

Fondamenti di programmazione  
Canale A-L - Prof. Sterbini  
AA 25-26  
Lezione 4



## RECAP lezione 3

- assegnamenti potenziati
- assegnamenti multipli
- if-then-else e match-case
- cicli for e while
- range
- cenni sui contenitori

FDPLEZ4



# I contenitori: liste, tuple, dizionari, insiemi

|| tipo | parentesi | esempio | che fa | indicizzata? | modificabile? :-|:-|:-|:-|:-|:-|:  
quadre | [1, 2, 3, 4] | lista di elementi eterogenei | indicizzata | modificabile **tupla** | tuple | ()  
tonde | (1, 2, 3, 4) | n-upla di elementi eterogeni | indicizzata | NON  
modificabile **insieme** | set | {}  
graffe | {1, 2, 3} | insieme di elementi eterogenei immutabili  
senza ripetizioni | NON indicizzato  
e in ordine non fisso | modificabile **dizionario** | dict | {}  
graffe | {'a': 1, 'b': 2} | coppie  
chiave unica : valore | indicizzato sulle chiavi | modificabile

In [1]

```
# Esempio: costruisco una lista
lista_valori = [2, 5, 7, 23, 45, 2, 7, 23, 'tre' ]
lista_valori
```

Out[1]

```
[2, 5, 7, 23, 45, 2, 7, 23, 'tre']
```

In [2]

```
# costruisco una tupla con gli elementi della lista
tupla_valori = tuple(lista_valori)
tupla_valori
```

Out[2]

```
(2, 5, 7, 23, 45, 2, 7, 23, 'tre')
```

In [3]

```
# costruisco un insieme con gli elementi della lista
set_valori = set(lista_valori)
set_valori # solo elementi unici in ordine non fisso
```

Out[3]

```
{2, 23, 45, 5, 7, 'tre'}
```

```
In [4...]: # costruisco un dizionario chiave (unica) -> valore  
dizionario = { 'a': 1, 'b': 2, 'c': 3, 45: 17, (2,5):'paperino', 'c': 'Minnie'}  
dizionario # le chiavi sono uniche ed appare l'ultima se ci sono doppioni
```

```
Out[4...]: {'a': 1, 'b': 2, 'c': 'Minnie', 45: 17, (2, 5): 'paperino'}
```

```
In [5...]: ## le liste sono indicizzate E **modificabili**  
print(lista_valori[4]) # ne stampo uno (in posizione 4)  
lista_valori[5] = 666 # cambio un elemento in posizione 5  
print(lista_valori) # la lista è cambiata
```

```
45  
[2, 5, 7, 23, 45, 666, 7, 23, 'tre']
```

```
In [6...]: ## gli insiemi NON sono indicizzati e sono **modificabili**  
print(set_valori)  
print(set_valori.pop()) # estraggo un elemento (a caso) con 'pop'  
  
{2, 5, 7, 45, 23, 'tre'}  
2
```

```
In [7...]: set_valori.add(666) # aggiungo un elemento  
print(set_valori)  
  
{5, 7, 45, 23, 666, 'tre'}
```

```
In [8...]: # leggo l'elemento in posizione 4  
set_valori[4] # ERRORE!!! gli insiemi NON sono indicizzati!
```

```
-----  
TypeError Traceback (most recent call last)  
Cell In[8], line 2  
      1 # leggo l'elemento in posizione 4  
----> 2 set_valori[4] # ERRORE!!! gli insiemi NON sono indicizzati!
```

```
TypeError: 'set' object is not subscriptable
```

```
In [9...]: ## le tuple sono indicizzate MA **immutabili**  
print(tupla_valori[4]) # ne leggo un valore  
tupla_valori[5] = 666 ### ma se provo a modificarla ERRORE!
```

```

TypeError                                     Traceback (most recent call last)
Cell In[9], line 3
      1 ## le tuple sono indicizzate MA **immutabili**
      2 print(tupla_valori[4]) # ne leggo un valore
----> 3 tupla_valori[5] = 666    ### ma se provo a modificarla ERRORE!
                                         |          |
                                         +-----+
                                         TypeError: 'tuple' object does not support item assignment

