Laboratorio 02

Programmazione - CdS Matematica

Mirko Polato

3 Novembre 2015

Contenitori

Spesso è necessario utilizzare *contenitori* di oggetti.

Contenitori

Spesso è necessario utilizzare *contenitori* di oggetti.

Per esempio:

- l'elenco degli studenti nella classe di Programmazione;
- i semi delle carte da gioco.

Contenitori

Spesso è necessario utilizzare *contenitori* di oggetti.

Per esempio:

- l'elenco degli studenti nella classe di Programmazione;
- i semi delle carte da gioco.

I contenitori sono quindi strutture dati che raggruppano altri oggetti.

Contenitori

Spesso è necessario utilizzare *contenitori* di oggetti.

Per esempio:

- l'elenco degli studenti nella classe di Programmazione;
- i semi delle carte da gioco.

I contenitori sono quindi strutture dati che raggruppano altri oggetti.

Conviene pensare ai contenitori come oggetti particolari che contengono al loro interno riferimenti ad altri oggetti.

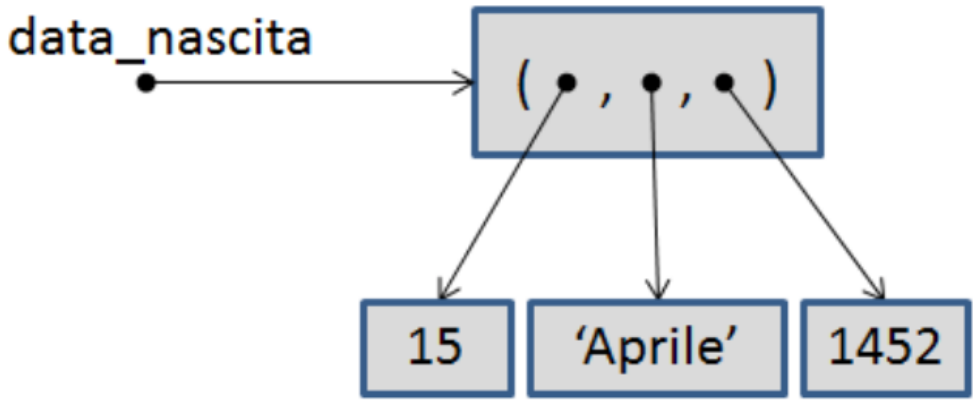
Tuple

Una tupla è una sequenza ordinata di oggetti **non sostituibili** (ovvero, la tulpa è **immutabile**) e non necessariamente omogenei.

Tuple

Una tupla è una sequenza ordinata di oggetti **non sostituibili** (ovvero, la tulpa è **immutabile**) e non necessariamente omogenei.

```
>>> data_nascita=15,"Aprile",1452
>>> # oppure:
>>> data_nascita=(15,"Aprile",1452)
```



