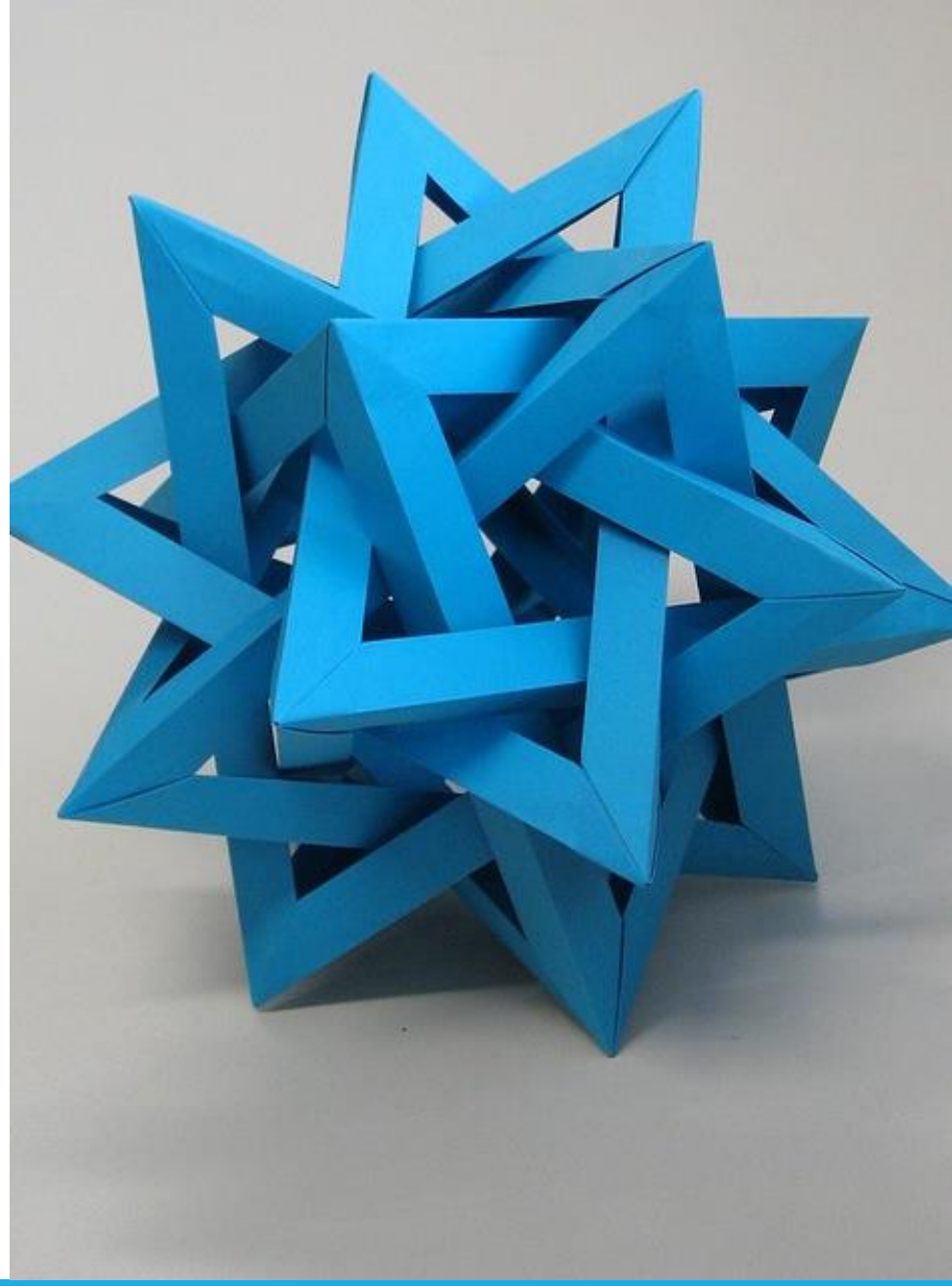
Politecnico di Torino

Dipartimento
di Automatica e Informatica

Unità P6: Liste e Tabelle

LISTE, OPERAZIONI CON LISTE, LISTE ANNIDATE

Capitolo 6



Obiettivi dell’unità

- Liste come contenitori di elementi
- Usare il ciclo for per scorrere la lista
- Imparare algoritmi comuni per lavorare con le liste
- Uso di liste con funzioni
- Lavorare con tabelle di dati

Contenuti

- Proprietà base delle liste
- Operazioni con le liste
- Algoritmi comuni con le liste
- Usare liste con funzioni
- Problem Solving: adattare algoritmi
- Problem Solving: scoprire algoritmi manipolando oggetti fisici
- Tabelle

Cos’è una lista?

- Una lista è una struttura dati versatile e dinamica, che contiene un numero variabile di elementi, di qualunque tipo, a cui si può avere accesso tramite la loro posizione (indice)
- Funzionalmente, è ciò che in altri linguaggi si potrebbe chiamare
  - Lista
  - Sequenza
  - Array
  - Vettore

Proprietà base delle liste

6.1

Creare una lista

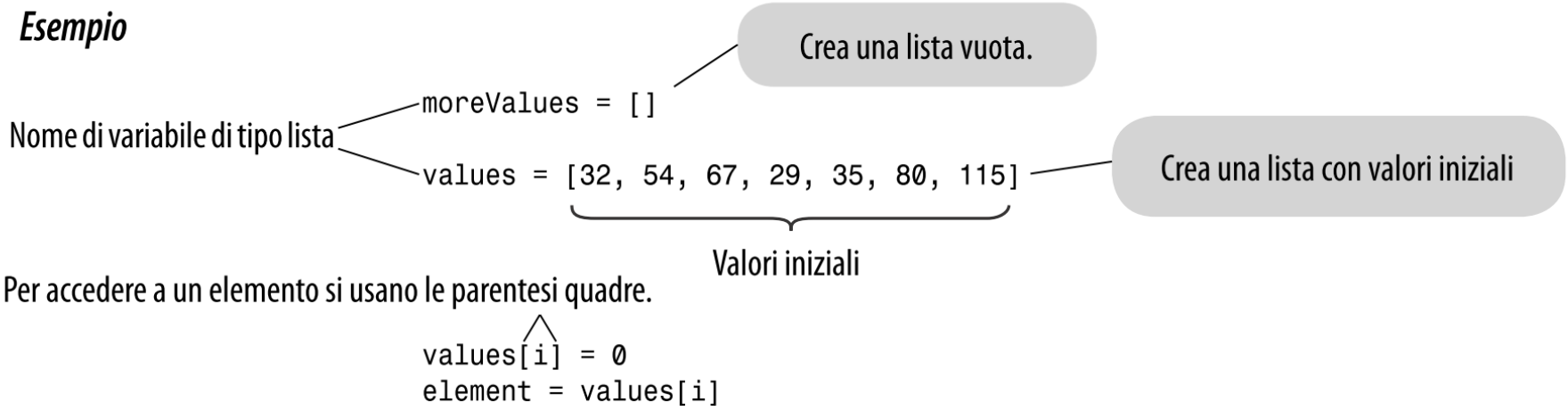
- Assegnare ad una variabile una nuova lista con l'operatore di indicizzazione **[ ]**

Sintassi

Per creare una lista: `[valore1, valore2, . . . ]`

Per accedere a un elemento: `riferimentoLista[indice]`

Esempio



Accesso alle liste di elementi

- Una lista è una sequenza di elementi, ognuno dei quali ha una posizione o indice che è un numero intero
- Per accedere ad un elemento della lista, specificare quale indice si vuole usare attraverso l'operatore di indicizzazione (allo stesso modo in cui si accede ad un singolo carattere in una stringa)
- Gli indici partono da `0`

Accesso ad un elemento della lista

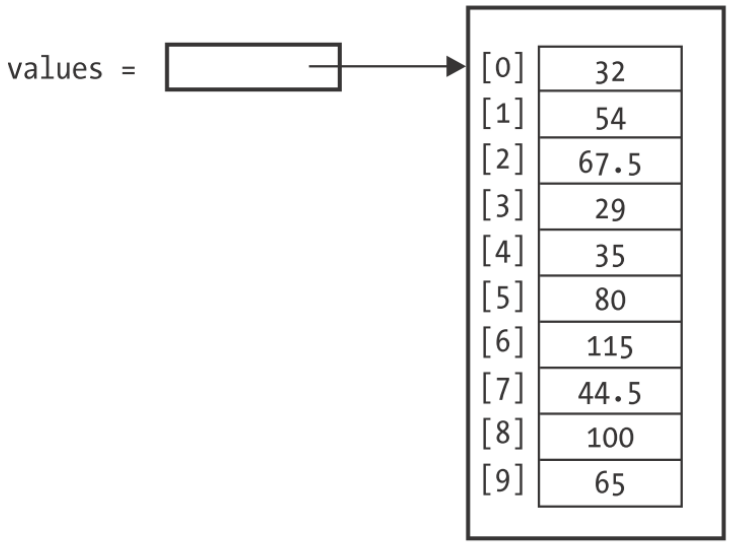
```
print(values[5])
```

Sostituzione di un elemento della lista

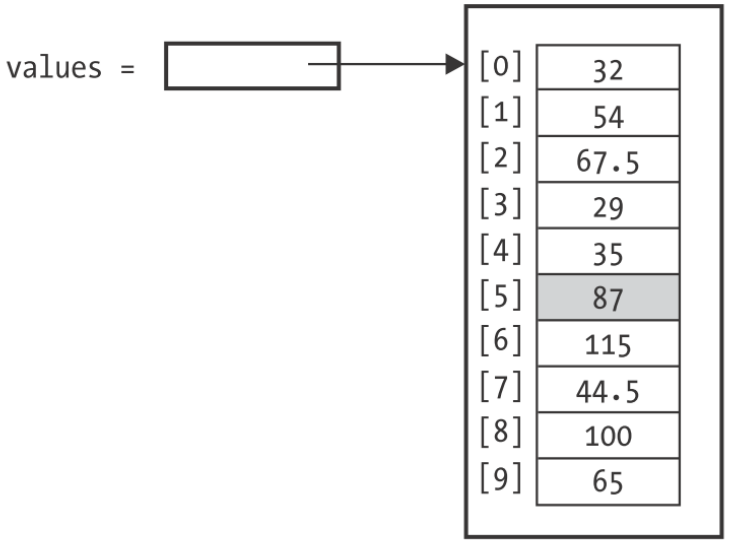
```
values[5] = 87
```

Creare liste / Accedere agli elementi

Accesso a un elemento della lista



② Sostituzione di un elemento della lista



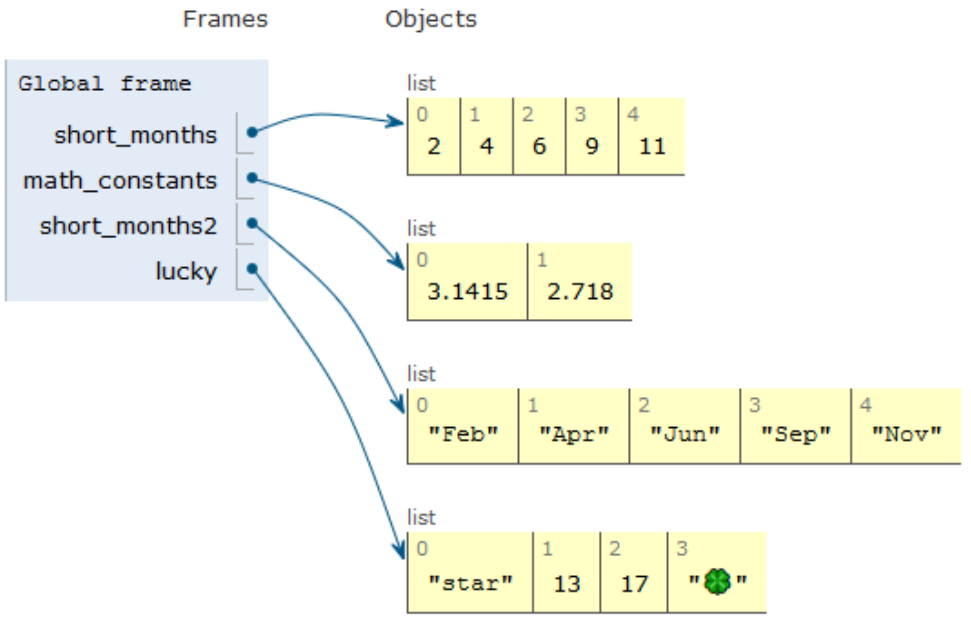

```
# 1: Creare una lista
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]

# 2: Accedere ad un elemento nella lista
values[5] = 87
```

Liste vs. Stringhe

- Sia le liste che le stringhe sono sequenze e l'operatore [ ] viene usato per accedere ad un elemento in qualsiasi sequenza
- Esistono due differenze fra liste e stringhe:
  - Le liste possono contenere valori di ogni tipo, invece le stringhe sono sequenze esclusivamente di caratteri
  - Le liste sono mutabili (il valore di ogni elemento può essere aggiornato, si possono aggiungere nuovi elementi o eliminare elementi esistenti), al contrario, le stringhe sono immutabili (non è possibile cambiare caratteri nella sequenza)

Tipi di elementi in una lista



- Lista di valori interi

  `short_months = [ 2, 4, 6, 9, 11 ]`

- Lista di valori reali

  `math_constants = [ 3.1415, 2.718 ]`

- Lista di stringhe

  `short_months2 = [ 'Feb', 'Apr', 'Jun', 'Sep', 'Nov' ]`

- Lista di valori misti
  - `lucky = [ 'star', 13, 17, '🍀' ]`
  - A meno che non ci sia una valida ragione, evita di mixare i tipi

Errori «Out-of-Range»

- Errori che si verificano quando si accede ad una lista fuori dall'intervallo consentito
- L'errore forse più comune trattando le liste è l'accesso ad un elemento che non esiste

```
values = [2.3, 4.5, 7.2, 1.0, 12.2, 9.0, 15.2, 0.5]
values[8] = 5.4
# Error -- values ha 8 elementi,
# quindi l'indice può variare solamente nel range 0-7
```

- Se il programma tenta di accedere ad una lista attraverso un indice fuori dal range consentito, il programma genererà un'eccezione durante l'esecuzione

Stampare una lista

- Una lista può essere passata come argomento alla funzione `print()`
- Tutti gli elementi sono stampati con una sintassi simile a quella della creazione della lista

```
>>> values = [ 1, 2, 3 ]
>>> print(values)
[1, 2, 3]
>>> print(values[0])
1
```

Determinare la lunghezza di una lista

- La funzione `len()` permette di ottenere la lunghezza della lista, ovvero il numero dei suoi elementi

```
num_elements = len(values)
```

Uso delle parentesi quadre

- Esistono due usi diversi delle parentesi quadre:
  - Quando seguono immediatamente il nome di una variabile svolgono la funzione di operatore di indicizzazione (identificazione di un elemento della lista)

```
values[4]
```

  - Quando seguono un "=" creano una nuova lista

```
values = [4]
```

Questa istruzione crea una lista con un elemento, l'intero 4

Cicli sui valori degli indici

- Data una lista chiamata "`values`" che contiene 10 elementi, si vuole accedere ogni elemento di essa
- Si potrebbe impostare una variabile detta `i` con valori 0, 1, 2, … fino a 9

```
# Prima versione (usa gli indici della lista)
for i in range(10) :
    print(i, values[i])
```

```
# Versione migliore (usa gli indici della
lista)
for i in range(len(values)) :
    print(i, values[i])
```

Questa istruzione crea sempre 10 iterazioni … ma se *values* avesse diversa lunghezza?