```

```
In [..]: # i dizionari sono indicizzati dalle chiavi e **modificabili**
print(dizionario)
print(dizionario.keys())           # estraggo le chiavi -> generatore
print(list(dizionario.keys()))     # estraggo le chiavi -> lista
```

```
In [..]: print(dizionario['a'])      # stampo il valore associato alla chiave 'a'
dizionario['paperino'] = 42 # aggiungo una coppia chiave -> valore
dizionario[(2, 5)] = 3.14 # modifco una coppia chiave -> valore
del dizionario['c']          # elimino la coppia con chiave 'c'
print(dizionario)            # il dizionario è cambiato
```

```
In [..]: dizionario['pluto']       # se la chiave non c'è ERRORE!!!
```

```
In [..]: # posso controllare se la chiave è presente con 'in'
print('pluto' in dizionario)
print(list(dizionario.values())) # e posso estrarre i soli valori
```

```
In [..]: # Che succede se metto due volte la stessa chiave 'uno'?
print({ 'uno':1, 'due':2, 'uno':11 })
# appare solo l'ultimo valore perchè le chiavi sono UNICHE
print(dict([('a',3),(17,'paperino'), ('a',33)])) # conversione da lista di coppie a dict
```

```
In [..]: # come stampare gli elementi dei diversi contenitori?
print(lista_valori)
for elemento in lista_valori:
    print(elemento, end=' ') # stampa seguita da spazio invece che '\n'
print('\t\tlista_valori')
```

```
In [ ... ]: print(set_valori)
    for elemento in set_valori:
        print(elemento, end=' ')
print('\t\tset_valori')
```

```
In [ ... ]: print(tupla_valori)
    for elemento in tupla_valori:
        print(elemento, end=' ')
print('\t\ttupla_valori')
```

```
In [ ... ]: # Python permette di eseguire più assegnamenti assieme
# e il metodo items dei dizionari produce ciascuna delle coppie (chiave, valore)
print(list(dizionario.items()))
for chiave,elemento in dizionario.items():
    print(chiave,elemento)
print(dizionario.items())      # produce un oggetto 'generatore' di coppie
print(list(dizionario.items())) # con list creo una lista che li contiene
```

## Scansione per valori e per indice

- posso direttamente scandire i **valori** e ignorare la posizione
- oppure scandire **gli indici** ed estrarre ciascun elemento
- oppure scandire gli elementi contando le posizioni con **enumerate**
  - CONSIGLIATO se vi serve sia il valore che la posizione

```
In [ ... ]: ##### come scandire un contenitore per valori
for elemento in lista_valori:
    print(elemento, end=' ')
print()
```

```
In [1...]: ##### come scandire un contenitore per indice
for indice in range(len(lista_valori)): # genero gli indici
    elemento = lista_valori[indice] # estraggo l'elemento in posizione 'indice'
    print(indice, elemento) # e stampo posizione ed elemento
```

```
0 2  
1 5  
2 7  
3 23  
4 45  
5 666  
6 7  
7 23  
8 tre
```

```
In [1]: # la funzione 'enumerate' estraе gli elementi E conta quanti ne ha estratti  
# per ciascun elemento estratto torna la coppia (indice, elemento)  
for conteggio, valore in enumerate(set_valori):  
    print(conteggio, valore)  
  
print(set_valori)  
list(enumerate(set_valori))      # uso list perch  enumerate   un generatore
```

```
0 5  
1 7  
2 45  
3 23  
4 666  
5 tre  
{5, 7, 45, 23, 666, 'tre'}
```

```
Out[1]: [(0, 5), (1, 7), (2, 45), (3, 23), (4, 666), (5, 'tre')]
```

## E' "PROIBITO" modificare la lista MENTRE ci faccio un ciclo?

What if elimino elementi dalla lista MENTRE la sto scorrendo?

- sono nella posizione 3,

```
0 1 2 3 4 5 6 7 8  
^  
3
```

- elimino l'elemento in pos 3
- l'elemento in posizione 4 si è spostato in posizione 3

```
0 1 2 4 5 6 7 8  
  ^  
  3
```

- il ciclo passa all'indice seguente 4

```
0 1 2 4 5 6 7 8  
  ^  
  4
```

- e **MI PERDO IL VALORE CHE ERA IN POSIZIONE 4 e ora è in 3!!!**
- (e inoltre alla fine **HO ERRORE SULL'ULTIMO ELEMENTO!!!**) perchè ora la lista ha un elemento in meno ma io continuo a contare fino alla lunghezza originale

Insomma, mi tiro via il tappeto da sotto i piedi!!!

```
In [1]: lista_interi = [ 0, 11, 22, 33, 44, 55, 66, 77, 88, ]  
for i in range(len(lista_interi)): # scorro la lista generando gli indici  
    print(lista_interi[i])          # stampo l'elemento all'indice corrente  
    if i == 3:                      # se sono all'indice 3  
        del lista_interi[i]         # elimino l'elemento in posizione 3  
        print('ho eliminato il 33 dalla lista e ho saltato il 44')  
# ho eliminato un elemento, l'indice di fine lista sborda dalla lista accorciata
```

```
0  
11  
22  
33  
ho eliminato il 33 dalla lista e ho saltato il 44  
55  
66  
77  
88
```

```
IndexError
```

```
Cell In[12], line 3
```

```
1 lista_interi = [ 0, 11, 22, 33, 44, 55, 66, 77, 88, ]  
2 for i in range(len(lista_interi)):  
----> 3     print(lista_interi[i])  
4     if i == 3:  
5         del lista_interi[i]
```

```
Traceback (most recent call last)
```

```
IndexError: list index out of range
```

```
In [2... lista_interi      # manca il 33
```

```
Out[2... [0, 11, 22, 44, 55, 66, 77, 88]
```