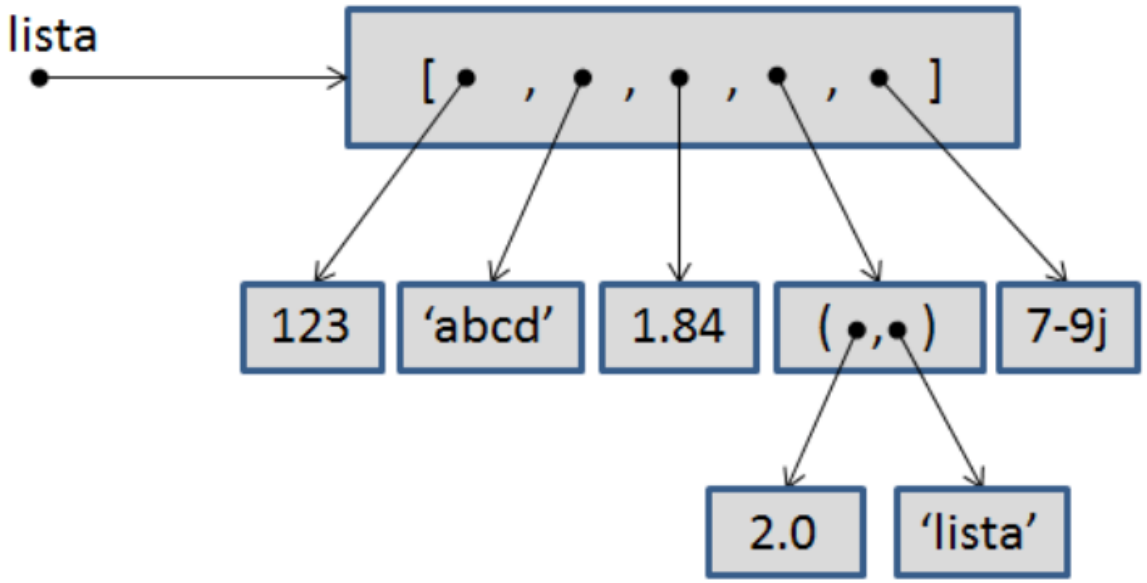
Liste

Una lista è una sequenza ordinata di oggetti non necessariamente omogenei ma, a differenza delle tuple, **mutabile**.

Liste

Una lista è una sequenza ordinata di oggetti non necessariamente omogenei ma, a differenza delle tuple, **mutabile**.

```
>>> lista=[123, "abcd", 1.84, (2.0,"lista"), 7-9j]
```



Esempio

Siano:

```
>>> v1 = 1
>>> v2 = 'a'
```

Può essere utile usare una tupla per scambiare i valori in modo rapido.

Esempio

Siano:

```
>>> v1 = 1
>>> v2 = 'a'
```

Può essere utile usare una tupla per scambiare i valori in modo rapido.

```
>>> v1,v2 = v2,v1
```

Esempio

Siano:

```
>>> v1 = 1
>>> v2 = 'a'
```

Può essere utile usare una tupla per scambiare i valori in modo rapido.

```
>>> v1,v2 = v2,v1
>>> v1
'a'
```

Esempio

Siano:

```
>>> v1 = 1
>>> v2 = 'a'
```

Può essere utile usare una tupla per scambiare i valori in modo rapido.

```
>>> v1,v2 = v2,v1
>>> v1
'a'
>>> v2
1
```

Operazioni sulle sequenze (1)

Per ogni oggetto sequenze (liste, tuple, stringhe) valgono le seguenti operazioni.

Operazioni sulle sequenze (1)

Per ogni oggetto sequenze (liste, tuple, stringhe) valgono le seguenti operazioni.
Consideriamo ad esempio l'oggetto `s = [1, 2, 3, 4, 5]`, possiamo:

Operazioni sulle sequenze (1)

Per ogni oggetto sequenze (liste, tuple, stringhe) valgono le seguenti operazioni.
Consideriamo ad esempio l'oggetto `s = [1, 2, 3, 4, 5]`, possiamo:

- accedere agli elementi della sequenza tramite l'operatore di indicizzazione `[]`,

```
>>> s[1]
2
```

Operazioni sulle sequenze (1)

Per ogni oggetto sequenze (liste, tuple, stringhe) valgono le seguenti operazioni.
Consideriamo ad esempio l'oggetto `s = [1, 2, 3, 4, 5]`, possiamo:

- accedere agli elementi della sequenza tramite l'operatore di indicizzazione `[]`,

```
>>> s[1]
2
```

- ottenere una sotto-sequenza attraverso l'operatore di slicing `[begin:end]` o striding `[begin:end:step]`,

```
>>> s[1:3]
[2, 3]
>>> s[1::2]
[2, 4]
```

Operazioni sulle sequenze (2)

(Ricordando, `s = [1, 2, 3, 4, 5]`)

- concatenare due sequenze ottenendo una nuova sequenza (+),

```
>>> s + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Operazioni sulle sequenze (2)

(Ricordando, `s = [1, 2, 3, 4, 5]`)

- concatenare due sequenze ottenendo una nuova sequenza (+),

```
>>> s + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

- generare una nuova sequenza in cui si ripetono *n* volte gli elementi di quella di partenza (*),

```
>>> s * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Operazioni sulle liste

- aggiungere elementi,

```
>>> s1 = ['a',1,('b',3+1j)] # lista di esempio
>>> s1.append('nuovo')      # aggiunge alla fine
>>> s1.extend([1,2])        # concatena alla fine
```

Operazioni sulle liste

- aggiungere elementi,

```
>>> s1 = ['a',1,('b',3+1j)] # lista di esempio
>>> s1.append('nuovo')      # aggiunge alla fine
>>> s1.extend([1,2])        # concatena alla fine
```

- invertire l'ordine degli elementi,

