...")` restituisce `["Uno", "due", "tre"]`

Accedere agli elementi di una lista: esempio

Una semplice variante del programma `stampaLista.py` consente di accedere agli elementi di una lista in ordine **inverso**, dall'ultimo al primo:

```
lista = input("Inserire una lista \
    (separando gli elementi con la ',' ): ").split(',')
print ("Gli elementi, dall'ultimo al primo, sono:" )
for k in range(len(lista)-1,-1,-1) :
     print (lista[k])
```

Si noti che il valore iniziale della variabile `k` corrisponde all'indice dell'ultimo elemento, e che tale variabile viene **decrementata** di un'unità in ogni iterazione.

Accedere agli elementi di una lista: esempio

In verità Python consente anche l'uso di indici negativi, che consentono l'accesso alla lista in ordine inverso. In particolare, l'indice `-1` consente di accedere all'ultimo elemento della lista e l'indice `-len(ls)`, con `ls` variabile a cui è stata assegnata la lista, consente di accedere al primo elemento di `ls`:

```
lista = input("Inserire una lista \
    (separando gli elementi con la ',' ): ").split(',')
print ("Gli elementi, dall'ultimo al primo, sono:" )
for k in range(1,len(lista)+1) :
     print (lista[-k])
```

Esempio

Questo programma acquisisce una lista di numeri e ne **stampa i soli valori negativi** (si veda il *file* `stampaNegativiLista.py`):

```
lista = input("Inserire una lista di interi \
    (separando gli elementi con la ',' ): ").split(',')
print ("I valori negativi sono:")
for elem in lista :
    if int(elem) < 0 :
        print (elem)
```

Nota: il programma assume che l'utente inserisca solo liste contenente interi. L'istruzione `int(elem)` è necessaria in quanto nell'acquisizione della lista gli elementi sono rappresentati tramite stringhe.

Esempio

Questa funzione (disponibile nel *file* `sommaLista.py`) riceve come argomento una lista che si assume essere composta da numeri, e restituisce la loro **somma**.

```
def sommaLista (lista) :
    somma = 0
    for elem in lista :
        somma = somma + elem
    return somma
```

Si noti che questa funzione realizza la funzionalità già messa a disposizione dalla funzione predefinita `sum`

Esempio

Questa funzione (disponibile nel *file* `massimoLista.py`) riceve come argomento una lista che si assume essere composta da numeri, e restituisce il **maggiore** di tali numeri.

```
def massimoLista (lista) :
    massimo = lista[0]
    for elem in lista[1:len(lista)] :
        if elem > massimo :
            massimo = elem
    return massimo
```

Si noti che questa funzione realizza la funzionalità già messa a disposizione dalla funzione predefinita `max`

Esempio

La funzione riportata di seguito verifica se gli elementi di una lista di numeri ricevuta come argomento siano ordinati in senso **non decrescente**, restituento `True` in caso affermativo, `False` altrimenti.

Una sequenza è ordinata in senso non decrescente se e solo se **ciascuno** degli elementi dal primo al **penultimo** è minore o uguale a quello **successivo**. La funzione restituisce quindi `False` non appena tale condizione risulta falsa per una coppia di elementi adiacenti.

Notare che usando la variabile `i` per memorizzare l'indice di uno degli elementi della lista, in ogni iterazione il confronto tra una coppia di elementi adiacenti può essere eseguito considerando gli indici `i` e `i+1`. Per questo motivo il valore più grande che `i` deve assumere è pari all'indice del **penultimo** elemento della lista.