## COME RISOLVERE?

- **opzione 1:** scandisco la lista dalla fine all'inizio
  - in questo modo le modifiche spostano elementi seguenti GIA' ESAMINATI
  - ed inoltre gli indici esistono tutti (no IndexError)
- **oppure:** (MEGLIO) costruisco una nuova lista
  - in questo modo non modifico la lista originale
- **oppure:** scandisco una sua copia che rimane intoccata (non ovvio)

```
In [2... # Esempio: scansione a rovescio eliminando un elemento in posizione 3
```

```

lista_interi = [ 0, 11, 22, 33, 44, 55, 66, 77, 88, ]
incremento = -1 # mi muovo per indici decrescenti
inizio = len(lista_interi)-1 # parto dall'ultimo
fine = -1 # mi fermo a zero (-1 non viene raggiunto)
RANGE = range(inizio, fine, incremento)
for i in RANGE: # range con incremento negativo
    print(lista_interi[i], end=' ') # stampo l'elemento (senza andare a capo)
    if i == 3: # quando sono sull'indice 3
        del lista_interi[i] # elimino l'elemento
print('\t(ho visitato tutte le posizioni in ordine opposto)')
print(lista_interi, '\tlista_interi è cambiata e non contiene più il 44')

```

```

88 77 66 55 44 33 22 11 0      (ho visitato tutte le posizioni in ordine opposto)
[0, 11, 22, 44, 55, 66, 77, 88]      lista_interi è cambiata e non contiene più il 44

```

In [2...]

```

# oppure costruisco una nuova lista e ci copio solo gli elementi "giusti"
nuova_lista = []
lista_interi = [ 0, 11, 22, 33, 44, 55, 66, 77, 88, ]
for i in range(len(lista_interi)):
    print(lista_interi[i], end=' ')
    if i != 3: # rovescio il test perchè voglio copiare tutti TRANNE il 3°
        nuova_lista.append(lista_interi[i])

print('\t(ho visitato tutte le posizioni in ordine normale)')
print(lista_interi, 'la lista_interi rimane invariata')
print(nuova_lista, '\tla nuova_lista è senza il 44')

```

```

0 11 22 33 44 55 66 77 88      (ho visitato tutte le posizioni in ordine normale)
[0, 11, 22, 33, 44, 55, 66, 77, 88] la lista_interi rimane invariata
[0, 11, 22, 44, 55, 66, 77, 88]      la nuova_lista è senza il 44

```

## Scorciatoie logiche per controllare i contenitori nei test if/then/else

Spesso dobbiamo controllare se un contenitore è vuoto o contiene elementi (oppure se una variabile è 0 o diversa da 0). Per semplificare gli if-then-else:

- un contenitore vuoto vale come **False**
- un contenitore con almeno un elemento vale **True**

Ed inoltre

- un valore 0 vale False
- un valore diverso da zero vale True

In [2...]:

```
## Per controllarlo trasformiamo contenitori vuoti in booleani
bool([]), bool(tuple()), bool({}), bool(set())
```

Out[2...]:

```
(False, False, False, False)
```

In [2...]:

```
## Per controllarlo trasformiamo contenitori con almeno un elemento in booleani
bool([1,2]), bool((2,)), bool({3:'tre','s':45}), bool({4,45})
type(tuple([])), tuple([]), 4

S = [20.3]
if len(S)>0:    # if S:
    print(S, 'contiene almeno un elemento')
else:
    print(S, 'non contiene elementi')
```

[20.3] contiene almeno un elemento

## Tutti i dati in Python sono "oggetti"

- un oggetto è un **gruppo di informazioni coerenti** (`attributi`)  
Es. un **cane** ha: nome, altezza, razza, genere, peso, colore, animazione, verso, ...  
Es. una **str** ha: lunghezza, sequenza di caratteri, ...
- assieme a tutte le **operazioni che potete eseguire** (`metodi`)  
Es. un **cane**: abbaia, scodinzola, cammina, corre, salta  
Es. una **str** può: join, split, upper, lower, startswith, ...

int, str, bool, float, etc... sono oggetti

per scoprirne le caratteristiche usiamo help oppure ctrl-I in Spyder oppure . e l'autocompletamento

In [3...]

```
str          # testo
int          # interi           # abs, mod, %, //
float        # numeri con virgola
complex      # numeri complessi   # conjugate, imag, real, ...
bool         # booleani (True, False) # and, or, not

# carichiamo nel programma la classe Fraction dalla libreria fractions
from fractions import Fraction    # Fraction implementa le frazioni intere
#help(Fraction)
f = Fraction(124,6)              # richiede due valori: denominatore, numeratore
f, float(f), int(f), str(f)     # notate la semplificazione!
```

Out[3...]

```
(Fraction(62, 3), 20.666666666666668, 20, '62/3')
```

I metodi sono le operazioni che possiamo compiere su un oggetto

e che ne possono manipolare il contenuto o creare nuovi oggetti

Es. le operazioni sulle stringhe sono i metodi della classe `str`

- join, split, lower, upper, find, ...