```
>>> s1.reverse()
```

Operazioni sulle liste

- aggiungere elementi,

```
>>> s1 = ['a',1,('b',3+1j)] # lista di esempio
>>> s1.append('nuovo')      # aggiunge alla fine
>>> s1.extend([1,2])        # concatena alla fine
```

- invertire l'ordine degli elementi,

```
>>> s1.reverse()
```

- inserire un elemento in un certo indice,

```
>>> s1.insert(1,'inserito')
```

Operazioni sulle liste

- aggiungere elementi,

```
>>> s1 = ['a',1,('b',3+1j)] # lista di esempio
>>> s1.append('nuovo')      # aggiunge alla fine
>>> s1.extend([1,2])        # concatena alla fine
```

- invertire l'ordine degli elementi,

```
>>> s1.reverse()
```

- inserire un elemento in un certo indice,

```
>>> s1.insert(1,'inserito')
```

- eliminare elementi, dato l'indice,

```
>>> del s1[5]
>>> s1
[2,'inserito',1,'nuovo',('b', (3+1j)),'a']
```

Esercizio

Ricordando che le stringhe sono sequenze, siano:

```
>>> s1 = "bu"
>>> s2 = "se"
>>> s3 = "te"
```

Usando `s1`, `s2` e `s3` comporre la stringa `"bubusettete"`.

Esercizio

Ricordando che le stringhe sono sequenze, siano:

```
>>> s1 = "bu"
>>> s2 = "se"
>>> s3 = "te"
```

Usando `s1`, `s2` e `s3` comporre la stringa `"bubusettete"`.

```
>>> s1*2 + s2 + s3[0] + s3*2
```

Esercizio

Ricordando che le stringhe sono sequenze, siano:

```
>>> s1 = "bu"
>>> s2 = "se"
>>> s3 = "te"
```

Usando `s1`, `s2` e `s3` comporre la stringa `"bubusettete"`.

```
>>> s1*2 + s2 + s3[0] + s3*2
'bubusettete'
```

Esercizio

A partire dalla stringa `s = "abecedario"`, e utilizzando le sue sotto-sequenze, creare una nuova stringa `s1 = "abe c'e' dario?"`.

Esercizio

A partire dalla stringa `s = "abecedario"`, e utilizzando le sue sotto-sequenze, creare una nuova stringa `s1 = "abe c'e' dario?"`.

```
>>> r = s[:3]+ " "+ s[3]+ "'"+ s[4]+ "' "+ s[5:]+ "?"
>>> r
"abe c'e' dario?"
```

Esercizio

Modificare la lista `a = ['c',('e','f'),['b','a']]` in modo da ottenere
`a = [['a','b'], 'c', 'd', ('e', 'f')]`
senza utilizzare altre variabili.

Esercizio

Modificare la lista `a = ['c',('e','f'),['b','a']]` in modo da ottenere
`a = [['a','b'], 'c', 'd', ('e', 'f')]`
senza utilizzare altre variabili.

```
>>> a = ['c',('e','f'), ['b', 'a']]
>>> a[0],a[2] = a[2],a[0]
>>> a[0].reverse()
>>> a[1],a[2] = a[2],a[1]
>>> a.insert(2,'d')
>>> a
[['a', 'b'], 'c', 'd', ('e', 'f')]
```

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato.

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
```

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
'I numeri 8 e 4 sono multipli di 2'
```

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
'I numeri 8 e 4 sono multipli di 2'
```

Ecco alcuni tipi utili:

- **Interi**: carattere (c), ottale (o), decimale (d), esadecimale (x, X);

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
'I numeri 8 e 4 sono multipli di 2'
```

Ecco alcuni tipi utili:

- **Interi**: carattere (c), ottale (o), decimale (d), esadecimale (x, X);
- **Float**: notazione esponenziale (e, E), fixed point (f);

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
'I numeri 8 e 4 sono multipli di 2'
```

Ecco alcuni tipi utili:

- **Interi**: carattere (c), ottale (o), decimale (d), esadecimale (x, X);
- **Float**: notazione esponenziale (e, E), fixed point (f);

```
>>> x = 0.123456789
>>> "Il valore di x troncato: %.1f  %.5f  %.9f" % (x
    ,x,x)
```

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
'I numeri 8 e 4 sono multipli di 2'
```

Ecco alcuni tipi utili:

- **Interi**: carattere (c), ottale (o), decimale (d), esadecimale (x, X);
- **Float**: notazione esponenziale (e, E), fixed point (f);