Esempio

La funzione è disponibile nel *file* `verificaOrdinamentoLista.py`

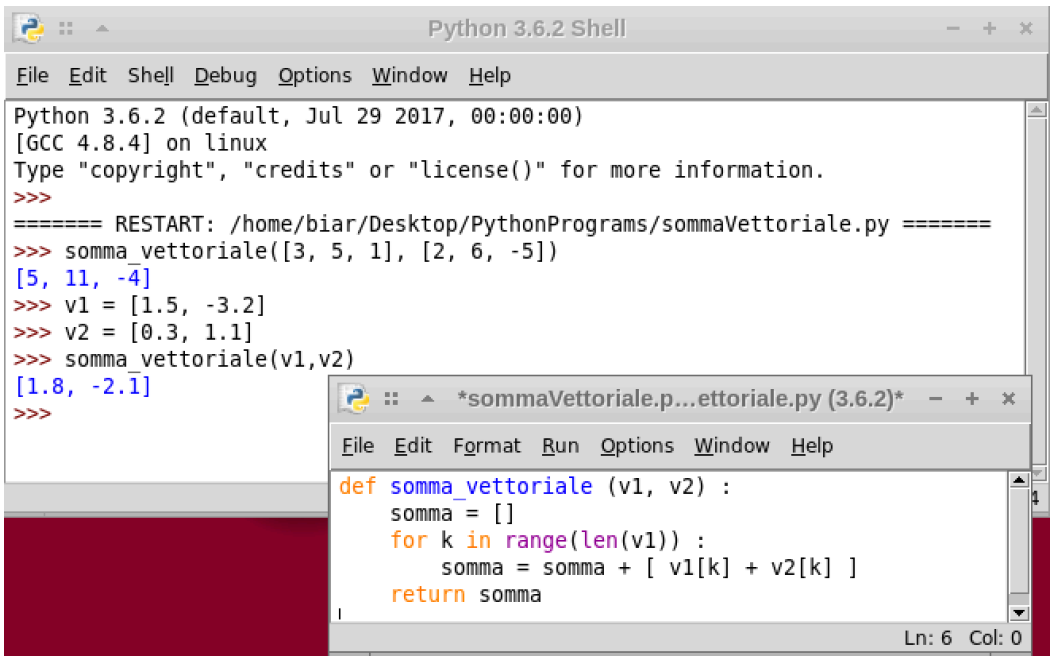
```
def ordinati (lista) :
    for i in range(len(lista)-1) :
        if lista[i] > lista[i+1] :
            return False
    return True
```


Esempio

Le liste possono essere usate per memorizzare gli elementi (coordinate) di vettori.
La funzione dell'esempio seguente (disponibile nel *file* `sommaVettoriale.py`) esegue la somma di due vettori (liste di numeri che si assume abbiano la stessa dimensione) ricevuti come argomenti, restituendo il risultato in una lista delle stesse dimensioni.
Si noti l'uso dell'operatore di concatenazione per costruire la lista che conterrà il risultato.

```
def somma_vettoriale (v1, v2) :
  somma = []
  for k in range(len(v1)) :
    somma = somma + [ v1[k] + v2[k] ]
return somma
```

Esempio



Stringhe e liste: *sequenze*

Le stringhe e le liste sono tipi di dato che hanno in comune il fatto di essere costituite da **sequenze ordinate** di un numero qualsiasi di elementi. Per questo motivo sono entrambe indicate in linguaggio Python con il termine più generale di sequenze.

Alcuni operatori e alcune funzioni predefinite che operano su sequenze ordinate possono essere applicati sia alle liste che alle stringhe (come si è già visto per l'operatore di concatenazione):

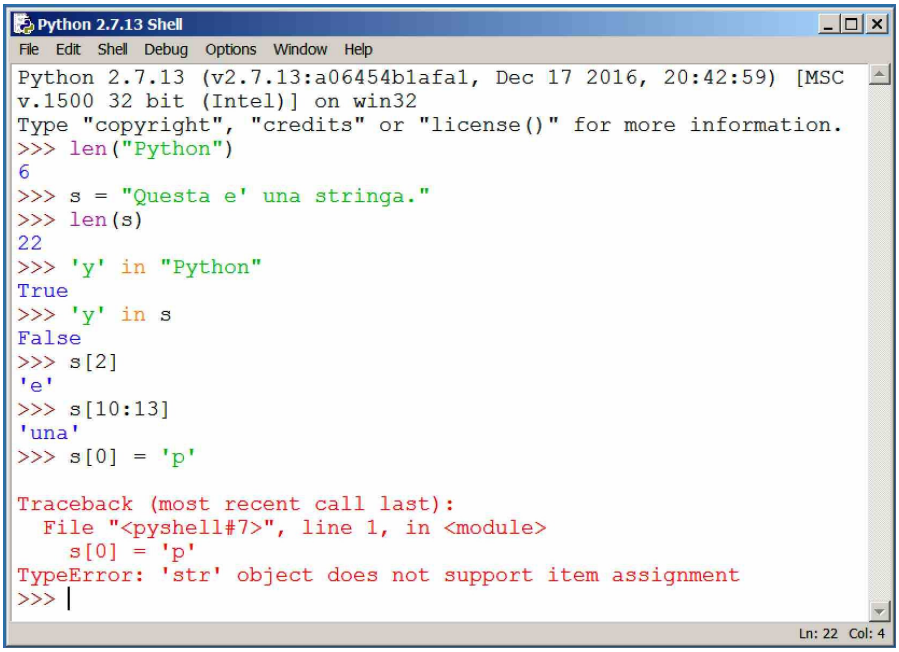
- funzioni predefinite: `len`
- operatori: `in` e `not in`, indicizzazione, *slicing*