Questa istruzione funziona indipendentemente dalla lunghezza di *values* … controlla l'effettiva lunghezza al momento dell'esecuzione

Cicli sui valori degli elementi

- Data una lista chiamata "`values`" che contiene 10 elementi, si vuole accedere ogni elemento di essa
- Se non c'è il bisogno diretto dell'indice `i`, è possibile creare un ciclo direttamente sui valori

```
# Terza versione: (non usa gli indici)
# (attraversa gli elementi della lista)
for element in values :
    print(element)
```

Questa istruzione è perfetta se non c'è il bisogno della variabile indice

- Il `for` itera direttamente sulla lista (`values`)
- Ad ogni ciclo, `element` prende il valore di ogni elemento della lista

Cicli su indici e valori insieme

- Si può usare la funzione `enumerate` per estrarre da una lista tutte le coppie (indice, valore), ed iterare su di esse

```
# Quarta versione: (indici+valori)
# (attraversa gli elementi della lista)
for (i, element) in enumerate(values) :
    print(i, element)
```

La sintassi `(i, element)` costruisce una 'tupla' (v. più avanti) composta dai 2 valori.

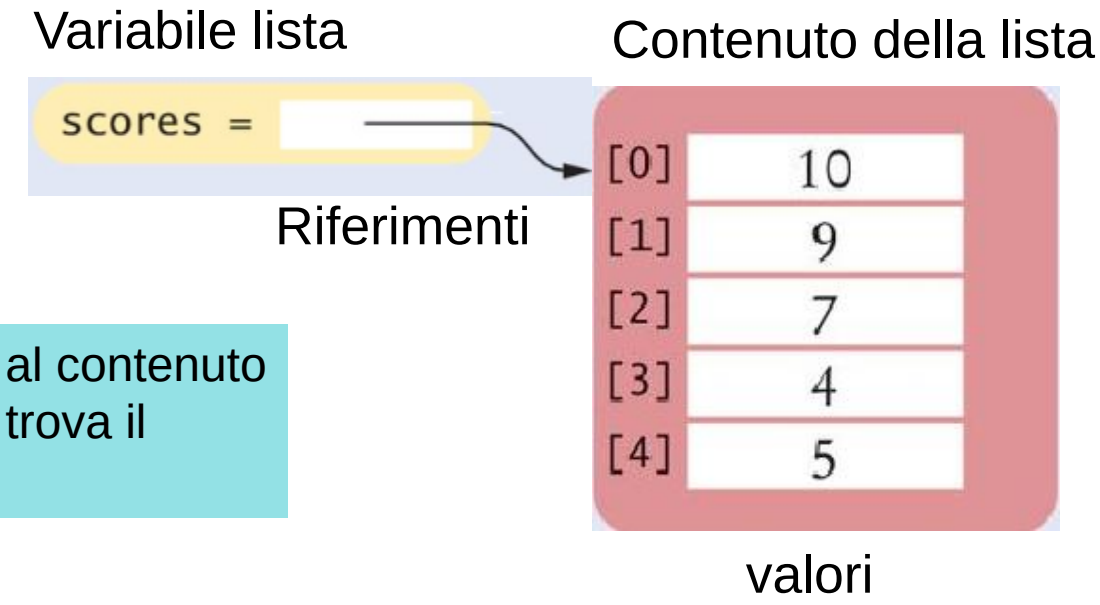
La funzione enumerate restituisce una lista di tuple, che viene quindi iterata.

- Il `for` itera sulle coppie (indice,valore)
- Ad ogni ciclo,
  - `i` prende il valore dell'indice di un elemento
  - `element` prende il valore di ogni elemento della lista

Riferimenti alla lista

- È importante ricordare la differenza fra:
  - Variabile: anche detta ‘puntatore’ o ‘riferimento’ alla lista
  - Contenuto della lista: memoria dove i dati sono contenuti

```
scores = [10, 9, 7, 4, 5]
```



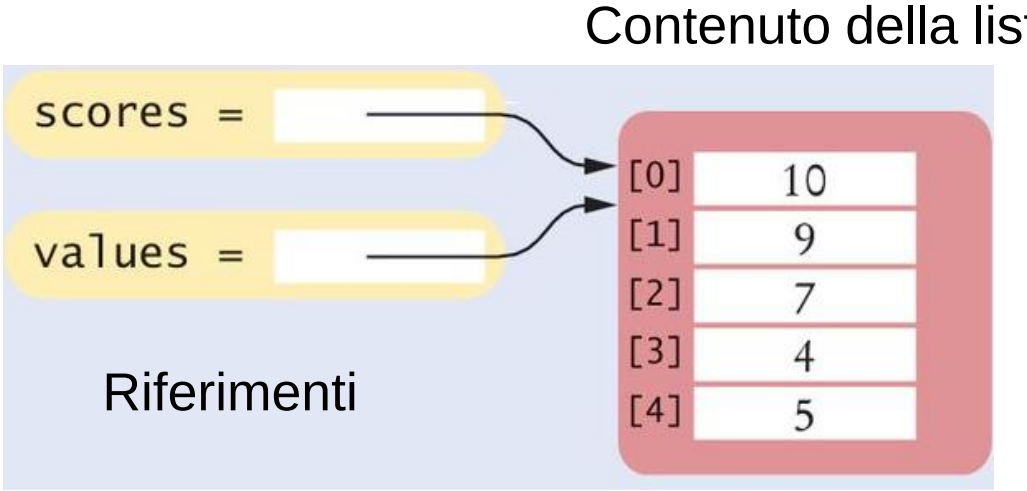
Una variabile contiene un **riferimento** al contenuto della lista, indica cioè il **luogo** in cui si trova il contenuto della lista (in memoria)

Alias di liste

- **Copiando** una variabile lista in un'altra, entrambe le variabili si riferiscono alla **stessa** lista
  - La seconda variabile è un *alias* per la prima perché entrambe le variabili si riferiscono ad una sola lista

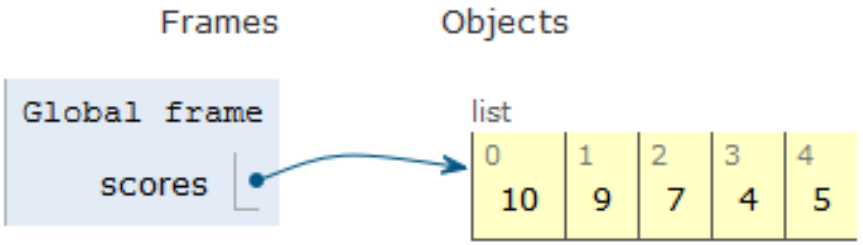
```
scores = [10, 9, 7, 4, 5]
values = scores # copia il riferimento alla lista
```

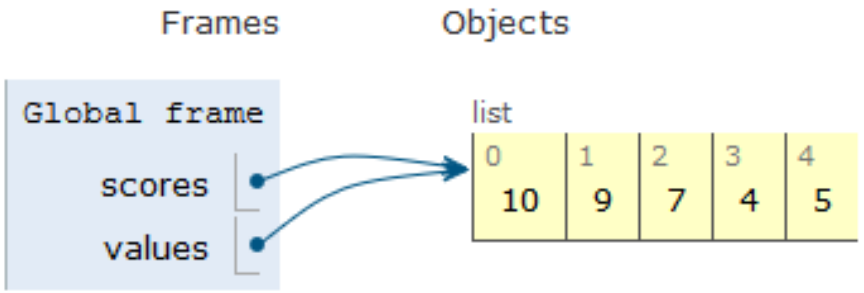
Una variabile lista specifica il **luogo** della lista. Copiando il riferimento si fornisce un secondo **riferimento** (**alias**) alla **stessa** lista.



Esercizio con PythonTutor

```
scores = [10, 9, 7, 4, 5]
values = scores   # copia la lista di riferimenti
```

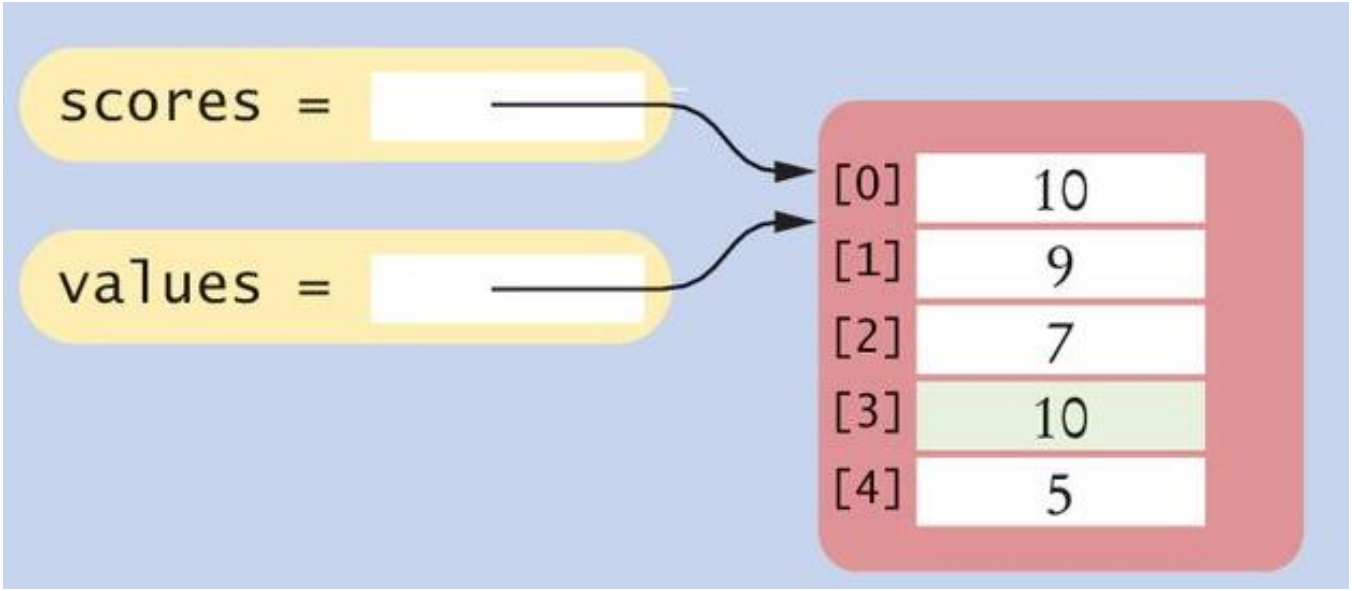




Modificare le liste in presenza di alias

- È possibile modificare le liste attraverso ciascuna delle variabili
- Infatti, si sta modificando la stessa lista accedendovi con due nomi diversi

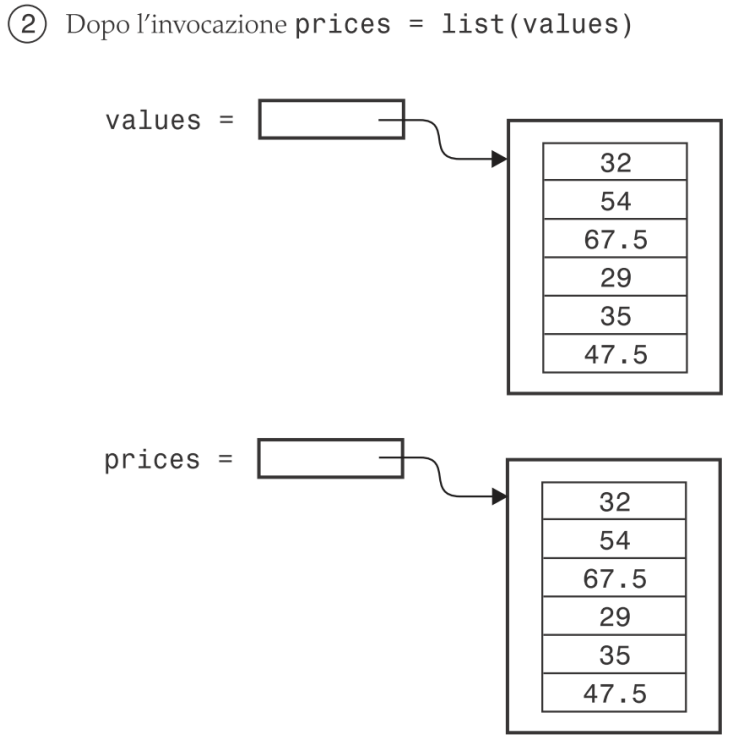
```
scores[3] = 10
print(values[3])   # Prints 10
```



Copiare liste

- Quando si vuole creare una copia della lista, ovvero una **nuova** lista che abbia gli stessi elementi nello stesso ordine rispetto ad una lista data
- Si usa la funzione `list()`:

```
prices = list(values)
```