```
>>> x = 0.123456789
>>> "Il valore di x troncato: %.1f  %.5f  %.9f" % (x
    ,x,x)
'Il valore di x troncato: 0.1 0.12346 0.123456789'
```

Operatori di formato

Operatore di formato per le stringhe: un operatore per generare una stringa con un formato dato. Un primo esempio:

```
>>> "I numeri %d e %d sono multipli di %d"%(8,4,2)
'I numeri 8 e 4 sono multipli di 2'
```

Ecco alcuni tipi utili:

- **Interi**: carattere (c), ottale (o), decimale (d), esadecimale (x, X);
- **Float**: notazione esponenziale (e, E), fixed point (f);

```
>>> x = 0.123456789
>>> "Il valore di x troncato: %.1f  %.5f  %.9f" % (x
    ,x,x)
'Il valore di x troncato: 0.1 0.12346 0.123456789'
```

- **Stringa**: rappresentazione data da str (s).

Esempio

Dato `x = 42`, visualizzarlo usando gli operatori di formato: carattere, decimale, ottale e esadecimale.

Esempio

Dato `x = 42`, visualizzarlo usando gli operatori di formato: carattere, decimale, ottale e esadecimale.

```
>>> x = 42
>>> "%c %d %o %x" % (x,x,x,x)
```

Esempio

Dato `x = 42`, visualizzarlo usando gli operatori di formato: carattere, decimale, ottale e esadecimale.

```
>>> x = 42
>>> "%c %d %o %x" % (x,x,x,x)
'* 42 52 2a'
```

Esercizio

Importare il modulo `math` e mettere in `P` il valore di π (`math.pi`). Stampare il valore di `P` con 3, 6 e 10 decimali. (Come avviene il taglio dei decimali?)

Esercizio

Importare il modulo `math` e mettere in `P` il valore di π (`math.pi`). Stampare il valore di `P` con 3, 6 e 10 decimali. (Come avviene il taglio dei decimali?)

```
>>> import math
>>> P = math.pi
>>> P
3.141592653589793
```

Esercizio

Importare il modulo `math` e mettere in `P` il valore di π (`math.pi`). Stampare il valore di `P` con 3, 6 e 10 decimali. (Come avviene il taglio dei decimali?)

```
>>> import math
>>> P = math.pi
>>> P
3.141592653589793
>>> "Pigreco con 3 decimali: %.3f" % P
'Pigreco con due decimali: 3.142'
```

Esercizio

Importare il modulo `math` e mettere in `P` il valore di π (`math.pi`). Stampare il valore di `P` con 3, 6 e 10 decimali. (Come avviene il taglio dei decimali?)

```
>>> import math
>>> P = math.pi
>>> P
3.141592653589793
>>> "Pigreco con 3 decimali: %.3f" % P
'Pigreco con due decimali: 3.142'
>>> "Pigreco con 6 decimali: %.6f" % P
'Pigreco con due decimali: 3.141593'
```

Esercizio

Importare il modulo `math` e mettere in `P` il valore di π (`math.pi`). Stampare il valore di `P` con 3, 6 e 10 decimali. (Come avviene il taglio dei decimali?)

```
>>> import math
>>> P = math.pi
>>> P
3.141592653589793
>>> "Pigreco con 3 decimali: %.3f" % P
'Pigreco con due decimali: 3.142'
>>> "Pigreco con 6 decimali: %.6f" % P
'Pigreco con due decimali: 3.141593'
>>> "Pigreco con 10 decimali: %.10f" % P
'Pigreco con due decimali: 3.1415926536'
```

Esercizio

A partire dalla stringa `s = "abecedario"`, creare, utilizzando gli operatori di formato, la stringa `"abe c'e' dario?"`.

Esercizio

A partire dalla stringa `s = "abecedario"`, creare, utilizzando gli operatori di formato, la stringa `"abe c'e' dario?"`.