Stringhe e liste: *sequenze*

Liste e stringhe presentano però una differenza fondamentale:

- attraverso l'istruzione di assegnamento e l'operatore di indicizzazione si è visto che è possibile **modificare** i singoli elementi di una lista: le liste sono perciò dette sequenze mutabili
- non è invece possibile modificare i singoli caratteri delle stringhe, che per questo sono dette immutabili (il tentativo di modificare un elemento di una stringa produce un errore)

Esempi



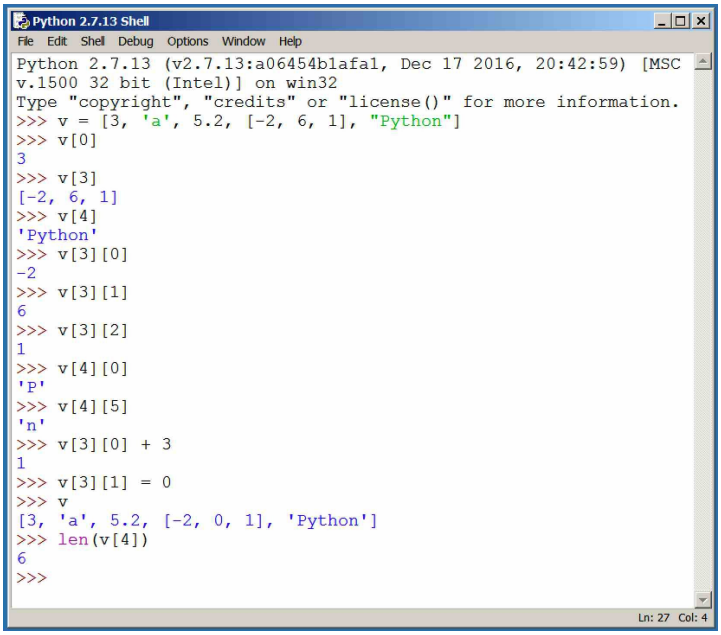
Accesso agli elementi di sequenze nidificate

Si è detto che gli elementi di una lista possono essere valori di tipi qualsiasi, quindi anche strutturati, come stringhe o altre liste.

L'operatore di indicizzazione consente di accedere anche agli elementi di strutture nidificate.

Se `s` è una variabile a cui è stata assegnata una lista, e l'elemento di indice *i* è a sua volta una sequenza (lista o stringa), sarà possibile accedere all'elemento di indice *j* di quest'ultima con la seguente **sintassi**: `s[i][j]`

Esempi



Liste nidificate: esempio

Si è visto che una delle possibili applicazioni delle liste è la rappresentazione di vettori.

Analogamente, le liste nidificate possono essere convenientemente usate per rappresentare **matrici**, cioè entità bidimensionali.

Liste nidificate: esempio

Una matrice di *m* righe e *n* colonne può essere rappresentata in un programma Python in diversi modi.

Una possibilità consiste nel memorizzare i suoi elementi, in un ordine opportuno, in una lista "semplice" (non nidificata) di $m \times n$ elementi.

Si consideri per esempio la seguente matrice:

$$\begin{pmatrix} -3 & 1 & 4 \\ 2 & 5 & -1 \end{pmatrix}$$

Scegliendo di procedere dall'alto verso il basso e da destra verso sinistra, tale matrice può essere rappresentata dalla lista:

```
[-3, 1, 4, 2, 5, -1]
```

Liste nidificate: esempio

Data una matrice di m righe e n colonne, non è difficile vedere che la scelta precedente associa l'elemento nella i-esima riga e nella j-esima colonna all'elemento di una lista nella posizione $(i - 1)n + j$, avente quindi indice $(i - 1)n + j - 1$.

Per esempio, se la matrice mostrata in precedenza ($m = 2$ righe e $n = 3$ colonne) fosse memorizzata in una lista assegnata a una variabile di nome `M`, l'elemento nella seconda riga ($i = 2$) e nella prima colonna ($j = 1$) corrisponderebbe all'elemento nella quarta posizione della lista: $(i - 1)n + j = 4$. Per accedere a tale elemento si dovrebbe quindi usare l'espressione `M[3]`.