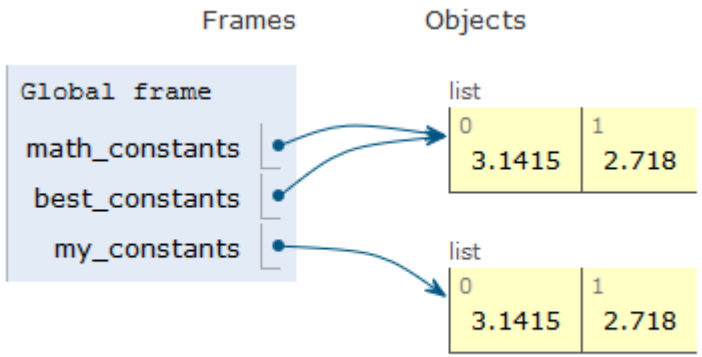


Esercizio con PythonTutor

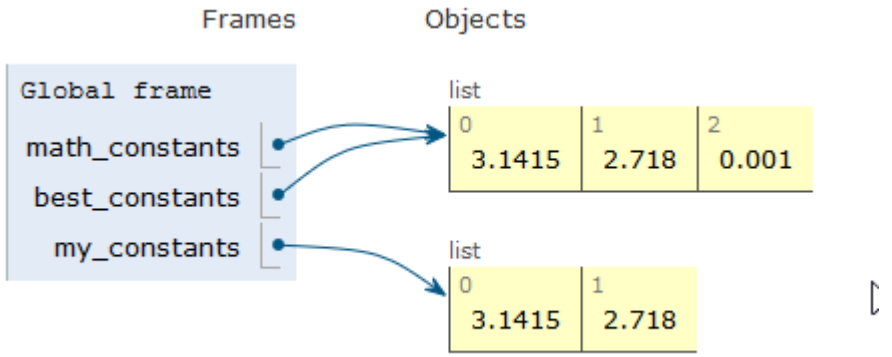
```
math_constants = [ 3.1415, 2.718 ]

best_constants = math_constants

my_constants =
list(math_constants)
```




```
math_constants.append(0.001)
```

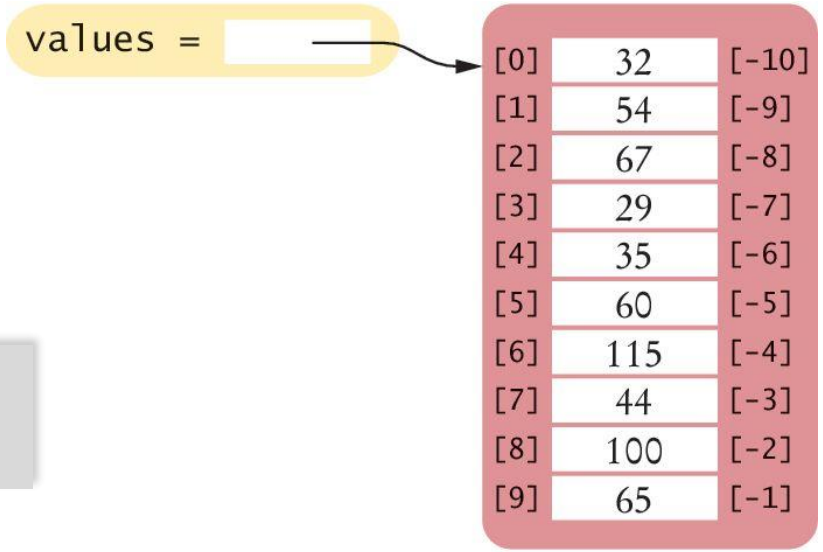


Indici negativi

- A differenza che in altri linguaggi, Python permette agli indici negativi di accedere alla lista di elementi in ordine contrario
  - Per esempio, un indice di -1 dà accesso all'ultimo elemento della lista
  - Allo stesso modo, `values[-2]` porta al penultimo elemento

```
last = values[-1]
print("l'ultimo elemento della
lista è", last)
```

```
last = values[len(values)-1]
# equivalente
```



Operazioni con le liste

6.2

Lista delle operazioni

- Aggiungere elementi
- Inserire elementi
- Trovare un elemento
- Rimuovere un elemento
- Concatenazione
- Test di uguaglianza/differenza
- Somma, massimo, minimo e ordinamento

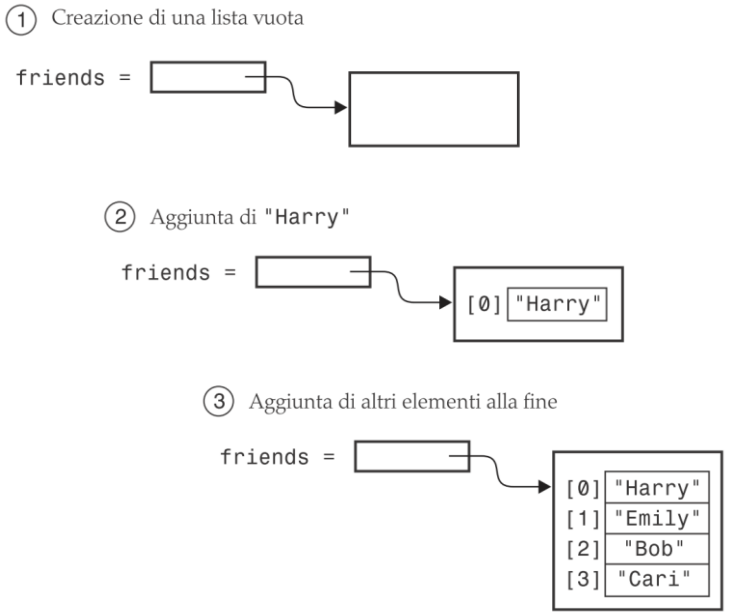
Aggiungere elementi

- Può capitare di non conoscere tutti i valori che saranno contenuti nella lista al momento della sua creazione
- In questo caso, è possibile creare una lista vuota e aggiungere successivamente gli elementi in coda alla lista, secondo necessità

```
#1
friends = []

#2
friends.append("Harry")

#3
friends.append("Emily")
friends.append("Bob")
friends.append("Cari")
```

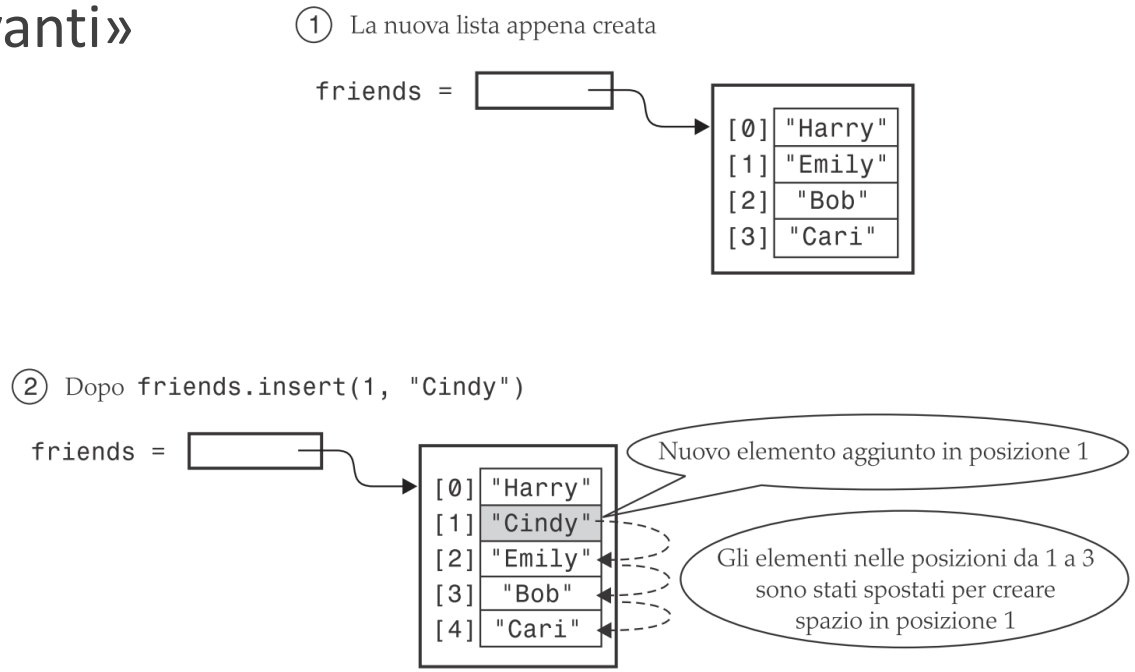


Inserire un elemento

- In alcuni casi l'ordine di aggiunta di ciascun elemento alla lista è importante
- È possibile inserire un nuovo elemento in una specifica posizione della lista
  - Gli altri elementi si sposteranno «in avanti»

```
#1
friends = ["Harry", "Emily",
"Bob", "Cari"]

#2
friends.insert(1, "Cindy")
```



Trovare un elemento

- Se si vuole sapere se un elemento è presente in una lista, si usa l'operatore `in`

```
if "Cindy" in friends :
    print("She's a friend")
```

- Il risultato è un valore booleano:
  - `True` (vero) se l'elemento è contenuto nella lista
  - `False` (falso) se l'elemento **non** è contenuto nella lista
  - Solitamente usato come condizione nelle istruzioni `while` o `if`

Trovare un elemento

- Spesso è utile conoscere la posizione di un elemento
  - Il metodo `index()` fornisce l'indice della prima corrispondenza
  - Questo metodo si applica alla lista e restituisce un valore intero

```
friends = ["Harry", "Emily", "Bob", "Cari", "Emily"]
n = friends.index("Emily") # imposta n a 1
```

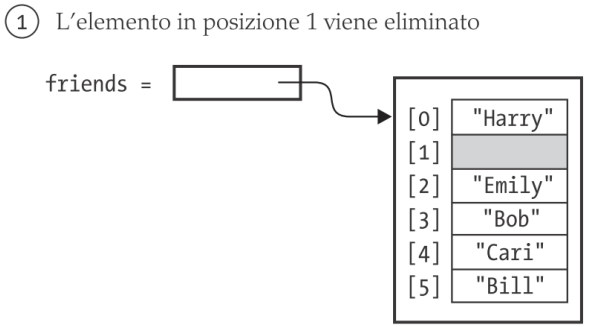
- Se l'elemento non viene trovato nella lista, si genera un ValueError
  - Meglio sempre controllare prima con l'operatore `in` se l'elemento è presente

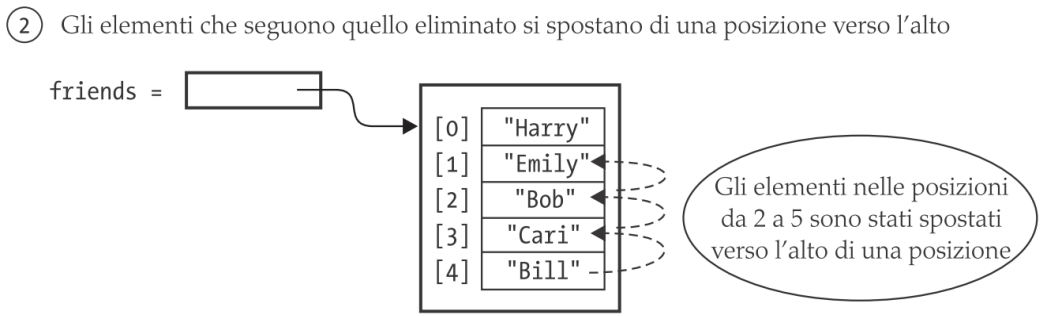
Rimuovere un elemento

- Il metodo `pop()` rimuove l'elemento nella posizione data

```
friends = ["Harry", "Cindy", "Emily", "Bob", "Cari","Bill"]
friends.pop(1)
```

- Tutti gli elementi che seguono quello rimosso vengono spostati verso l'alto per riempire il vuoto creatosi
  - La lunghezza della lista è quindi ridotta di 1





Concatenazione

- La concatenazione di due liste crea una nuova lista che contiene gli elementi della prima lista seguiti da quelli della seconda
- Due liste possono essere concatenate usando l'operatore più (**+**)

```
myFriends = ["Fritz", "Cindy"]
yourFriends = ["Lee", "Pat", "Phuong"]
```

```
ourFriends = myFriends + yourFriends
# imposta ourFriends a ["Fritz", "Cindy", "Lee", "Pat", "Phuong"]
```

Aggiunta (estensione) di una lista

- Per aggiungere in coda ad una lista il contenuto di una seconda lista:

```
pari = [2, 4, 6, 8]
dispari = [1, 3, 5]
# voglio ottenere: pari = [2, 4, 6, 8, 1, 3, 5]
```

- `pari = pari + dispari`
  - N.B. crea una nuova lista
- `pari.extend(dispari)`
  - N.B. modifica la lista esistente

⚠Non confondere:

```
lista.append(un_solo_elemento)
lista.extend(una_lista_intera)
```

Replicazione

- La replicazione di una lista genera molte copie dei suoi elementi (similmente alla replicazione delle stringhe)

```
monthInQuarter = [ 1, 2, 3 ] * 4
```

  - Risulta nella lista `[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2 ,3]`

- L'intero specifica quante copie della lista dovranno essere concatenate
  - È possibile mettere il numero intero in entrambi i lati dell'operatore "*"

- Un uso comune della replicazione è l'inizializzazione di una lista con un valore fisso

```
monthlyScores = [0] * 12
```

Test di uguaglianza / differenza

- L'operatore == è usato per verificare se due liste hanno gli stessi elementi nello stesso ordine

```
[1, 4, 9] == [1, 4, 9]    # True
[1, 4, 9] == [4, 1, 9]    # False.
```

- Il contrario di == è !=

```
[1, 4, 9] != [4, 9]     # True.
```

Somma, Massimo, Minimo

- Avendo una lista di numeri, la funzione `sum()` fornisce la somma di tutti i valori della lista.

```
sum([1, 4, 9, 16]) # restituisce 30
```

- Per una lista di numeri *o stringhe*, le funzioni `max()` e `min()` restituiscono il valore massimo e minimo

```
max([1, 16, 9, 4])                #restituisce 16
min(["Fred", "Ann", "Sue"])       #restituisce  "Ann"
```

Ordinamento

- Il metodo `sort()` ordina una lista di numeri o stringhe, dal valore minore al maggiore
- L'ordinamento avviene «sul posto», modificando la lista stessa

```
values = [1, 16, 9, 4]
values.sort() # ora saranno [1, 4 , 9, 16]
```

- La funzione `sorted()` crea una **nuova** lista ordinata, senza modificare la lista di partenza

```
values = [1, 16, 9, 4]
other = sorted(values) # other conterrà [1, 4 , 9, 16]
                       # values rimane immutata
```

Ordinamento inverso

- Normalmente il metodo `sort` e la funzione `sorted` ordinano gli elementi dal più piccolo al più grande
- Per ordinare in ordine inverso (dal più grande al più piccolo) si può usare il parametro opzionale `reverse=True`

```
values = [1, 16, 9, 4]
values.sort(reverse=True) # ora saranno [16, 9, 4, 1]
```

```
values = [1, 16, 9, 4]
other = sorted(values, reverse=True)
    # other conterrà [16, 9, 4, 1]
    # values rimane immutata
```

Porzioni (*slice*) di una lista

- Il meccanismo di 'slicing' già visto per le stringhe, funziona nello stesso modo anche per le liste
- Esempio (temperature di 12 mesi):

```
temperatures = [18, 21, 24, 33, 39, 40, 39, 36, 30, 22, 18]
```

  - Estraiamo le temperature del terzo quadrimestre (indici 6, 7 e 8) usando l'operatore di slice:

```
thirdQuarter = temperatures[6 : 9]
```

- Gli argomenti sono l'indice del primo elemento (incluso, se omesso vale 0) e dell'ultimo (escluso, se omesso indica la fine della lista)
- Il terzo argomento anche qui rappresenta il *passo* (positivo o negativo)

Esempi

`temperatures[ : 6]`

- Inclusi gli elementi dal primo al 6 (escluso), indici 0, 1, 2, 3, 4, 6

`temperatures[6 : ]`

- Inclusi gli elementi dal 6 (compreso) fino alla fine della lista, indici 6, 7, 8, 9, 10, 11

`temperatures[ : ]`

- Tutti gli elementi, dal primo all'ultimo (fa una copia, equivalente a `list(temperatures)`)

- `temperatures[::2]`
  - tutti gli elementi di indice pari (0, 2, 4, 6, 8, 10)
- `temperatures[::-1]`
  - tutta la lista, in ordine inverso, indici 11, 10, 9, 8, … 2, 1, 0

Assegnazione a Porzioni (1)

- È possibile assegnare valori alla porzione:
  `temperatures[6 : 9] = [45, 44, 40]`
- Essi rimpiazzeranno gli elementi di indice 6, 7 e 8
  - Gli altri elementi non verranno modificati

Novità: con le stringhe non era possible modificare una porzione, con le liste invece si può!