```
>>> r = "%s %c'%c' %s?" %(s[:3], s[3], s[4], s[5:])
>>> r
"abe c'e' dario?"
```

Range

Un utile comando per generare una lista di interi è il comando **range**: `range(stop)`, `range(start, stop)`, `range(start, stop, step)`.

Range

Un utile comando per generare una lista di interi è il comando **range**: `range(stop)`, `range(start, stop)`, `range(start, stop, step)`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Range

Un utile comando per generare una lista di interi è il comando **range**: `range(stop)`, `range(start, stop)`, `range(start, stop, step)`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(6,10)
[6, 7, 8, 9]
```

Range

Un utile comando per generare una lista di interi è il comando **range**: `range(stop)`, `range(start, stop)`, `range(start, stop, step)`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(6,10)
[6, 7, 8, 9]
>>> range(6,10,2)
[6, 8]
```

Range

Un utile comando per generare una lista di interi è il comando **range**: `range(stop)`, `range(start, stop)`, `range(start, stop, step)`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(6,10)
[6, 7, 8, 9]
>>> range(6,10,2)
[6, 8]
>>> range(10,-10,-2)
[10, 8, 6, 4, 2, 0, -2, -4, -6, -8]
```

Range

Un utile comando per generare una lista di interi è il comando **range**: `range(stop)`, `range(start, stop)`, `range(start, stop, step)`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(6,10)
[6, 7, 8, 9]
>>> range(6,10,2)
[6, 8]
>>> range(10,-10,-2)
[10, 8, 6, 4, 2, 0, -2, -4, -6, -8]
```

Nota: `stop` **non** è incluso!

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
>>> range(13,100,13)
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
>>> range(13,100,13)
[13, 26, 39, 52, 65, 78, 91]
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
>>> range(13,100,13)
[13, 26, 39, 52, 65, 78, 91]
>>> range(13,100,13)[1::2]
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
>>> range(13,100,13)
[13, 26, 39, 52, 65, 78, 91]
>>> range(13,100,13)[1::2]
[26, 52, 78]
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
>>> range(13,100,13)
[13, 26, 39, 52, 65, 78, 91]
>>> range(13,100,13)[1::2]
[26, 52, 78]
>>> range(187,100,-17)[1::2]
```

Esercizio

Generare le seguenti liste di numeri:

- multipli di 7 compresi tra 21 e 56 inclusi;
- pari multipli di 13 compresi tra 10 e 104 inclusi;
- dispari multipli di 17 compresi tra 100 e 200 inclusi in ordine decrescente.

```
>>> range(21,57,7)
[21, 28, 35, 42, 49, 56]
>>> range(13,100,13)
[13, 26, 39, 52, 65, 78, 91]
>>> range(13,100,13)[1::2]
[26, 52, 78]
>>> range(187,100,-17)[1::2]
[170, 136, 102]
```

Immutabili?

```
>>> h = ['uno', 'due']
>>> t = (h[0], h[1], h)
>>> t
```

Immutabili?

```
>>> h = ['uno', 'due']
>>> t = (h[0], h[1], h)
>>> t
('uno', 'due', ['uno', 'due'])
```

Immutabili?

```
>>> h = ['uno', 'due']
>>> t = (h[0], h[1], h)
>>> t
('uno', 'due', ['uno', 'due'])
>>> t[0] = 1
```

Immutabili?

```
>>> h = ['uno', 'due']
>>> t = (h[0], h[1], h)
>>> t
('uno', 'due', ['uno', 'due'])
>>> t[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment
```

Immutabili?

```
>>> h = ['uno', 'due']
>>> t = (h[0], h[1], h)
>>> t
('uno', 'due', ['uno', 'due'])
>>> t[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment
>>> h[0] = 3
>>> h[1] = 4
>>> t
```

Immutabili?

```
>>> h = ['uno', 'due']
>>> t = (h[0], h[1], h)
>>> t
('uno', 'due', ['uno', 'due'])
>>> t[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment
>>> h[0] = 3
>>> h[1] = 4
>>> t
('uno', 'due', [3, 4])
```

Esercizio

Dati:

```
>>> x = 1
>>> y = 0
>>> t = (x,x+1,x+3,[y,3])
```

Esercizio

Dati:

```
>>> x = 1
>>> y = 0
>>> t = (x,x+1,x+3,[y,3])
```

Si eseguano le seguenti istruzioni:

```
>>> x = x * 4
>>> t[3][1] = x
>>> t[3].extend([x,y])
>>> y = 2
>>> t[3][3] += 1
```

Esercizio

Dati:

```
>>> x = 1
>>> y = 0
>>> t = (x,x+1,x+3,[y,3])
```

Si eseguano le seguenti istruzioni:

```
>>> x = x * 4
>>> t[3][1] = x
>>> t[3].extend([x,y])
>>> y = 2
>>> t[3][3] += 1
```

Nel caso in cui l'esecuzione vada a buon fine, dire come è composta la tupla t.