Liste nidificate: esempio

È però possibile rappresentare una matrice con una lista nidificata, in modo da ottenere una corrispondenza più diretta e intuitiva tra gli indici di riga e colonna di un elemento della matrice e quelli dello stesso elemento della lista.

Questo risultato si ottiene considerando una matrice come una **sequenza ordinata** di m righe, ciascuna delle quali è a sua volta una **sequenza ordinata** di n valori semplici (numeri reali). Ogni riga può quindi essere rappresentata con una lista di n numeri, mentre l'intera matrice sarà rappresentata da una lista di m liste (righe della matrice).

In questo modo l'elemento della matrice nella i-esima riga e nella j-esima colonna corrisponderà all'elemento della lista avente indici $i - 1$ e $j - 1$.

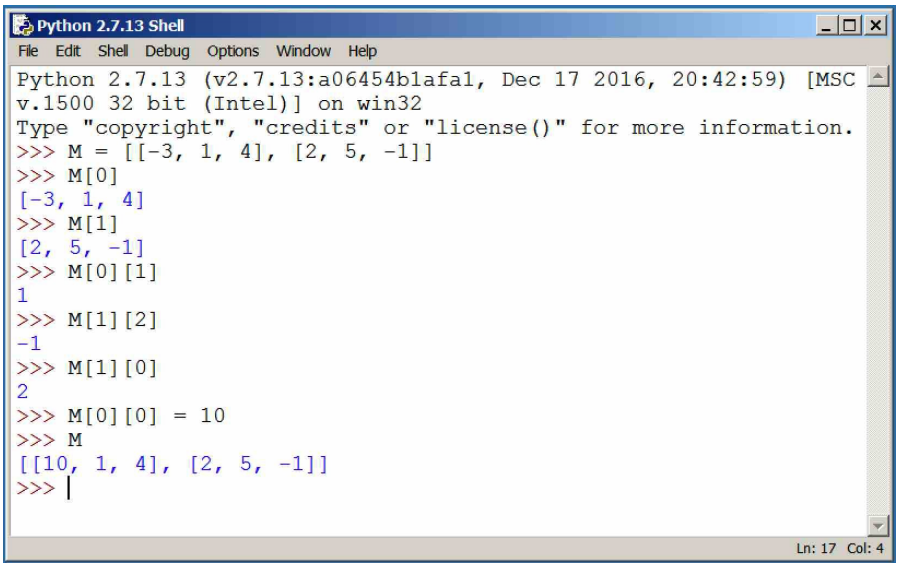
Liste nidificate: esempio

La matrice considerata in precedenza può essere rappresentata dalla seguente lista nidificata:

```
[ [-3, 1, 4], [2, 5, -1] ]
```

Assumendo come in precedenza che tale lista sia stata assegnata a una variabile di nome `M`, per accedere all'elemento nella seconda riga ($i = 2$) e nella prima colonna ($j = 1$) si userà l'espressione `M[1][0]`

Liste nidificate: esempio



Liste nidificate: esempio

Data una matrice memorizzata in una lista nidificata, per accedere ai suoi elementi secondo un certo ordine (per es., dalla riga in alto a quella in basso, e dalla colonna a sinistra a quella a destra) si dovranno usare due istruzioni iterative nidificate: la prima farà variare l'indice di riga, la seconda l'indice di colonna. Tali indici dovranno ovviamente essere memorizzati in due variabili distinte.

La funzione mostrata di seguito stampa gli elementi di una matrice, uno per riga, nell'ordine indicato sopra.