Le liste *non* sono immutabili

Assegnazione a Porzioni (2)

- Se la lunghezza della lista ‘rimpiazzo’ è diversa dalla lunghezza della porzione, verranno aggiunti o rimossi gli elementi in eccesso o in difetto
- Concettualmente, ciò avviene in 2 passi
  1. Eliminare dalla lista gli elementi corrispondenti alla slice selezionata
  2. Inserire in quella posizione gli elementi della nuova lista
- Esempi
  - `temperatures[ 6:10 ] = [ 3, 2 ]`
    - Sostituisco una porzione di 4 elementi con una lista di 2 elementi: sto «accorciando» la lista
  - `temperatures[ 6:8 ] = [ 3, 3, 3, 3, 3 ]`
    - Sostituisco una porzione di 2 elementi con una lista di 5 elementi: sto «allungando» la lista
  - `temperatures[ 6:8 ] = []`
    - Sostituisco una porzione di 2 elementi con una lista di 0 elementi: sto «cancellando» la porzione
  - `temperatures[ : ] = []`
    - Cancello tutti gli elementi della lista (equivalente a `temperatures.clear()`)

Funzioni e operatori comuni sulle liste

Table 1 Common List Functions and Operators

Operation	Description
`[]` `[`$elem_1$, $elem_2$, ..., $elem_n$`]`	Creates a new empty list or a list that contains the initial elements provided.
`len(`*l*`)`	Returns the number of elements in list *l*.
`list(`*sequence*`)`	Creates a new list containing all elements of the sequence.
`values * num`	Creates a new list by replicating the elements in the `values` list `num` times.
`values + moreValues`	Creates a new list by concatenating elements in both lists.

Funzioni e operatori comuni sulle liste (2)

Table 1 Common List Functions and Operators

Operation	Description
`l[from : to]`	Creates a sublist from a subsequence of elements in list *l* starting at position `from` and going through but not including the element at position `to`. Both `from` and `to` are optional. (See Special Topic 6.2.)
`sum(l)`	Computes the sum of the values in list *l*.
`min(l)` `max(l)`	Returns the minimum or maximum value in list *l*.
l_1 == l_2	Tests whether two lists have the same elements, in the same order.

Metodi comuni sulle liste

Table 2 Common List Methods

Method	Description
l.pop() *l*.pop(*position*)	Removes the last element from the list or from the given position. All elements following the given position are moved up one place.
l.insert(*position*, *element*)	Inserts the element at the given position in the list. All elements at and following the given position are moved down.
l.append(*element*)	Appends the element to the end of the list.
l.index(*element*)	Returns the position of the given element in the list. The element must be in the list.
l.remove(*element*)	Removes the given element from the list and moves all elements following it up one position.
l.sort()	Sorts the elements in the list from smallest to largest.

Operatori sulle liste

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

https://docs.python.org/3/library/stdtypes.html#common-sequence-operations

Operazioni per modificare liste

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from *s* (same as `del s[:]`)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*
`s.reverse()`	reverses the items of *s* in place

https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types

Metodi delle liste

Python List Methods
append() - Add an element to the end of the list
extend() - Add all elements of a list to the another list
insert() - Insert an item at the defined index
remove() - Removes an item from the list
pop() - Removes and returns an element at the given index
clear() - Removes all items from the list
index() - Returns the index of the first matched item
count() - Returns the count of the number of items passed as an argument
sort() - Sort items in a list in ascending order
reverse() - Reverse the order of items in the list
copy() - Returns a shallow copy of the list

https://www.programiz.com/python-programming/list

Algoritmi comuni con le liste

6.3

Algoritmi comuni con le liste

- Riempire una lista
- Combinare elementi della lista
- Separatori di elementi
- Massimo e minimo
- Ricerca lineare
- Trovare e contare corrispondenze
- Rimuovere corrispondenze
- Scambiare elementi
- Leggere imput

Provali PythonTutor!

Provali su replit.com!

Riempire una lista

- Questo ciclo crea e riempie una lista con i quadrati dei numeri interi (0, 1, 4, 9, 16, ..., $(n-1)^2$)

```
values = []
for i in range(n) :
    values.append(i * i)
```

Combinare elementi della lista

- Per elaborare una somma di numeri:

```
result = 0.0
for element in values :
    result = result + element
```

- Per concatenare stringhe, occorre avere un valore iniziale di tipo stringa

```
result = ""
for element in names :
    result = result + element
```

Separatori di elementi

- Quando si visualizzano gli elementi di una lista, solitamente è utile separarli tramite l'uso di virgole, o linee verticali. Es.:

```
Harry, Emily, Bob
```

Separatori di elementi (2)

- Aggiungere un separatore prima di ogni elemento (c'è un separatore in meno rispetto ai numeri) della sequenza ad eccezione del primo (con indice 0), così:

```
names = ["marco", "paolo", "marta"]
```

```
result = ''

for i in range(len(names)) :
    if i > 0 :
        result = result + ", "
    result = result + names[i]
```

```
result = names[0]
# nota: si assume che len(names)>=1

for i in range(1, len(names)) :
  result = result + ", " + names[i]
```

```
print(result)
```

Separatori di elementi (3)

- Per stampare direttamente i valori, senza prima aggiungerli ad una stringa, è possibile impedire il ‘a capo’ automatico:

```
for i in range(len(values)) :
    if i > 0 :
        print(" | ", end="")
    print(values[i], end="")
print()
```

Scorciatoie di Python: `.join()`

- Il metodo `.join` delle stringhe unisce automaticamente gli elementi di una lista, usando una stringa come separatore
  `separator_string.join(list)`

```
result2 = ', '.join(names)

print(result2)
```

`str.`**`join`**`(`*`iterable`*`)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

https://docs.python.org/3/library/stdtypes.html#str.join

Scorciatoie Python: `.split()`

`str.`**`split`**`(`*`sep=None, maxsplit=-1`*`)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

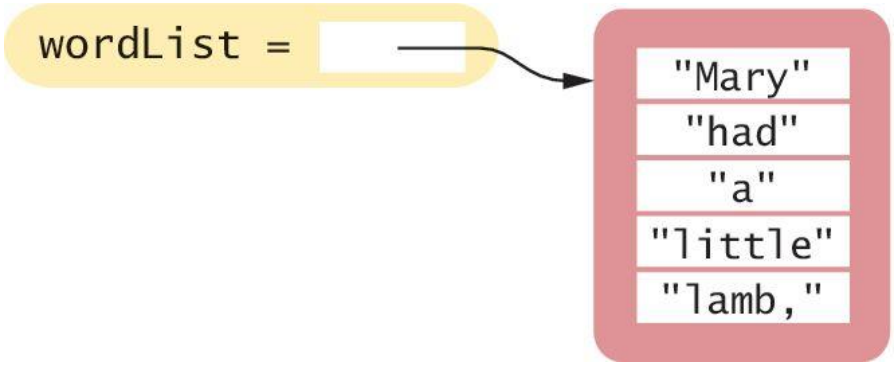
https://docs.python.org/3/library/stdtypes.html#str.split

Split

- Il metodo `split()` restituisce una lista delle sottostringhe risultanti dalla divisione della stringa in corrispondenza di ciascuno spazio (o tab o newline)
- Per esempio, se `line` contiene la stringa:

- Sarà divisa in 5 sotto-stringhe, memorizzate nello stesso ordine in cui compaiono nella stringa

```
wordList = line.split()
```



Massimo e minimo

- Ecco le implementazioni degli algoritmi per la ricerca di massimo e minimo

```
largest = values[0]
for i in range(1, len(values)) :
    if values[i] > largest :
        largest = values[i]
```

```
# equivalente a
largest = max(values)
```

```
smallest = values[0]
for i in range(1, len(values)) :
    if values[i] < smallest :
        smallest = values[i]
```

```
# equivalente a
largest = min(values)
```

Ricerca lineare

- Ricerca del primo numero che sia > 100.
- È necessario controllare tutti gli elementi fino a trovare una corrispondenza o arrivare alla fine della lista

```
limit = 100
pos = 0
found = False
while pos < len(values) and not found :
    if values[pos] > limit :
        found = True
    else :
        pos = pos + 1
if found :
    print("Found at position:", pos)
else :
    print("Not found")
```

Una ricerca lineare ispeziona gli elementi di una lista fino a trovare una corrispondenza.

Trovare e contare corrispondenze

- Trovare corrispondenze

```
limit = 100
result = []
for element in values :
    if (element > limit) :
        result.append(element)
```

- Contare corrispondenze

```
limit = 100
counter = 0
for element in values :
    if (element > limit) :
        counter = counter + 1
```

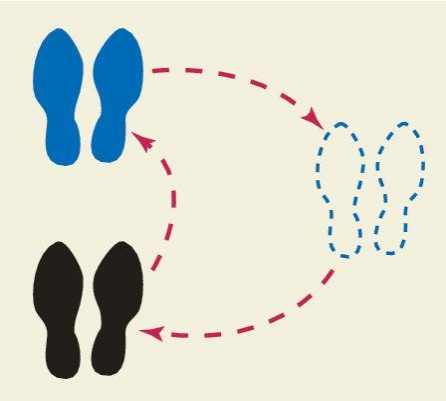
Rimuovere corrispondenze

- Rimuovere tutti gli elementi che rientrano in particolari condizioni
  - Esempio: rimuovere tutte le stringhe di lunghezza < 4 da una lista

```
i = 0
while i < len(words) :
    word = words[i]
    if len(word) < 4 :
        words.pop(i) # cancellare l'elemento i-esimo
        # NON incrementare i
    else :
        i = i + 1
```

Scambiare elementi

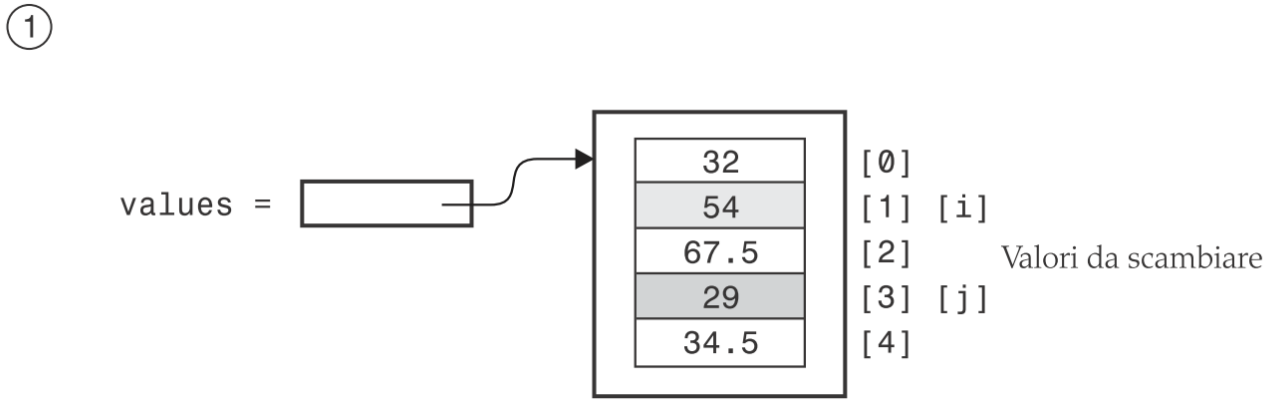
- Un modo per ordinare una lista è attraverso ripetuti scambi di elementi che non sono in ordine
- L'operazione fondamentale è scambiare gli elementi in posizioni `i` e `j` di una lista di valori
- È necessario copiare in `values[i]` il valore `values[j]`. Ma questo sovrascriverebbe il valore precedente di `values[i]`, quindi occorrerà salvarlo prima:



Prima di spostare un nuovo valore (nero) in un luogo (blu), copiare il valore blu da qualche parte, poi spostare il nero nel blu. Successivamente, spostare il valore temporaneo (che originalmente era blu) nel nero.