Esercizio

Dati:

```
>>> x = 1
>>> y = 0
>>> t = (x,x+1,x+3,[y,3])
```

Si eseguano le seguenti istruzioni:

```
>>> x = x * 4
>>> t[3][1] = x
>>> t[3].extend([x,y])
>>> y = 2
>>> t[3][3] += 1
```

Nel caso in cui l'esecuzione vada a buon fine, dire come è composta la tupla t.

```
>>> t
```

Esercizio

Dati:

```
>>> x = 1
>>> y = 0
>>> t = (x,x+1,x+3,[y,3])
```

Si eseguano le seguenti istruzioni:

```
>>> x = x * 4
>>> t[3][1] = x
>>> t[3].extend([x,y])
>>> y = 2
>>> t[3][3] += 1
```

Nel caso in cui l'esecuzione vada a buon fine, dire come è composta la tupla t.

```
>>> t
(1, 2, 4, [0, 4, 4, 1])
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
>>> s.split() # separatori impliciti: " " e "\n"
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
>>> s.split() # separatori impliciti: " " e "\n"
['manca', 'poco', 'alla', 'fine', 'della', 'lezione', '
    molto', 'alla', 'fine', 'del', 'corso']
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
>>> s.split() # separatori impliciti: " " e "\n"
['manca', 'poco', 'alla', 'fine', 'della', 'lezione', '
    molto', 'alla', 'fine', 'del', 'corso']
>>> s.split("\n")
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
>>> s.split() # separatori impliciti: " " e "\n"
['manca', 'poco', 'alla', 'fine', 'della', 'lezione', '
    molto', 'alla', 'fine', 'del', 'corso']
>>> s.split("\n")
['manca poco alla fine della lezione', ' molto alla fine
     del corso']
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
>>> s.split() # separatori impliciti: " " e "\n"
['manca', 'poco', 'alla', 'fine', 'della', 'lezione', '
    molto', 'alla', 'fine', 'del', 'corso']
>>> s.split("\n")
['manca poco alla fine della lezione', ' molto alla fine
     del corso']
>>> s.split('m')
```

Liste e stringhe: split()

Date le stringhe `s` e `sep`, `s.split(sep)` ritorna la lista delle parole presenti in `s` usando `sep` come separatore.

```
>>> s = "manca poco alla fine della lezione\n molto alla
     fine del corso"
>>> s.split() # separatori impliciti: " " e "\n"
['manca', 'poco', 'alla', 'fine', 'della', 'lezione', '
    molto', 'alla', 'fine', 'del', 'corso']
>>> s.split("\n")
['manca poco alla fine della lezione', ' molto alla fine
     del corso']
>>> s.split('m')
['', 'anca poco alla fine della lezione\n ', 'olto alla
    fine del corso']
```

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

```
>>> "".join(['a','b','c']) # senza separatore
```

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

```
>>> "".join(['a','b','c']) # senza separatore
'abc'
```

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

```
>>> "".join(['a','b','c']) # senza separatore
'abc'
>>> " ".join(['a','b','c']) # separatore spazio
```

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

```
>>> "".join(['a','b','c']) # senza separatore
'abc'
>>> " ".join(['a','b','c']) # separatore spazio
'a b c'
```

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

```
>>> "".join(['a','b','c']) # senza separatore
'abc'
>>> " ".join(['a','b','c']) # separatore spazio
'a b c'
>>> "\n".join(['a','b','c']) # separatore termine riga
```

Liste e stringhe: join()

La funzionalità inversa è ottenuta con il metodo delle stringhe `join()` che unisce una lista di stringhe.

Anche se può sembrare controintuitivo, il metodo `join()` è un metodo della classe string e **va applicato alla stringa che contiene il separatore**.

```
>>> "".join(['a','b','c']) # senza separatore
'abc'
>>> " ".join(['a','b','c']) # separatore spazio
'a b c'
>>> "\n".join(['a','b','c']) # separatore termine riga
'a\nb\nc'
```

Esercizio

Dati:

```
>>> s = "long a what remember never I"
>>> t = ['sentence','I made','up']
```

Ottenere: "I * never * remember * what * a * long * sentence * I made * up".