Per eseguire un metodo di un oggetto

si usa la sintassi `oggetto.nomedelmetodo(argomenti)`

In [3...]

```
# Esempi sulle stringhe
S = 'Paperino andò\t al, mare\n a      nuotare'
S
```

Out[3...]

```
'Paperino andò\t al, mare\n a      nuotare'
```

In [3...]

```
# split (e varianti) spezza la stringa su un carattere/stringa separatore
L = S.split()                  # per default sulle sequenze di spazi/tab/accapo
print('frammenti: ', L)
print('righe    : ', S.split('\n')) # spezzo in righe usando '\n' come separatore
```

```
frammenti:  ['Paperino', 'andò', 'al,', 'mare', 'a', 'nuotare']
righe     :  ['Paperino andò\t al, mare', ' a    nuotare']
```

```
In [3...]: # operazione opposta
# separatore.join(lista_di_stringhe) unisce una sequenza di stringhe
# interponendo un separatore
' | '.join({'a','b','c'})
```

```
Out[3...]: 'a | b | c'
```

```
In [1...]: # come spezzare una stringa in caratteri o in un insieme di caratteri unici
list('lkruhgaljhrl'), set('lkruhgaljhrl')
```

```
Out[1...]: ([{'l', 'k', 'r', 'u', 'h', 'g', 'a', 'l', 'j', 'h', 'r', 'l'},
 {'a', 'g', 'h', 'j', 'k', 'l', 'r', 'u'}])
```

```
In [3...]: # lower genera una copia di S tutta in minuscole
S.lower()
```

```
Out[3...]: 'paperino andò\t al, mare\n a    nuotare'
```

```
In [3...]: # islower, isupper, isalpha, isnumeric sono dei test True/False
print('S è tutta minuscola?', S.islower())
```

```
S è tutta minuscola? False
```

```
In [3...]: # genero la copia di tutte maiuscole
S.upper()
```

```
Out[3...]: 'PAPERINO ANDÒ\t AL, MARE\n A    NUOTARE'
```

```
In [3...]: # find    cerco la posizione delle parole 'mare' e 'Gnomo'
S.find('mare'), S.find('Gnomo')
# e ottengo -1 perchè 'Gnomo' non è presente
```

```
Out[3...]: (19, -1)
```

**REMINDER:** le **stringhe sono IMMUTABILI**, ogni volta che ci fate operazioni sopra ne create una nuova

## Curiosità tecnica: pool di numeri e stringhe unici

Python, per rendere più efficiente e veloce l'uso della memoria, invece che duplicarla, tiene copie uniche:

- dei **numeri piccoli** nell'intervallo [-5, 256]
- e delle **stringhe corte senza spazi** (che sono ad esempio usate nel codice come **nomi** di variabili e funzioni)

In [3...]

```
B = 93
C = 92+1
print(id(B), id(C))
print("Sono identici in memoria?", B is C)
```

4302272312 4302272312

Sono identici in memoria? True

In [4...]

```
# per le stringhe l'ottimizzazione è applicata alle 'parole'
# che non contengono spazi (particolarmente importante nella
# ottimizzazione del codice in memoria)
S1 = 'giovanni' * 50
S2 = 'giovanni' * 50
print(id(S1), id(S2))
print("Sono identici in memoria?", S1 is S2)
```

4405747184 4405747184

Sono identici in memoria? True

In [4...]

```
# QUANDO C'E' LO SPAZIO!!!
S1 = 'giovanni ' * 50
S2 = 'giovanni ' * 50
print(id(S1), id(S2))
print("Sono identici in memoria?", S1 is S2)
```

4405761904 4405759920

Sono identici in memoria? False

In [4...]

```
# esempio di ottimizzazione dei piccoli numeri (usando esponente piccolo)
esponente = 1
A = 34**esponente
```

```
B = (30+4)**esponente
print("Sono uguali ?", A == B)
print("Sono identici?", A is B)
```

Sono uguali ? True  
Sono identici? True

In [4...]: # esempio di ottimizzazione dei piccoli numeri (usando esponente grande)

```
esponente = 3
A = 34**esponente
B = (30+4)**esponente
print("Sono uguali ?", A == B)
print("Sono identici?", A is B)
```