Scambiare elementi (2)

- Scambio degli elementi `[1]` e `[3]`
  - Consideriamo la situazione di partenza illustrata in figura

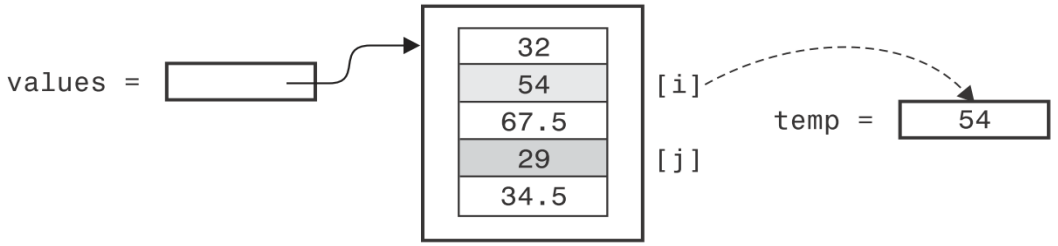


Scambiare elementi(3)

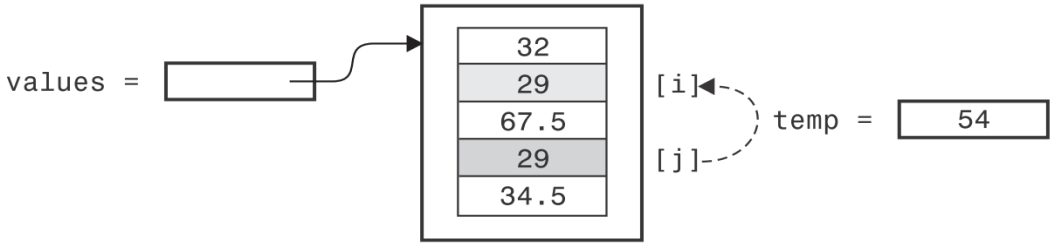
```
# Step 1
temp = values[i]

# Step 2
values[i] = values[j]
```

②

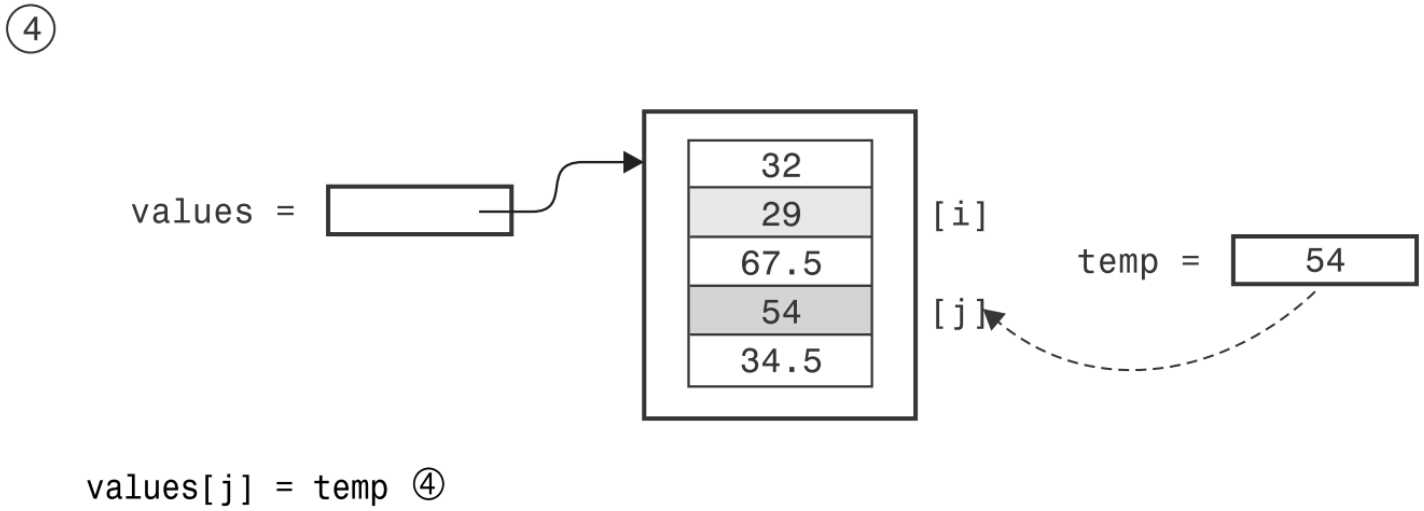


③



Scambiare elementi (4)

```
# Step 3
# temp contains values[i]
values[j] = temp
```



Scorciatoie Python: assegnazioni multiple

- È possibile fare in più di un'assegnazione in parallelo (ed evitando la variabile temporanea) usando una sintassi con una 'tupla'
- `(a, b) = (3, 4)` è equivalente a a=3 e b=4,
  - allo stesso tempo `(a, b) = (b, a)` scambia `a` e `b`

```
# scorciatoia per lo scambio

( values[1], values[3] ) = ( values[3], values[1] )
```

Leggere valori in input

- È molto comune leggere input da un utente e raccoglierli in una lista per poi elaborarli

```
values = []
print("Please enter values, Q to quit:")
userInput = input("")
while userInput.upper() != "Q" :
    values.append(float(userInput))
    userInput = input("")
```

```
Please enter values, Q to quit:
32
29
67.5
Q
```

Esecuzione del programma

Riepilogo operazioni predefinite per liste

- Il metodo `.insert()` inserisce un elemento in una data posizione della lista
- L'operatore `in` verifica se un elemento sia contenuto in una lista
- Il metodo `.pop()` rimuove un elemento da una data posizione della lista
- Il metodo `.remove()` toglie un elemento da una lista sulla base del suo valore
- Due liste possono essere concatenate usando l'operatore (`+`)
- La funzione `list()` crea la copia di una lista esistente
- L'operatore di slice (`:`) estrae una sottolista o una sottostringa

⚠ Errore frequente

```
mesi = [ "Gen", "Feb", "Mar", "Apr", "Mag", "Giu"]

for mese in mesi:
    pos = mesi.index(mese)
    print(f'Il mese {pos+1} è {mese}')
```

- Ri-calcola un valore che potresti **già conoscere** (indice = numero di iterazione)
- Il metodo `index` è obbligato a ricercare in tutta la lista → **inefficiente**
- In caso di elementi duplicati, restituisce sempre il primo → **errore**

✔ Errore frequente: soluzione (scomoda)

```
mesi = [ "Gen", "Feb", "Mar", "Apr", "Mag", "Giu"]

for pos in range(len(mesi)):
    mese = mesi[pos]
    print(f'Il mese {pos+1} è {mese}')
```

- Per evitare il problema, si può iterare sull'indice anziché sul valore
- Ad ogni iterazione, occorre *estrarre* il valore (operazione efficiente)

✔✔ Errore frequente: soluzione migliore

```
mesi = [ "Gen", "Feb", "Mar", "Apr", "Mag", "Giu"]

for (pos, mese) in enumerate(mesi):
    print(f'Il mese {pos+1} è {mese}')
```

- Con **enumerate**, l'iterazione conosce sempre sia l'indice che il valore dell'elemento
- Non faccio lavoro inutile
- Funziona anche se ci fossero dei duplicati

Esempio

- Aprire il file largest.py

- Analizzare il codice e provare con diversi dati di ingresso

Esempi di problemi

- Aprire il file largest.py

largest.py

- Modificare il programma per trovare e visualizzare il numero più piccolo, oltre a quello più grande
  - Trovare il maggiore
  - Stampare la lista
    - Stampare la stringa " <== largest value " vicino al valore trovato
  - Trovare il valore minore
  - Stampare la lista
    - Stampare la stringa " <== smallest value " vicino valore trovato
- Ulteriore modifica del programma
  - Trovare il numero maggiore
  - Trovare il numero minore
  - Stampare la lista
    - Stampare la stringa " <== largest value " vicino al valore maggiore
    - Stampare la stringa " <== smallest value " vicino al valore minore

Uso di liste nelle funzioni

6.4

Uso di liste con funzioni

- Una funzione può accettare una lista come argomento
- La seguente funzione visita gli elementi della lista, senza però modificarli

```
def sumsq(values) :
  total = 0
  for element in values :
    total = total + element**2
  return total
```

Provalo con PythonTutor!

Modificare elementi di una lista

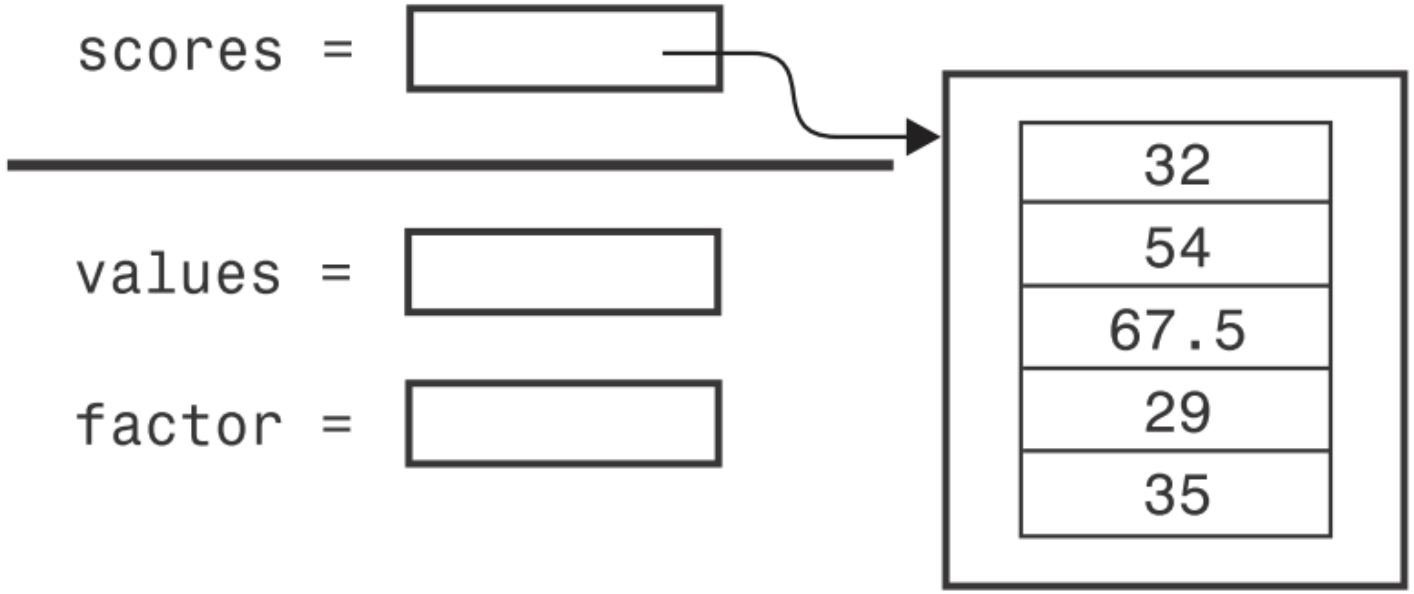
- La seguente funzione moltiplica tutti gli elementi di una lista per un fattore dato
- Viene modificata la lista originale!

```
def multiply(values, factor) :
    for i in range(len(values)) :
        values[i] = values[i] * factor
```

Provalo con PythonTutor!

Esempio: Step 1

- Vengono create le variabili parametro `values` e `factor`

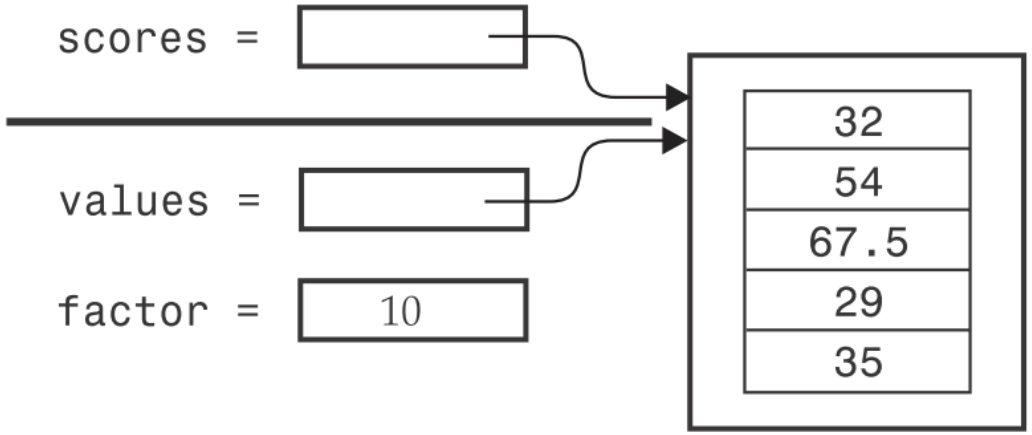


Esempio: Step 2

- Le variabili parametro sono inizializzate con gli argomenti passati nella chiamata della funzione

```
# chiamata della funzione
multiply(scores, 10)
```

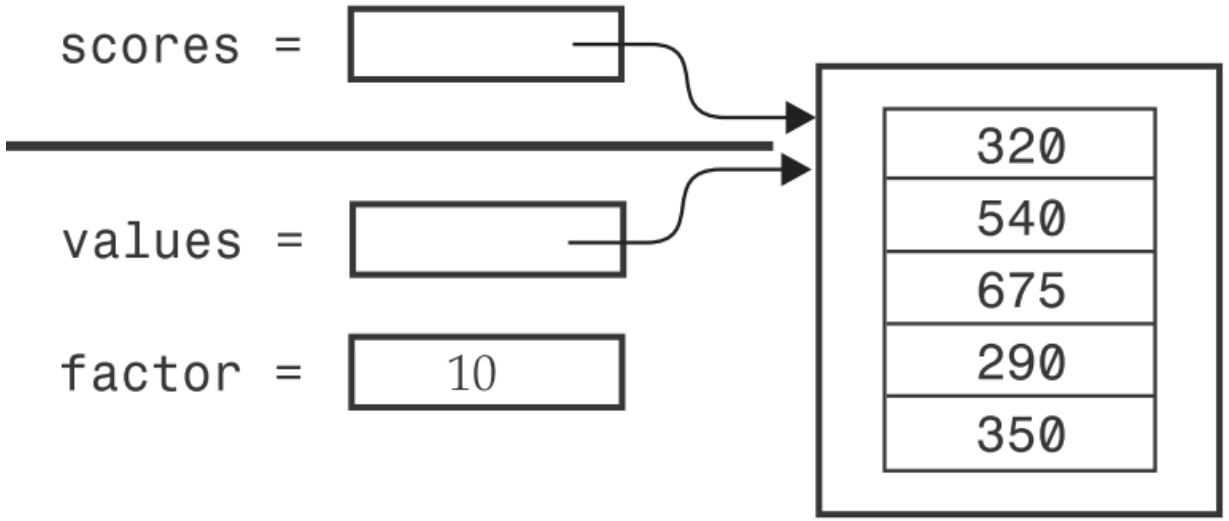
- In questo caso, `values` è inizializzato con il valore di `scores` e `factor` con `10`
  - Notare che `values` e `scores` si riferiscono alla stessa lista (`values` è un ***alias*** di `scores`)