Esercizio

Dati:

```
>>> s = "long a what remember never I"
>>> t = ['sentence','I made','up']
```

Ottenere: "I * never * remember * what * a * long * sentence * I made * up".

```
>>> k = s.split()
```

Esercizio

Dati:

```
>>> s = "long a what remember never I"
>>> t = ['sentence','I made','up']
```

Ottenere: "I * never * remember * what * a * long * sentence * I made * up".

```
>>> k = s.split()
>>> k.reverse()
```

Esercizio

Dati:

```
>>> s = "long a what remember never I"
>>> t = ['sentence','I made','up']
```

Ottenere: "I * never * remember * what * a * long * sentence * I made * up".

```
>>> k = s.split()
>>> k.reverse()
>>> s = k + t
```

Esercizio

Dati:

```
>>> s = "long a what remember never I"
>>> t = ['sentence','I made','up']
```

Ottenere: "I * never * remember * what * a * long * sentence * I made * up".

```
>>> k = s.split()
>>> k.reverse()
>>> s = k + t
>>> s = " * ".join(s)
```

Esercizio

Dati:

```
>>> s = "long a what remember never I"
>>> t = ['sentence','I made','up']
```

Ottenere: "I * never * remember * what * a * long * sentence * I made * up".

```
>>> k = s.split()
>>> k.reverse()
>>> s = k + t
>>> s = " * ".join(s)
>>> 'I * never * remember * what * a * long * sentence *
     I made * up'
```

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

```
>>> elle = [1,2,3,4,5]
>>> elle[2] = 6
>>> elle
```

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

```
>>> elle = [1,2,3,4,5]
>>> elle[2] = 6
>>> elle
[1,2,6,4,5]
```

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

```
>>> elle = [1,2,3,4,5]
>>> elle[2] = 6
>>> elle
[1,2,6,4,5]
>>> elle[1:3] = [7,8,9]
>>> elle
```

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

```
>>> elle = [1,2,3,4,5]
>>> elle[2] = 6
>>> elle
[1,2,6,4,5]
>>> elle[1:3] = [7,8,9]
>>> elle
[1,7,8,9,4,5]
```

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

```
>>> elle = [1,2,3,4,5]
>>> elle[2] = 6
>>> elle
[1,2,6,4,5]
>>> elle[1:3] = [7,8,9]
>>> elle
[1,7,8,9,4,5]
>>> elle[1:5:2] = [10,11]
>>> elle
```

Modifica di sotto-sequenze

Per oggetti sequenza di tipo mutabile è possibile anche utilizzare l'indicizzazione, lo slicing e lo striding per modificare delle sotto-sequenze.

```
>>> elle = [1,2,3,4,5]
>>> elle[2] = 6
>>> elle
[1,2,6,4,5]
>>> elle[1:3] = [7,8,9]
>>> elle
[1,7,8,9,4,5]
>>> elle[1:5:2] = [10,11]
>>> elle
[1,10,8,11,4,5]
```

Assegnazione o copia?

Quando assegnamo (un riferimento ad) una lista ad una variabile `list1`, `id(list1)` identifica la lista riferita da `list1`.
L'istruzione:

```
>>> list2 = list1
```

assegna a `list2` lo stesso riferimento di `list1` (`list1` e `list1` puntano alla stessa lista).

Assegnazione o copia?

Quando assegnamo (un riferimento ad) una lista ad una variabile `list1`, `id(list1)` identifica la lista riferita da `list1`.
L'istruzione:

```
>>> list2 = list1
```

assegna a `list2` lo stesso riferimento di `list1` (`list1` e `list1` puntano alla stessa lista).
Se questo non è il comportamento desiderato, possiamo clonare `list1` tramite l'operatore di slicing:

```
>>> list2 = list1[:]
```

Ora `list1` e `list2` si riferiscono a due oggetti list **diversi**.
Possiamo verificarlo confrontando `id(list1)` e `id(list2)`.