Sono uguali ? True  
Sono identici? False

## Metodi dei Contenitori

### Operazioni sulle liste (list)

- **L[indice]** legge il valore di L all'indice `indice` e dà errore se l'indice non esiste
- **L[indice] = espressione** sostituisce il valore all'indice `indice` e dà errore se l'indice non esiste
- **elemento in L** True se l'elemento è presente in L
- **L1 + L2** nuova lista concatenazione di L1 e L2
- **L1 \* N** nuova lista come replicazione N volte e concatenazione
- **L.append(elemento)** aggiunta di un elemento alla fine

In [4...]:

```
# Esempi di operazioni sulle liste
L1 = [ 1, 2, 3, 4, 5, ]
L2 = [ 11, 22, 33, ]
print('concateno L1 ed L2', L1 + L2)          # nuova lista dalla concatenazione di L1 e L2
print('replico L1 5 volte', L1 * 5)            # nuova lista dalla ripetizione di L1
```

concateno L1 ed L2 [1, 2, 3, 4, 5, 11, 22, 33]
replico L1 5 volte [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

```
In [4...]: print(L1.pop(2))      # elimino quello a indice 2  
L1.insert(78, 55)        # inserisco ad indice 78 il valore 55 (va alla fine)  
L1.insert(-78, 333)      # inserisco ad indice -78 il valore 333 (va all'inizio)  
L1
```

3

```
Out[4...]: [333, 1, 2, 4, 5, 55]
```

## Assegnazione a slice

Oltre alla modifica di singoli elementi

```
lista[indice] = nuovo_valore
```

Possiamo sostituire un GRUPPO di elementi (una SLICE) con un diverso gruppo (un contenitore)

```
lista[inizio:fine]          = contenitore    # SOLO SE CONSECUTIVI: con un numero qualsiasi di  
elementi (anche 0)  
lista[inizio:fine:incremento] = contenitore   # SE NON CONSECUTIVI: con lo stesso numero di  
elementi!!!
```

```
In [4...]: XXX = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
XXX[2:8:1] = list('abcdefg')                 # se assegno una slice di elementi CONSECUTIVI  
XXX
```

```
Out[4...]: [1, 2, 'a', 'b', 'c', 'd', 'e', 'f', 'g', 9, 10]
```

```
In [4...]: XXX = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
XXX[2:8:2] = list('abcdefg')                 # se assegno una slice slice di elementi NON CONSECUTIVI  
XXX
```

```
ValueError
Cell In[47], line 2
    1 XXX = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
----> 2 XXX[2:8:2] = list('abcdefg')           # se assegno una slice slice di elementi NON CONSECUTI
VI
    3 XXX

ValueError: attempt to assign sequence of size 7 to extended slice of size 3
```

```
In [4...]: L2[-2:] = []      # elimino gli ultimi due elementi con assegnazione a slice
L2
```

```
Out[4...]: [11]
```

```
In [4...]: # POSSO sostituire distruttivamente TUTTO il CONTENUTO di L1 con L2
# con un ASSEGNAZIONE A SLICE
L1[:] = L2
L1
```

```
Out[4...]: [11]
```

```
In [5...]: L1 = [1, 2, 3, 4, 5]
S1 = {'99', '55'}
D1 = {44:12, 55:13, 66:34, 77:35}
L1[1::2] = S1  # ci metto gli elementi di un set!!!
L1[4:5] = D1  # di un dizionario mette solo le chiavi (se voglio i valori uso D1.values())
L1
```

```
Out[5...]: [1, '99', 3, '55', 44, 55, 66, 77]
```

## Altre operazioni sulle liste (list)

- `L.pop()` estrazione distruttiva dell'ultimo elemento (ERRORE se vuota)
- `L.pop(i)` estrazione distruttiva dell'elemento all'indice `i` (ERRORE se `i` non esiste)
- `L.insert(i, elemento)` inserisce l'elemento all'indice `i` (o agli estremi se sbordo)
- `L.remove(elemento)` elimina la **prima occorrenza** dell'elemento (ERR se non presente)

- `L.index(elemento)` trova il primo indice in cui c'è l'elemento (ERR se non presente)
- `L.count(elemento)` conta l'elemento
- `L.reverse()` rovesciamento distruttivo della lista
- `L.sort()` ordinamento distruttivo della lista

```
In [5...]: # Altri esempi di operazioni sulle liste  
L = [10, 43, 92, 1, 3, -43, 90, -200, 12, 3]  
L.count(3)
```

```
Out[5...]: 2
```

```
In [5...]: L.index(-43)
```

```
Out[5...]: 5
```

```
In [5...]: L.sort()      # riordinamento DISTRUTTIVO (modifica la lista)  
L
```

```
Out[5...]: [-200, -43, 1, 3, 3, 10, 12, 43, 90, 92]
```

```
In [5...]: help(list)
```