Esempio: Step 3

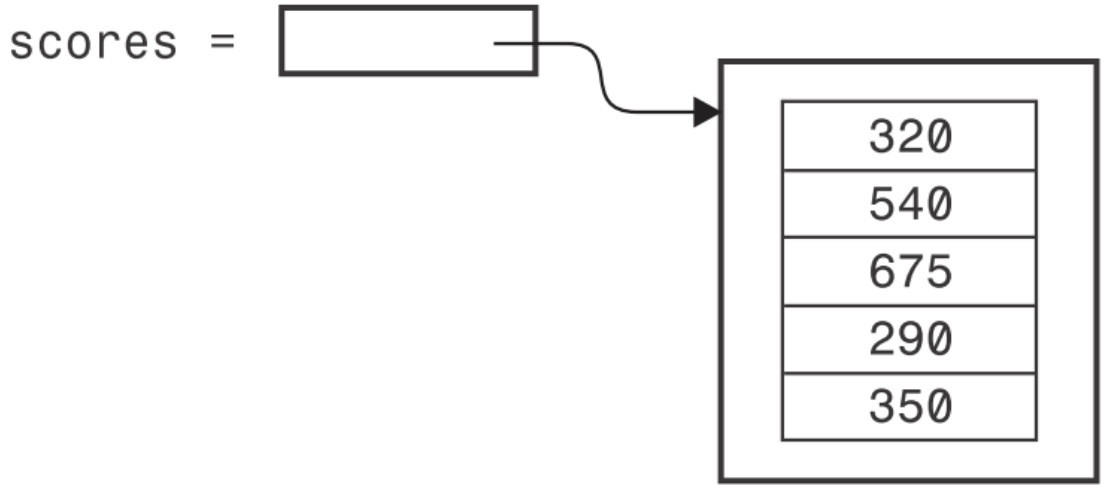
- La funzione moltiplica tutti gli elementi per 10

```
def multiply(values, factor) :
    for i in range(len(values)) :
        values[i] = values[i] * factor
```



Esempio: Step 4

- La funzione termina. Le sue variabili parametro vengono rimosse
- Al contrario, `scores` continua a riferirsi alla lista con gli elementi modificati



Restituire liste da funzioni

- È sufficiente costruire il risultato della funzione e restituirlo
- In questo esempio, la funzione `squares()` restituisce una lista di quadrati da 0^2 fino a $(n - 1)^2$

```
def squares(n) :
    result = []
    for i in range(n) :
        result.append(i * i)
    return result
```

Esempio

- Aprire e studiare il file reverse.py

- Questo programma leggere valori dall'utente, li moltiplica per 10 e li stampa in ordine inverso
- La funzione read_floats restituisce una lista
- La funzione moltiplicazione ha un argomento di tipo lista e modifica gli elementi di essa
- La funzione print_reversed ha un argomento di tipo lista , ma non modifica gli elementi della lista

⚠ Non confondere alias e copia

- La variabile parametro è inizializzata come alias dell'argomento
- Attraverso l'alias, è possibile modificare il valore dell'argomento
  - Se esso non è di tipo immutabile (numero, stringa)
  - Usando i metodi e gli operatori che agiscono sul valore
- Non ri-assegnare mai la variabile, altrimenti si «perde» l'alias
  - Mai usare una variabile parametro alla sinistra del segno **=**

```
# chiamata della funzione
multiply(scores, 10)
```

```
def multiply(values, factor) :
    . . .
```

```
values[i] = values[i]*2.0
values.append(5)
values.insert(4)
values.pop()
values.extend([2,3,4])
values.sort()
```

✓

```
values = []
values = values + [3]
values = sorted(values)
```

✗

Tuple

Tuple

- Una tupla è simile ad una lista, ma una volta creata, il suo contenuto non può essere modificato (una tupla è una versione immutabile di una lista)

- Quindi, una tupla è un contenitore ordinato e immutabile

- Nota: abbiamo già usato le tuple, pur senza conoscerle completamente, come valori di ritorno multipli dalle funzioni e con la funzione enumerate() utilizzata per ciclare su una lista

Definire Tuple

- Una tupla viene creata specificandone il contenuto in una sequenza separata da virgole. Per indicare che si tratta di una tupla, si racchiude la sequenza fra parentesi ( )

```
triple = (5, 10, 15)
```

- Se si preferisce, si possono anche omettere le parentesi, ma per chiarezza non è consigliato:

```
triple = 5, 10, 15
```

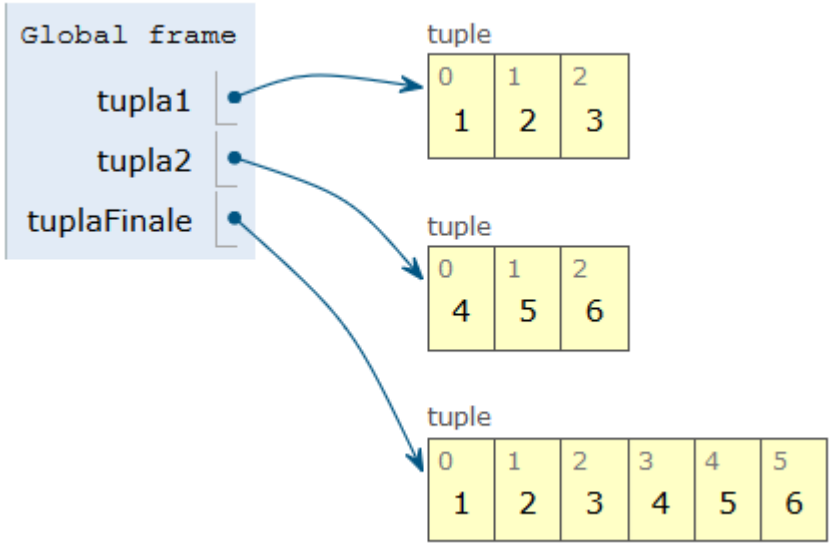
- Nota: le tuple sono una struttura dati avanzata, con molte altre caratteristiche. In questo corso non verranno studiate in modo approfondito.

Operazioni sulle tuple (mutuate da list)

- Accesso ad elementi con `[ ]`
- Concatenazione con operatore `+`

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)

tuplaFinale = tupla1 + tupla2
```

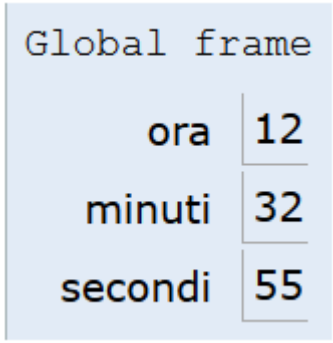


- Funzioni `len(), sum(), max()`
- Iterare sugli elementi con `for`
- Verificare appartenenza con `in` e `not in`

Assegnazione di tuple

- In Python (ma solo in Python) si possono assegnare variabili multiple in una sola operazione usando le tuple

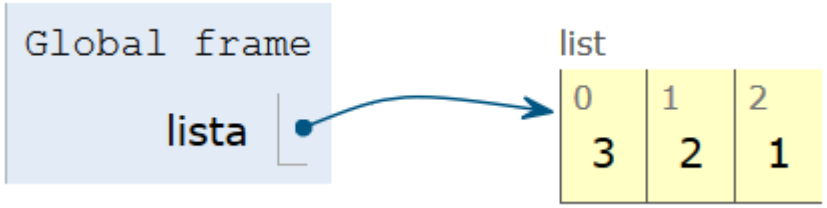
```
(ora, minuti, secondi) = (12, 32, 55)
```



- Utile anche per lo scambio di variabili…

```
lista = [1, 2, 3]

(lista[0], lista[2]) = (lista[2], lista[0])
```



Restituzione di valori multipli

- È pratica comune in Python usare tuple per restituire valori multipli da una funzione

```
# definizione della funzione
def read_date() :
    print("Enter a date:")
    month = int(input(" month: "))
    day = int(input(" day: "))
    year = int(input(" year: "))
    return (month, day, year) # restituisce una tupla.

# chiamata della funzione: assegna i valori interi ai
campi della tupla
    date = read_date()

# chiamata della funzione: estrazione dei componenti
della tupla
(month, day, year)  = read_date()
```

Scorciatoie di Python: Comprehension

https://realpython.com/list-comprehension-python/

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

Schema per la creazione di liste

- Spesso occorre **creare una lista** (o una tupla) **a partire da un'altra** struttura di tipo lista/tupla/stringa/range, applicando una determinata operazione a tutti gli elementi
- Lo schema predominante utilizza il metodo **append** per costruire il risultato in elemento alla volta, all'interno di un ciclo
- Chiaro, funzionante… ma lungo

```
giorni = ['Lunedì', 'Martedì', 'Mercoledì', 'Giovedì',
'Venerdì', 'Sabato', 'Domenica']

lunghezza = []
for giorno in giorni:
  lunghezza.append( len(giorno) )

print(lunghezza)

# [6, 7, 9, 7, 7, 6, 8]

invertiti = []
for giorno in giorni:
  invertiti.append( giorno[::-1] )
print(invertiti)

# ['ìdenuL', 'ìdetraM', 'ìdelocreM', 'ìdevoiG', 'ìdreneV',
'otabaS', 'acinemoD']
```

List Comprehension*

* Nel significato arcaico di: «*inclusion*»

- Sintassi semplificata per **creare** una **lista** a partire dall'iterazione e modifica degli elementi di un **altro contenitore**

```
[ espressione(elemento)
for elemento in contenitore ]
```

- `[ ]` : costruiamo una lista
- **`for`** *`elemento`* **`in`** *`contenitore`* : definisce l'iterazione
- *`espressione(elemento)`* : l'elaborazione sugli elementi del contenitore, per ottenere gli elementi del risultato

```
lunghezza = []
for giorno in giorni:
    lunghezza.append( len(giorno) )
```

```
lunghezza2 = [ len(giorno) for giorno in giorni ]

invertiti2 = [ giorno[::-1] for giorno in giorni ]
```

```
invertiti = []
for giorno in giorni:
    invertiti.append( giorno[::-1] )
```

List Comprehension con condizione

- Nella creazione della nuova lista, si possono selezionare solo gli elementi che soddisfano una certa condizione

```
[ espressione(elemento)
for elemento in contenitore
 if condizione(elemento) ]
```

```
accentati = [ giorno for giorno in giorni if giorno[-1]=='ì' ]

# ['Lunedì', 'Martedì', 'Mercoledì', 'Giovedì', 'Venerdì']
```

```
# multipli di 3 o di 5
fizzbuzz = [ n for n in range(1,101)
    if n % 3 == 0 or n % 5 == 0]

# [3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25, 27, 30, 33, 35,
 36, 39, 40, 42, 45, 48, 50, 51, 54, 55, 57, 60, 63, 65, 66,
69, 70, 72, 75, 78, 80, 81, 84, 85, 87, 90, 93, 95, 96, 99,
100]
```

List Comprehension con cicli annidati

- È possibile indicare più di un contenitore su cui ciclare, semplicemente ripetendo la parte «for elemento in contenitore» più volte

```
[ espressione(elem1, elem2)
  for elem1 in contenitore1
  for elem2 in contenitore2
if condizione(elem1, elem2) ]
```

```
righe = 'ABCD'
colonne = '1234'

caselle = [ r+c for r in righe for c in colonne ]

# ['A1', 'A2', 'A3', 'A4', 'B1', 'B2', 'B3', 'B4', 'C1', 'C2',
  'C3', 'C4', 'D1', 'D2', 'D3', 'D4']
```

È consigliabile solo se non impatta la leggibilità del codice!

Comprehension per altre strutture dati

- La sintassi delle comprehension funziona con
  - Liste
  - Tuple
  - Set (v. unità P8)
  - Dizionari (v. unità P8)
- È sufficiente utilizzare il tipo corretto di parentesi per creare la struttura dati voluta

```
lista =      [      i*i for i in range(0,10)      ]
tupla =      tuple( i*i for i in range(0,10)      )
insieme =    {      i*i for i in range(0,10)      }
dizionario = {      i+1 : i*i for i in range(0,10) }
```

Adattamento di algoritmi

6.5

Adattare algoritmi

- Dati i risultati del quiz di uno studente, calcolare il punteggio finale, che è la somma di tutti i risultati dopo aver **escluso il più basso**
  - Per esempio, se i punteggi parziali fossero:

```
8    7    8.5    9.5    7    5    10
```

  - allora il punteggio finale sarebbe 50

Adattare una soluzione

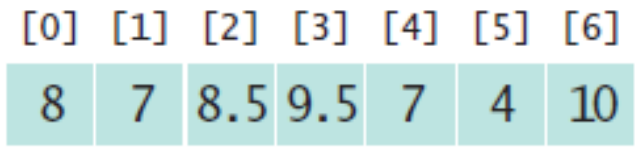
- Quali sono gli step per la soluzione?
  - Trovare il minimo
  - Rimuoverlo dalla lista
  - Calcolare la somma

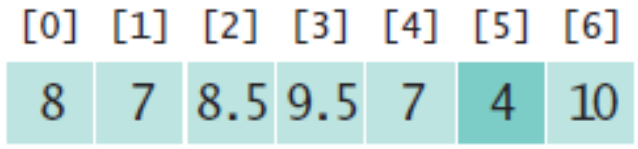
[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	7	8.5	9.5	7	4	10

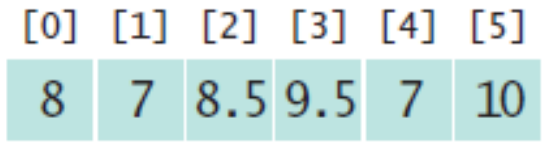
- Quali strumenti ci sono a disposizione?
  - Ricerca del valore minimo (sezione 6.3.4)
  - Rimozione delle corrispondenze (sezione 6.3.7)
  - Calcolo della somma (sezione 6.4)
- Ma attenzione… è necessario trovare la POSIZIONE del valore minimo, non il valore stesso
  - mmmh. Tempo di adattamenti

Programmazione di una soluzione

- Ridefinizione degli step:
  - Ricerca del valore minimo
  - Trovare la sua posizione
  - Rimuoverlo dalla lista
  - Calcolo della somma
- Ragioniamo per analogia
  - Ricerca della posizione del valore minimo :
    - Posizione 5
  - Rimozione dalla lista
  - Calcolo della somma







Adattamento del codice

- **Adattare** il calcolo del valore più piccolo al calcolo della posizione del minimo:

Algoritmo originale

```
smallest = values[0]
for i in range(1, len(values)) :
    if values[i] < smallest :
        smallest = values[i]
```

Algoritmo adattato