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
>>> list2.reverse()
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
>>> list2.reverse()
>>> list2[1].reverse()
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
>>> list2.reverse()
>>> list2[1].reverse()
>>> list1
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
>>> list2.reverse()
>>> list2[1].reverse()
>>> list1
[[7, 8, 9], [5, 5, 5], [1, 2, 3]]
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
>>> list2.reverse()
>>> list2[1].reverse()
>>> list1
[[7, 8, 9], [5, 5, 5], [1, 2, 3]]
>>> list2
```

Esercizio

Data la lista `list1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare `list2` a partire da `list1` in modo che

- `list1 = [[7,8,9],[5,5,5],[1,2,3]]`
- `list2 = [[1,2,3],[9,8,7]]`

```
>>> list1 = [[7,8,9],[5,5,5],[1,2,3]]
>>> list2 = [list1[0][:], list1[2][:]]
>>> list2.reverse()
>>> list2[1].reverse()
>>> list1
[[7, 8, 9], [5, 5, 5], [1, 2, 3]]
>>> list2
[[1, 2, 3], [9, 8, 7]]
```

Esercizio

Data la matrice

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

rappresentata come lista: `A=[[1,2,3],[4,5,6],[7,8,9]]`. Costruire `B` a partire da una copia profonda di `A` e modificare `B` per ottenere la trasposta di `A`, definita come:

$$B = \begin{bmatrix} A_{00} & A_{10} & A_{20} \\ A_{01} & A_{11} & A_{21} \\ A_{02} & A_{12} & A_{22} \end{bmatrix}$$

Esercizio

Data la matrice

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

rappresentata come lista: `A=[[1,2,3],[4,5,6],[7,8,9]]`. Costruire `B` a partire da una copia profonda di `A` e modificare `B` per ottenere la trasposta di `A`, definita come:

$$B = \begin{bmatrix} A_{00} & A_{10} & A_{20} \\ A_{01} & A_{11} & A_{21} \\ A_{02} & A_{12} & A_{22} \end{bmatrix}$$

```
>>> A = [[1,2,3],[4,5,6],[7,8,9]]
```

Esercizio

Data la matrice

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

rappresentata come lista: `A=[[1,2,3],[4,5,6],[7,8,9]]`. Costruire `B` a partire da una copia profonda di `A` e modificare `B` per ottenere la trasposta di `A`, definita come:

$$B = \begin{bmatrix} A_{00} & A_{10} & A_{20} \\ A_{01} & A_{11} & A_{21} \\ A_{02} & A_{12} & A_{22} \end{bmatrix}$$

```
>>> A = [[1,2,3],[4,5,6],[7,8,9]]
>>> B = [A[0][:],A[1][:],A[2][:]]
```

Esercizio

Data la matrice

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

rappresentata come lista: `A=[[1,2,3],[4,5,6],[7,8,9]]`. Costruire `B` a partire da una copia profonda di `A` e modificare `B` per ottenere la trasposta di `A`, definita come:

$$B = \begin{bmatrix} A_{00} & A_{10} & A_{20} \\ A_{01} & A_{11} & A_{21} \\ A_{02} & A_{12} & A_{22} \end{bmatrix}$$

```
>>> A = [[1,2,3],[4,5,6],[7,8,9]]
>>> B = [A[0][:],A[1][:],A[2][:]]
>>> B[0][1],B[1][0]=B[1][0],B[0][1]
```

Esercizio

Data la matrice

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

rappresentata come lista: `A=[[1,2,3],[4,5,6],[7,8,9]]`. Costruire `B` a partire da una copia profonda di `A` e modificare `B` per ottenere la trasposta di `A`, definita come:

$$B = \begin{bmatrix} A_{00} & A_{10} & A_{20} \\ A_{01} & A_{11} & A_{21} \\ A_{02} & A_{12} & A_{22} \end{bmatrix}$$

```
>>> A = [[1,2,3],[4,5,6],[7,8,9]]
>>> B = [A[0][:],A[1][:],A[2][:]]
>>> B[0][1],B[1][0]=B[1][0],B[0][1]
>>> B[0][2],B[2][0]=B[2][0],B[0][2]
```

Esercizio

Data la matrice

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

rappresentata come lista: `A=[[1,2,3],[4,5,6],[7,8,9]]`. Costruire `B` a partire da una copia profonda di `A` e modificare `B` per ottenere la trasposta di `A`, definita come:

$$B = \begin{bmatrix} A_{00} & A_{10} & A_{20} \\ A_{01} & A_{11} & A_{21} \\ A_{02} & A_{12} & A_{22} \end{bmatrix}$$

```
>>> A = [[1,2,3],[4,5,6],[7,8,9]]
>>> B = [A[0][:],A[1][:],A[2][:]]
>>> B[0][1],B[1][0]=B[1][0],B[0][1]
>>> B[0][2],B[2][0]=B[2][0],B[0][2]
>>> B[2][1],B[1][2]=B[1][2],B[2][1]
```