Help on class list in module builtins:

```
class list(object)
| list(iterable=(), /)
|
| Built-in mutable sequence.
|
| If no argument is given, the constructor creates a new empty list.
| The argument must be an iterable if specified.
|
| Methods defined here:
|
| __add__(self, value, /)
|     Return self+value.
|
| __contains__(self, key, /)
|     Return bool(key in self).
|
| __delitem__(self, key, /)
|     Delete self[key].
|
| __eq__(self, value, /)
|     Return self==value.
|
| __ge__(self, value, /)
|     Return self>=value.
|
| __getattribute__(self, name, /)
|     Return getattr(self, name).
|
| __getitem__(self, index, /)
|     Return self[index].
|
| __gt__(self, value, /)
|     Return self>value.
|
| __iadd__(self, value, /)
```

Implement self+=value.

`__imul__(self, value, /)`  
Implement self\*=value.

`__init__(self, /, *args, **kwargs)`  
Initialize self. See help(type(self)) for accurate signature.

`__iter__(self, /)`  
Implement iter(self).

`__le__(self, value, /)`  
Return self<=value.

`__len__(self, /)`  
Return len(self).

`__lt__(self, value, /)`  
Return self<value.

`__mul__(self, value, /)`  
Return self\*value.

`__ne__(self, value, /)`  
Return self!=value.

`__repr__(self, /)`  
Return repr(self).

`__reversed__(self, /)`  
Return a reverse iterator over the list.

`__rmul__(self, value, /)`  
Return value\*self.

`__setitem__(self, key, value, /)`  
Set self[key] to value.

```
| __sizeof__(self, /)
|     Return the size of the list in memory, in bytes.

| append(self, object, /)
|     Append object to the end of the list.

| clear(self, /)
|     Remove all items from list.

| copy(self, /)
|     Return a shallow copy of the list.

| count(self, value, /)
|     Return number of occurrences of value.

| extend(self, iterable, /)
|     Extend list by appending elements from the iterable.

| index(self, value, start=0, stop=9223372036854775807, /)
|     Return first index of value.

|         Raises ValueError if the value is not present.

| insert(self, index, object, /)
|     Insert object before index.

| pop(self, index=-1, /)
|     Remove and return item at index (default last).

|         Raises IndexError if list is empty or index is out of range.

| remove(self, value, /)
|     Remove first occurrence of value.

|         Raises ValueError if the value is not present.
```

```
| reverse(self, /)
|     Reverse *IN PLACE*.

sort(self, /, *, key=None, reverse=False)
    Sort the list in ascending order and return None.
```

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

---

Class methods defined here:

```
| __class_getitem__(...)
|     See PEP 585
```

---

Static methods defined here:

```
| __new__(*args, **kwargs)
|     Create and return a new object. See help(type) for accurate signature.
```

---

Data and other attributes defined here:

```
| __hash__ = None
```

## Operazioni sulle tuple

- **T[i]** accesso all'elemento all'indice i (**SOLO lettura!**)
- **elemento in T** True se l'elemento è presente in T
- **T1 + T2** nuova tupla concatenazione di T1 e T2

- `T * N` nuova tupla come replicazione N volte e concatenazione
- `L.index(elemento)` trova il primo indice in cui c'è l'elemento cercato (ERR se non presente)
- `L.count(elemento)` conta l'elemento

In [5...]

```
# Esempi di operazioni sulle tuple
```

```
T1 = 1, 2, 3, 4
T2 = 24, 52, 67, 31
print('T1 + T2 =', T1 + T2)
print('T1 * 5 =', T1*5)
```

```
T1 + T2 = (1, 2, 3, 4, 24, 52, 67, 31)
```

```
T1 * 5 = (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

In [5...]

```
T1[2] = 55 # ERRORE
```

```
-----
```

```
TypeError
Cell In[56], line 1
----> 1 T1[2] = 55 # ERRORE
```

```
Traceback (most recent call last)
```

```
TypeError: 'tuple' object does not support item assignment
```

In [5...]

```
T1[1:3] # la slice di una tupla produce una tupla
```

```
# ovviamente non esistono assegnamenti a slice
```

Out[5...]

```
(2, 3)
```

In [5...]