```
smallestPosition = 0
for i in range(1, len(values)) :
    if values[i] < values[smallestPosition] :
        smallestPosition = i
```

Esempio di problema

- Testo del problema: il risultato finale del quiz di uno studente è calcolato sommando tutti i punteggi esclusi i **due** più bassi
  - Per esempio, se i punteggi fossero: `8, 4, 7, 9, 9, 7, 5, 10`
  - Il risultato finale sarebbe `50`
- Di seguito si svilupperà l'algoritmo e si scriverà un programma per il calcolo del punteggio finale

Step 1

- Per prima cosa, occorre un alto livello di scomposizione del problema:
  - Lettura dei dati nella lista
  - Elaborare i dati
  - Mostrare il risultato
- Per risolvere il problema sarà utile riferirsi alla lista di algoritmi e operazioni. La maggior parte dei compiti associati a questo problema sono risolvibili usando o adattando uno o più degli algoritmi
- Il prossimo passo nella ridefinizione degli step è l'identificazione degli step necessari elaborare i dati:
  - Lettura degli input
  - Rimozione del minimo
  - Ulteriore rimozione del minimo
  - Calcolo della somma

Step 2

- A questo punto, si inizia a determinare gli algoritmi che saranno necessari
- Sono stati presentati gli algoritmi per la lettura degli input e il calcolo della somma
- Per rimuovere il valore minimo si può trovare il minimo (conoscendo l'algoritmo a ciò deputato) e rimuoverlo.
  - Trovare la posizione del valore minimo e "pop" quella posizione

Step 3

- Pianificare le funzioni necessarie
  - Per calcolare la somma, uso della funzione `sum()`
  - Per la lettura dei numeri in floating point: `read_floats()`
  - Per rimuovere il minimo, la funzione `remove_minimum()` (che sarà necessario chiamare due volte)
- La funzione principale sarà strutturata così:

```
scores = read_floats()
Remove_minimum(scores)
Remove_minimum(scores)
total = sum(scores)
print('Final Score : ', total)
```

Step 4

- Assemblamento e prova del codice
- La revisione del codice è utile ad assicurarsi di poter gestire sia i casi “normali ” che quelli “eccezionali”
- Come è possibile gestire una lista vuota?
  - Una lista con un singolo elemento?
  - E se non si trova un solo valore minimo?
- Si ricorda che il problema richiede di escludere due dati
- Non è possibile cercare il minimo se la lista è vuota o formata da un solo elemento
  - In questi casi, si dovrà terminare il programma con un messaggio di errore prima di chiamare la funzione che rimuove il minimo
- Si dovrà quindi sviluppare test per questi casi e i rispettivi output

Test per le eccezioni

- Sviluppo dei test e rispettivi output

Caso di prova	Risultato previsto	Commento
`8 4 7 8.5 9.5 7 5 10`	`50`	Vedere la Fase 1.
`8 7 7 7 9`	`24`	Si devono eliminare due sole copie del voto minimo.
`8 7`	`0`	Dopo aver eliminato i due punteggi più bassi, non rimane alcun punteggio.
(nessun dato)	Errore	I dati non sono validi.

scores.py

- Aprire il file scores.py
- Analizzare la soluzione proposta

Esempio 2

- Testo del problema: è necessario analizzare se un dado è truccato contando quanto spesso ciascun valore (1, 2, 3, 4, 5, 6) compare
- Gli input saranno una sequenza di valori di lanci di dado
  - Per esempio, se i valori fossero: `1, 2, 1, 3, 4, 6, 5, 6`
  - Il risultato sarà: `1: 2; 2: 1; 3: 1; 4: 1; 5: 1; 6: 2`
- Si dovrà, quindi, sviluppare l'algoritmo e scrivere un programma che elabori e stampi la frequenza di ciascun valore del dado

Step 1

- Per prima cosa, vediamo la scomposizione del problema ad alto livello:
  - Lettura dei valori del dado
  - Contare quanto spesso ciascun valore (1, 2, …, 6) appare
  - Visualizzare il conteggio
- Pensando a cosa si potrebbe semplificare, è necessario memorizzare i valori?
  - Si sta solo contando quante volte ogni valore esce. Creando una lista di contatori, sarà possibile leggere gli input e poi scartarli
- Il prossimo passo nella ridefinizione degli step è l'identificazione degli step necessari processare i dati:
  - Lettura degli input
  - Per ogni valore di input:
    - Incrementare il contatore corrispondente
  - Stampare i contatori

Step 2

- Determinazione degli algoritmi che saranno necessari
- Non esiste (ancora) un algoritmo per la lettura di input e incremento di un contatore, ma è semplice crearne uno
  - Avendo una lista di lunghezza 6:

```
counters[value - 1] = counters[value - 1] + 1
```

- Per semplicità è possibile non usare la posizione [0] e avere:

```
counters[value] = counters[value] + 1
```

- Si può inizialmente definire: `counters = [0] * (sides + 1)`
- Ora è necessario concentrarsi sulla stampa dei contatori
- Si può usare un ciclo controllato dal contatore e usare un formato stringa per stampare i risultati

Step 3

- Pianificazione delle funzioni necessarie:
  - `countInputs(sides)` `# conterà gli input`
  - `printCounters(counters)` `# stamperà i contatori`

- La funzione principale chiamerà queste due funzioni:

```
counters = countInputs(6)
printCounters(counters)
```

Step 4

- Assemblare e testare il programma:
- Nell'aggiornare i contatori, occorre assicurarsi di non generare un errore di sconfinamento, respingendo i valori di input < 1 e > 6

Caso di prova	Risultato previsto	Commento
1 2 3 4 5 6	1 1 1 1 1 1	Ciascun numero ricorre una sola volta.
1 2 3	1 1 1 0 0 0	I numeri che non compaiono devono avere il contatore a zero.
1 2 3 1 2 3 4	2 2 2 1 0 0	I contatori devono rispecchiare la frequenza di ciascun valore in ingresso.
(nessun dato)	0 0 0 0 0 0	Questo è un insieme di dati valido: tutti i contatori devono essere a zero.
0 1 2 3 4 5 6 7	**Errore**	Ciascun dato in ingresso deve essere compreso tra 1 e 6.

dice.py

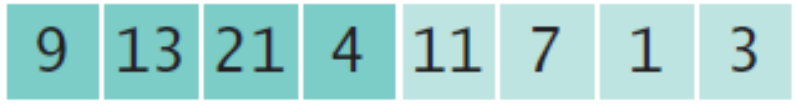
- Aprire il file dice.py
- Analizzare la soluzione proposta

Scoprire algoritmi manipolando oggetti fisici

6.6

Scoprire nuovi algoritmi

- Considerando il seguente problema:
  - Viene data una lista la cui dimensione è un numero pari, è richiesto di scambiare la prima metà con la seconda
- Per esempio, se la lista contiene 8 numeri:



- Si dovrà riarrangiarla così:

Manipolazione di oggetti

- Una tecnica utile per scoprire un algoritmo è la manipolazione di oggetti fisici
- Partiamo allineando alcuni oggetti per rappresentare una sequenza
  - Monete, carte da gioco o piccoli giocattoli sono ottime scelte

Manipolazione di oggetti

- Visualizzazione della rimozione di un oggetto

Manipolazione di oggetti

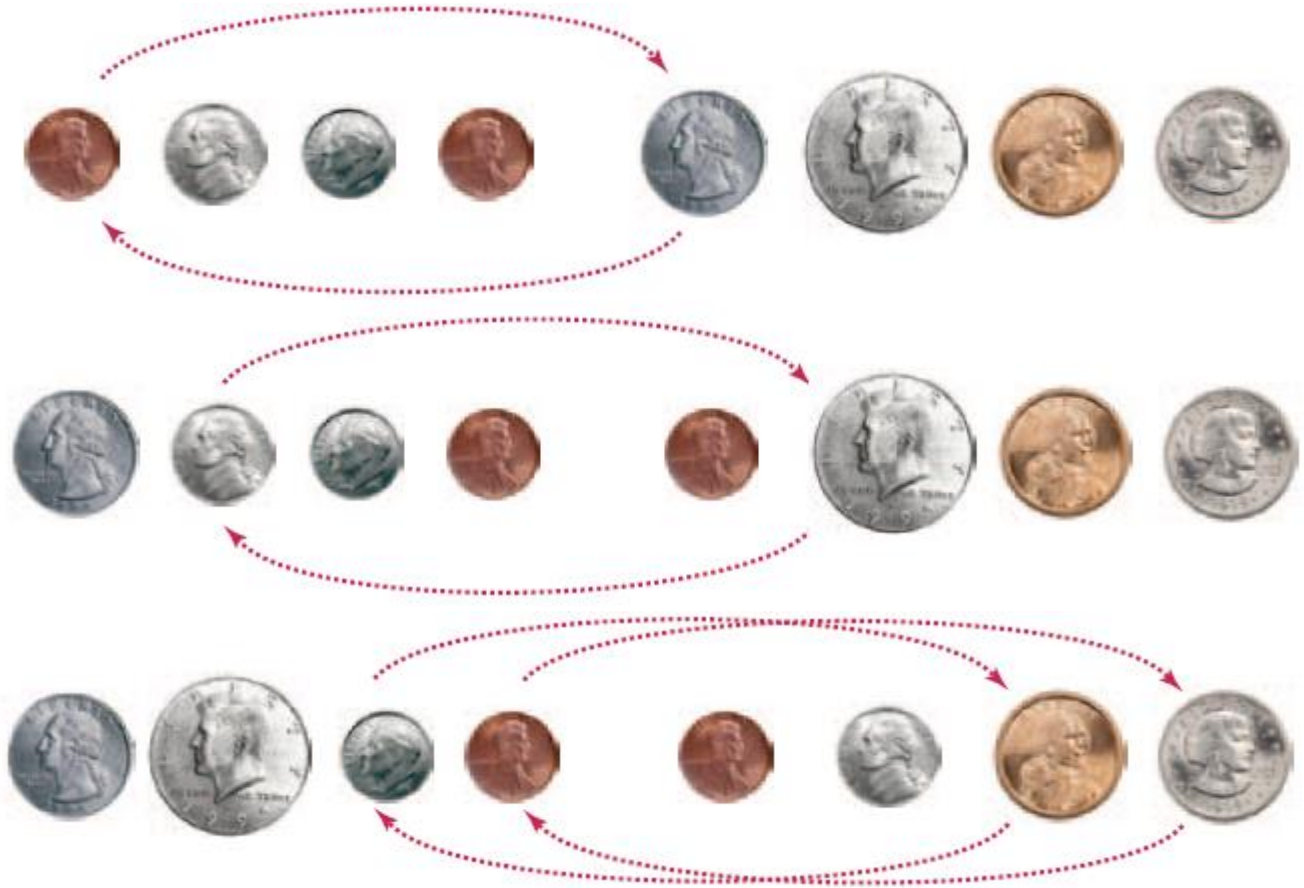
- Visualizzazione dell'aggiunta di un oggetto

Manipolazione di oggetti

- E per lo scambio di due oggetti?

Manipolazione di oggetti

- Tornando al problema originario, quali funzioni sono utili?
  - Si potrebbero effettuare degli scambi? Quattro volte?



Sviluppo di un algoritmo

- Prendendo due posizioni (indici) per il primo scambio e iniziare un ciclo
  - Il primo scambio è con i=0
- Come potrebbe essere impostato j per gestire qualsiasi numero di elementi?
  - … se la dimensione è 8, j è l'indice 4…
- E quando si dovrà interrompere il ciclo?


```
i = 0
j = ... (we'll think about that in a minute)
While (don't know yet)
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

Sviluppo di un algoritmo

- Prendendo due posizioni (indici) per il primo scambio e iniziare un ciclo
  - Il primo scambio è con i=0
- Come potrebbe essere impostato j per gestire qualsiasi numero di elementi?
  - … se la dimensione è 8, j è l'indice 4…
- E quando si dovrà interrompere il ciclo?


```
i = 0
j = size/2
While ( size/2 )
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

Sviluppo di un algoritmo

- Prendendo due posizioni (indici) per il primo sc
  ciclo
  - Il primo scambio è con i=0
- Come potrebbe essere impostato j per gestire c
  elementi?
  - … se la dimensione è 8, j è l'indice 4…
- E quando si dovrà interrompere il ciclo?

```
i = 0
j = length / 2
While (i < length / 2)
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

```
i = 0
j =  size/2
While (  size/2  )
    Swap elements at positions i and j
    i = i + 1
    j = j + 1
```

swaphalves.py

- Aprire il file swaphalves.py

- Analizzare la soluzione proposta
  - Funziona sia nel caso di liste di lunghezza pari che di lunghezza dispari?
- Si può migliorare/semplificare?
  - La funzione swap è necessaria?
  - Si potrebbero sfruttare le «porzioni» (slice) per semplificare il lavoro?