Liste nidificate: esempio

Questa funzione è disponibile nel *file* `stampaMatrice.py`

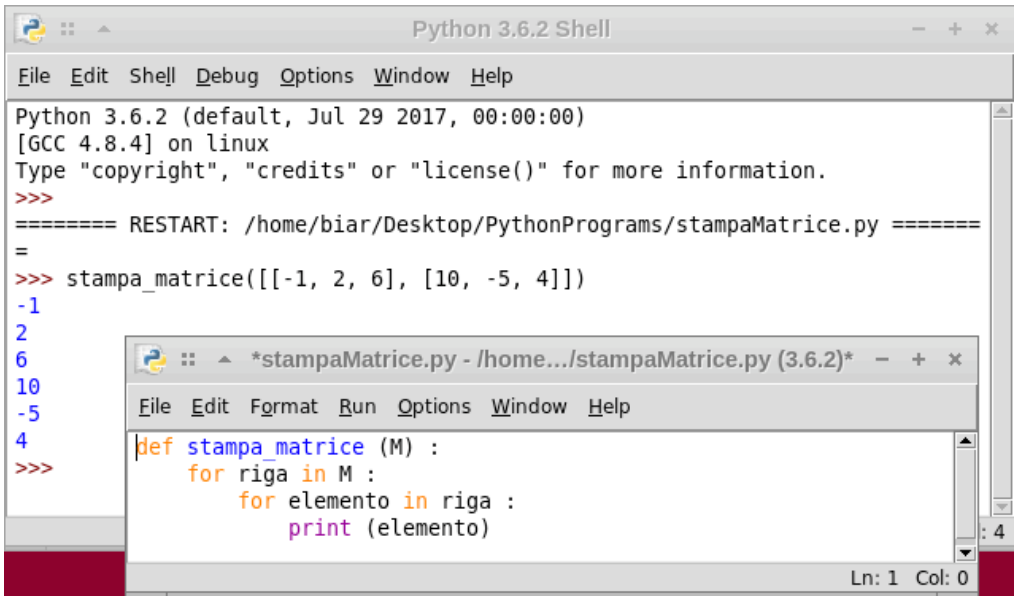
```
def stampa_matrice (M) :
    for riga in M :
        for elemento in riga :
            print(elemento)
```

Liste nidificate: esempio

Per completezza, si riporta in seugito anche la versione che fa uso del ciclo `while` (*file* `stampaMatriceWhile.py`)

```
def stampa_matrice (M) :
    indice_riga = 0
    while indice_riga < len(M) :
        indice_colonna = 0
        while indice_colonna < len(M[indice_riga]) :
            print(M[indice_riga][indice_colonna])
            indice_colonna = indice_colonna + 1
        indice_riga = indice_riga + 1
```

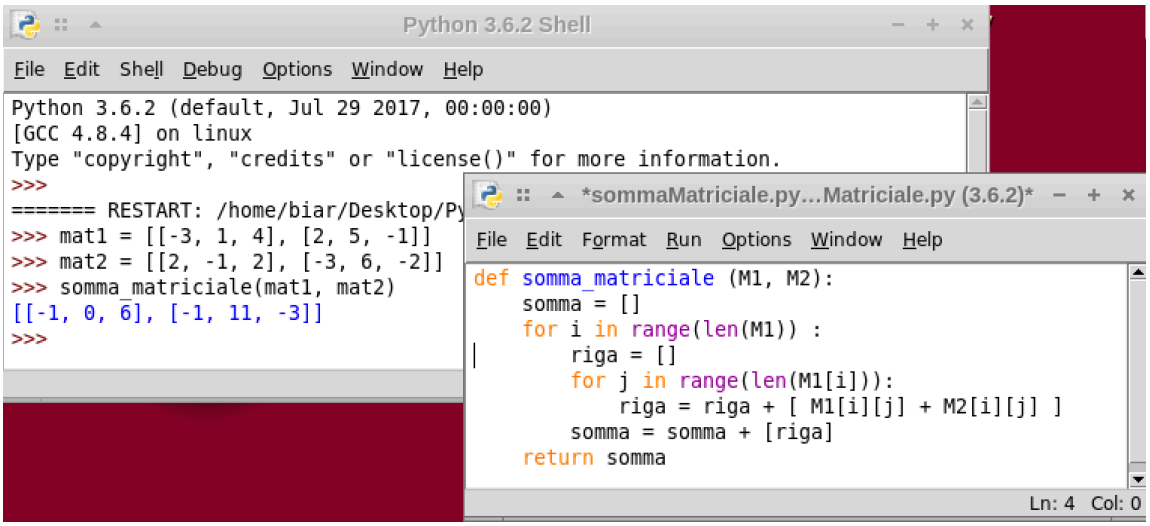

Liste nidificate: esempio



Liste nidificate: esempio

Questa funzione (disponibile nel *file* `sommaMatriciale.py`) restituisce la somma di due matrici (che si assumono avere le stesse dimensioni) ricevute come argomento. Come nel caso della somma vettoriale, la lista contenente il risultato viene costruita per concatenazione.

```
def somma_matriciale (M1, M2):
    somma = []
    for i in range(len(M1)) :
        riga = []
        for j in range(len(M1[i])):
            riga = riga + [ M1[i][j] + M2[i][j] ]
        somma = somma + [riga]
    return somma
```


Liste nidificate: esempio