Tabelle

6.7

Tabelle

- Le liste possono essere usate per contenere dati in due dimensioni (2D), come in un foglio di calcolo
  - Righe e colonne
  - Anche detto ‘matrice’

	Oro	Argento	Bronzo
Canada	0	3	0
Italia	0	0	1
Germania	0	0	1
Giappone	1	0	0
Kazakistan	0	0	1
Russia	3	1	1
Corea del Sud	0	1	0
Stati Uniti d’America	1	0	1

Figura 11
Numero di medaglie vinte nel pattinaggio artistico

Creare tabelle

- Questo codice crea una tabella che contiene 8 righe e 3 colonne, quindi è adatto a contenere i dati relativi al conteggio delle medaglie

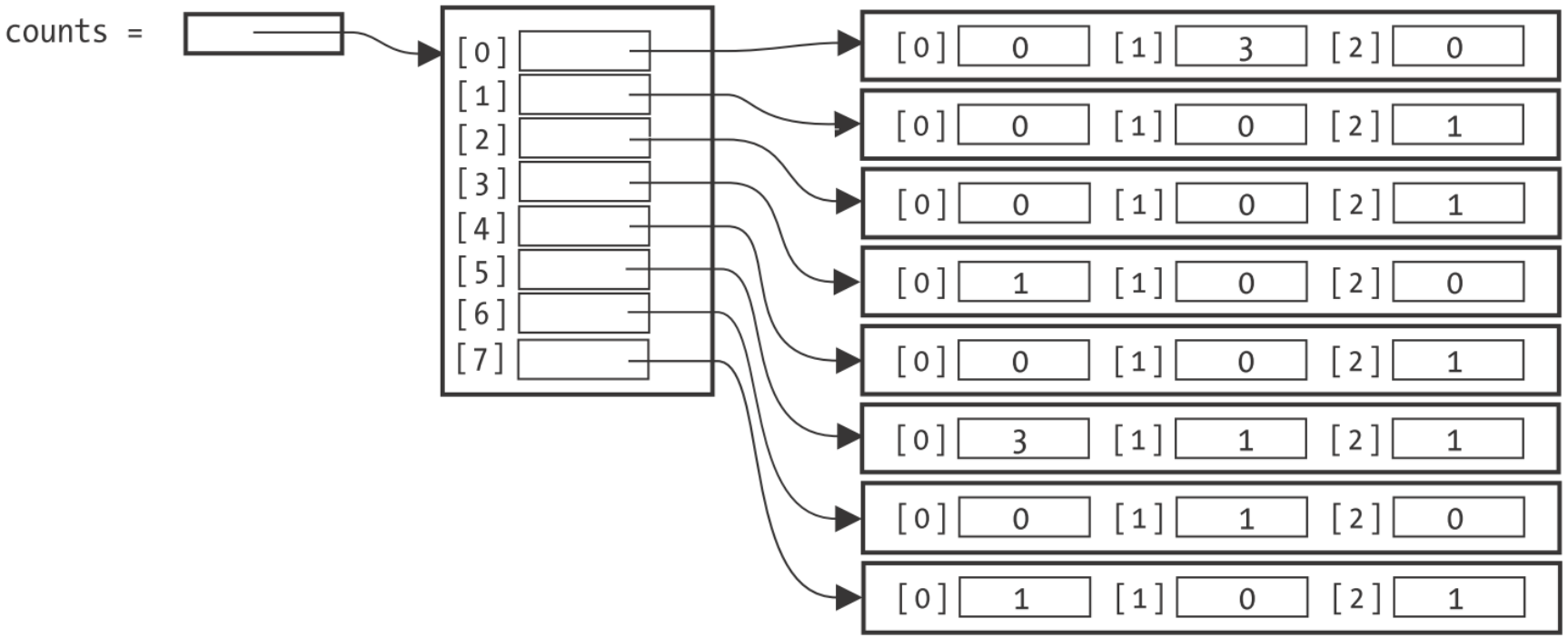
```
COUNTRIES = 8
MEDALS = 3

counts = [
        [ 0, 3, 0 ],
        [ 0, 0, 1 ],
        [ 0, 0, 1 ],
        [ 1, 0, 0 ],
        [ 0, 0, 1 ],
        [ 3, 1, 1 ],
        [ 0, 1, 0 ],
        [ 1, 0, 1 ]
    ]
```

Provalo con PythonTutor!

Creare tabelle (2)

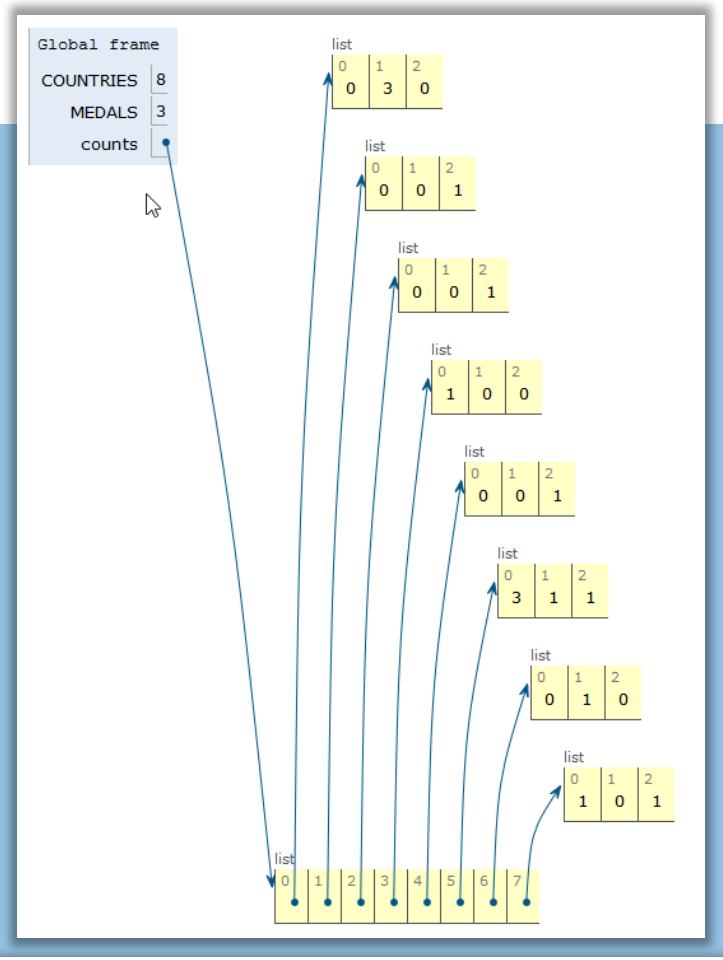
- Creare una lista in cui ogni elemento sia esso stesso una lista:

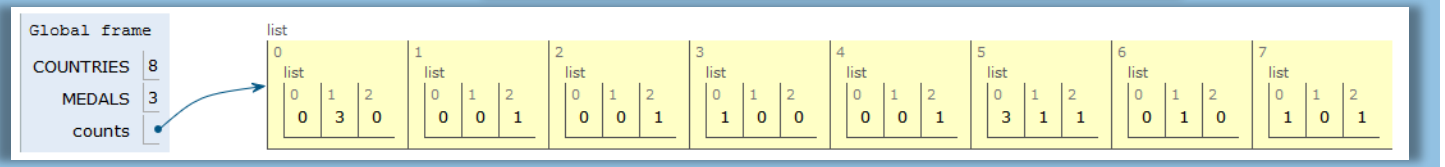


Cosa accade realmente

```
COUNTRIES = 8
MEDALS = 3

counts = [
    [ 0, 3, 0 ],
    [ 0, 0, 1 ],
    [ 0, 0, 1 ],
    [ 1, 0, 0 ],
    [ 0, 0, 1 ],
    [ 3, 1, 1 ],
    [ 0, 1, 0 ],
    [ 1, 0, 1 ]
    ]
```





Creare tabelle (3)

- A volte, c'è bisogno di creare tabelle con dimensioni troppo grandi da poter inizializzare con una costante esplicita (*literal value*)
- In questo caso, occorre costruire la tabella in modo incrementale
- Prima di tutto, si crea una lista che verrà usata per contenere le righe

```
table = []
```

Creare tabelle (4)

- Poi si crea una nuova lista per ogni riga della tabella, che rappresenta le colonne di quella riga. Si può creare replicando un valore (tante volte quante sono le colonne), poi la si aggiunge alla lista di righe:

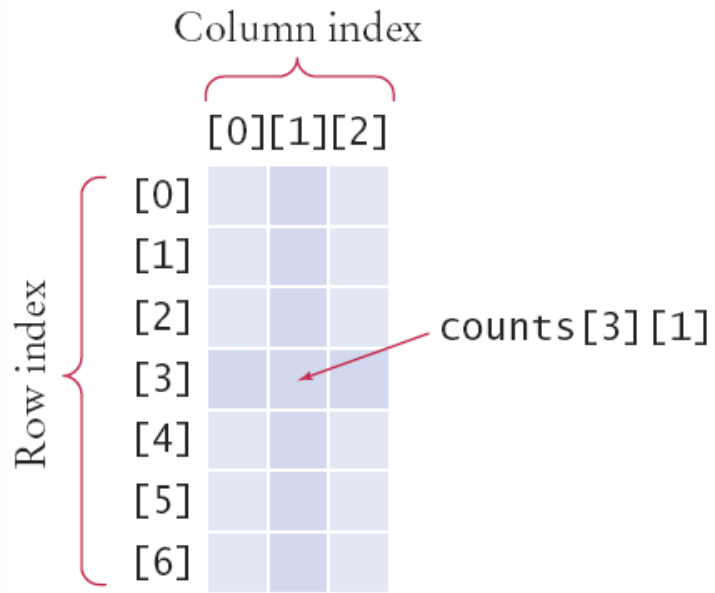
```
ROWS = 5
COLUMNS = 20
for i in range(ROWS) :
    row = [0] * COLUMNS
    table.append(row)
```

- Il risultato è una tabella formata da 5 righe e 20 colonne

Accedere agli elementi

- Si usano due indici:
  - prima l'indice di riga, poi di colonna

```
medalCount = counts[3][1]
```



- Per stampare
  - Usare il ciclo `for` annidato
  - Ciclo esterno sulle righe (`i`), ciclo interno sulle colonne (`j`)

```
for i in range(COUNTRIES):
    # processa l'i-esima riga
    for j in range(MEDALS) :
        # processa la j-esima Colonna della i-esima riga
        print("%8d" % counts[i][j], end="")
    print() # va a capo alla fine della riga
```

```
       0       3       0
       0       0       1
       0       0       1
       1       0       0
       0       0       1
       3       1       1
       0       1       0
       1       0       1
```

Trovare elementi vicini

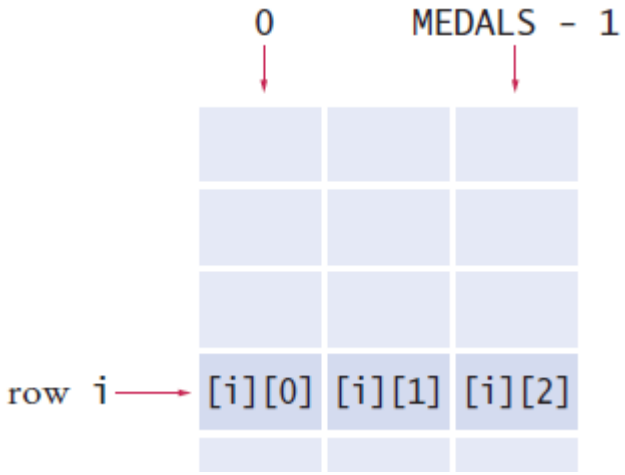
- Alcuni programmi che lavorano con liste bidimensionali necessitano di trovare gli elementi che sono adiacenti ad un elemento dato
- Questa caratteristica è particolarmente utile nei giochi
- Partendo dalla posizione `i`, `j`
- Fare attenzione ai bordi!
  - No indici negativi!
  - No 'sconfinamenti'

`[i - 1][j - 1]`	`[i - 1][j]`	`[i - 1][j + 1]`
`[i][j - 1]`	`[i][j]`	`[i][j + 1]`
`[i + 1][j - 1]`	`[i + 1][j]`	`[i + 1][j + 1]`

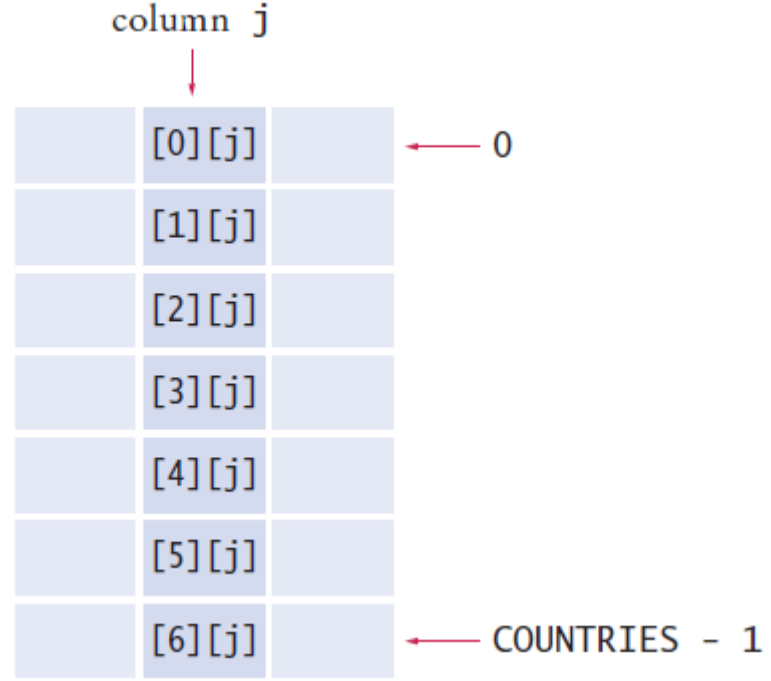
Sommare per righe e per colonne

Somma della riga i)

```
rowtotal = 0
for j in range(MEDALS):
    rowtotal = rowtotal + counts[i][j]
```



Somma della colonna j




```
coltotal = 0
for i in range(COUNTRIES):
    coltotal = coltotal + counts[i][j]
```

Usare tabelle con funzioni

- Passando una tabella ad una funzione, è necessario estrarre le dimensioni della tabella. Data una tabella chiamata `values`:
  - `len(values)` è il numero delle righe
  - `len(values[0])` è il numero di colonne
- Per esempio, la seguente funzione elabora la somma di tutti gli elementi di una tabella

```
def sum(values) :
    total = 0
    for i in range(len(values)) :
        for j in range(len(values[0])) :
            total = total + values[i][j]
return total
```

Esempio

- Aprire il file medals.py
- Analizzare la soluzione proposta

Sommario

Sommario: liste

- Una lista è un contenitore per sequenze di valori
- Ogni elemento in un lista è accessibile attraverso un indice intero `i`, usando la sintassi `list[i]`
- L'indice di una lista deve essere maggiore di zero e minore del numero di elementi della lista
- Un errore di out-of-range si presenta quando si fornisce un indice non valido, questo potrebbe causare l'interruzione del programma
- È possibile iterare sul valore dell'indice o sugli elementi di una lista

Sommario: liste

- Il riferimento di una lista specifica la locazione di essa. Copiando il riferimento verrà prodotto un secondo riferimento alla stessa lista
- Una ricerca lineare ispeziona gli elementi in sequenza fino a trovare la corrispondenza cercata
- Usare una variabile temporanea quando si vogliono scambiare due elementi
- Le liste possono essere usate come parametri di funzioni ed essere restituite

Sommario: liste

- Chiamando una funzione che ha come parametro una lista, la funzione riceverà un riferimento alla lista, non una sua copia
- Una tupla si crea come una sequenza separata da virgole e racchiusa fra parentesi
- Combinando algoritmi fondamentali, è possibile risolvere problemi di programmazione complessi
- È quindi necessario acquisire familiarità con l'uso degli algoritmi fondamentali per poterli adeguatamente adattare
- Per scoprire algoritmi può essere utile la manipolazione di oggetti fisici

Sommario: liste

- Per contenere dati tabulari si usano liste bidimensionali
- A ciascun elemento di una lista bidimensionale si può accedere usando due indici, `table[i][j]`