```
help(tuple)
```

Help on class tuple in module builtins:

```
class tuple(object)
| tuple(iterable=(), /)
|
| Built-in immutable sequence.
|
| If no argument is given, the constructor returns an empty tuple.
| If iterable is specified the tuple is initialized from iterable's items.
|
| If the argument is a tuple, the return value is the same object.
|
| Built-in subclasses:
|     asyncgen_hooks
|     UnraisableHookArgs
|
| Methods defined here:
|
| __add__(self, value, /)
|     Return self+value.
|
| __contains__(self, key, /)
|     Return bool(key in self).
|
| __eq__(self, value, /)
|     Return self==value.
|
| __ge__(self, value, /)
|     Return self>=value.
|
| __getattribute__(self, name, /)
|     Return getattr(self, name).
|
| __getitem__(self, key, /)
|     Return self[key].
|
| __getnewargs__(self, /)
```

```
| __gt__(self, value, /)
|     Return self>value.

| __hash__(self, /)
|     Return hash(self).

| __iter__(self, /)
|     Implement iter(self).

| __le__(self, value, /)
|     Return self<=value.

| __len__(self, /)
|     Return len(self).

| __lt__(self, value, /)
|     Return self<value.

| __mul__(self, value, /)
|     Return self*value.

| __ne__(self, value, /)
|     Return self!=value.

| __repr__(self, /)
|     Return repr(self).

| __rmul__(self, value, /)
|     Return value*self.

| count(self, value, /)
|     Return number of occurrences of value.

| index(self, value, start=0, stop=9223372036854775807, /)
|     Return first index of value.
```

Raises `ValueError` if the value is not present.

Class methods defined here:

`__class_getitem__(...)`  
See PEP 585

Static methods defined here:

`__new__(*args, **kwargs)`  
Create and return a new object. See `help(type)` for accurate signature.

## Operazioni sugli insiemi (set)

- `elemento in S` True se elemento è presente in S
- `S1 | S2` oppure `S1.union(S2)` unione (or, tutti gli elementi delle due)
- `S1 & S2` oppure `S1.intersection(S2)` intersezione (and, solo gli elementi in comune)
- `S1 - S2` oppure `S1.difference(S2)` insieme degli elementi di S1 che non sono in S2
- `S1 ^ S2` oppure `S1.symmetric_difference(S2)` insieme degli elementi NON in comune (xor)
  - ci sono anche gli assegnamenti potenziati `|=`, `&=`, `-=` e `^=`

In [6...]

```
# Esempi di operazioni sugli insiemi
N = 20
S1 = set(range(N))           # insiemi dei numeri [0..N)
S3 = set(range(0,N,3))       # insieme dei multipli di 3 in [0..N)
S5 = set(range(0,N,5))       # insieme dei multipli di 5 in [0..N)
print(S1,S3,S5, sep='\n')
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}
{0, 3, 6, 9, 12, 15, 18}
{0, 10, 5, 15}
```

In [6...]

```
print('NON multipli di 3  (S1-S3)=', S1-S3)
print('multipli di 3 o di 5 (S3|S5)=', S3|S5)
```

```
NON multipli di 3      (S1-S3)= {1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19}
multipli di 3 o di 5 (S3|S5)= {0, 3, 5, 6, 9, 10, 12, 15, 18}
```

```
In [6...]: print('multipli di 3 o di 5 ma non di 15 (S3^S5)=', S3^S5)
print('multipli di 3 e di 5           (S3&S5)=', S3&S5)
```

```
multipli di 3 o di 5 ma non di 15 (S3^S5)= {3, 5, 6, 9, 10, 12, 18}
multipli di 3 e di 5           (S3&S5)= {0, 15}
```

- `S.pop()` estrazione distruttiva di un elemento a caso (ERRORE if empty)
- `S.add(elemento)` aggiunta di un elemento
- `S.remove(elemento)` rimozione di un elemento (ERRORE if missing)
- `S1.update(S2)` come `S1 |= S2` (distruttiva)

```
In [6...]: S = {12, 45, 2, 78, 19}
print('S prima di pop', S, "\t(notate che l'ordine è diverso)")
X = S.pop()
print('S dopo    pop', S, '\tX=', X)
```

```
S prima di pop {2, 19, 12, 45, 78}      (notate che l'ordine è diverso)
S dopo    pop {19, 12, 45, 78}          X= 2
```

```
In [6...]: S.add(999)
print('S dopo S.add(999)', S)
S.remove(12)
print('S dopo remove(12)', S)
S.remove(92)      # ERRORE se l'elemento non c'è (prima usate in)
```

```
S dopo S.add(999) {19, 999, 12, 45, 78}
S dopo remove(12) {19, 999, 45, 78}
```

---

```
KeyError                                         Traceback (most recent call last)
Cell In[64], line 5
      3 S.remove(12)
      4 print('S dopo remove(12)', S)
----> 5 S.remove(92)      # ERRORE se l'elemento non c'è (prima usate in)
```

```
KeyError: 92
```

In [9... help(set)

Help on class set in module builtins:

```
class set(object)
|   set() -> new empty set object
|   set(iterable) -> new set object
|
|   Build an unordered collection of unique elements.
|
| Methods defined here:
|
|   __and__(self, value, /)
|       Return self&value.
|
|   __contains__(...)
|       x.__contains__(y) <=> y in x.
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattribute__(self, name, /)
|       Return getattr(self, name).
|
|   __gt__(self, value, /)
|       Return self>value.
|
|   __iand__(self, value, /)
|       Return self&=value.
|
|   __init__(self, /, *args, **kwargs)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   __ior__(self, value, /)
|       Return self|=value.
```

```
| __isub__(self, value, /)
|     Return self-=value.

| __iter__(self, /)
|     Implement iter(self).

| __ixor__(self, value, /)
|     Return self^=value.

| __le__(self, value, /)
|     Return self<=value.

| __len__(self, /)
|     Return len(self).

| __lt__(self, value, /)
|     Return self<value.

| __ne__(self, value, /)
|     Return self!=value.

| __or__(self, value, /)
|     Return self|value.

| __rand__(self, value, /)
|     Return value&self.

| __reduce__(...)
|     Return state information for pickling.

| __repr__(self, /)
|     Return repr(self).

| __ror__(self, value, /)
|     Return value|self.

| __rsub__(self, value, /)
```

```
|     Return value-self.  
|  
|__rxor__(self, value, /)  
|     Return value^self.  
|  
|__sizeof__(...)  
|     S.__sizeof__() -> size of S in memory, in bytes  
|  
|__sub__(self, value, /)  
|     Return self-value.  
|  
|__xor__(self, value, /)  
|     Return self^value.  
|  
add(...)  
    Add an element to a set.  
  
    This has no effect if the element is already present.  
|  
clear(...)  
    Remove all elements from this set.  
|  
copy(...)  
    Return a shallow copy of a set.  
|  
difference(...)  
    Return the difference of two or more sets as a new set.  
    (i.e. all elements that are in this set but not the others.)  
|  
difference_update(...)  
    Remove all elements of another set from this set.  
|  
discard(...)  
    Remove an element from a set if it is a member.  
  
    Unlike set.remove(), the discard() method does not raise
```

an exception when an element is missing from the set.

`intersection(...)`

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

`intersection_update(...)`

Update a set with the intersection of itself and another.

`isdisjoint(...)`

Return True if two sets have a null intersection.

`issubset(self, other, /)`

Test whether every element in the set is in other.

`issuperset(self, other, /)`

Test whether every element in other is in the set.

`pop(...)`

Remove and return an arbitrary set element.

Raises KeyError if the set is empty.

`remove(...)`

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

`symmetric_difference(...)`

Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`symmetric_difference_update(...)`

Update a set with the symmetric difference of itself and another.

`union(...)`

| Return the union of sets as a new set.

| (i.e. all elements that are in either set.)

| update(...)

|     Update a set with the union of itself and others.

---

| Class methods defined here:

| \_\_class\_getitem\_\_(...)

|     See PEP 585

---

| Static methods defined here:

| \_\_new\_\_(\*args, \*\*kwargs)

|     Create and return a new object. See help(type) for accurate signature.

---

| Data and other attributes defined here:

| \_\_hash\_\_ = None

## Operazioni sui dizionari (dict)

- `key in D` True se la chiave `key` è presente in `D`
- `D[key]` accesso al valore con chiave `key` (R/W) (ERR se non presente in lettura)
- `D.keys()` generatore dell'elenco delle chiavi (che sono uniche)
- `D.values()` generatore dell'elenco dei valori (con duplicati)
- `D.items()` generatore dell'elenco delle coppie (chiave, valore)
- `D.popitem()` torna l'ultima coppia `(k,v)` e la rimuove
- `D1 | D2` nuovo dizionario con tutte le coppie di `D1` e di `D2` (prese nell'ordine)
- `D1.update(D2)` modifica `D1` aggiungendo le coppie `(K,V)` di `D2` (prese nell'ordine)
  - la stessa cosa di `D1 |= D2` oppure `D1 = D1 | D2`

In [9...]

```
D1 = { 1: 'uno', 2: 'due', 3:'tre' }
D2 = { 11: 'undici', 2: 'ventidue', 33:'trentatre' }
D1.update(D2)
# c'è anche l'assegnamento potenziato |= (union)
#D1 |= D2
D1
```

Out[9...]

```
{1: 'uno', 2: 'ventidue', 3: 'tre', 11: 'undici', 33: 'trentatre'}
```

In [9...]

```
# Esempi di operazioni sui dizionari
D1 = { 1: 'uno', 2: 'due', 3:'tre' }
D2 = { 11: 'undici', 2: 'ventidue', 33:'trentatre' }
D = D1 | D2    # creo un dizionario con tutte le voci di D1 e D2
D
```

Out[9...]

```
{1: 'uno', 2: 'ventidue', 3: 'tre', 11: 'undici', 33: 'trentatre'}
```

## Altre operazioni sui dizionari

- `D.get(key, default)` se la chiave è presente ne torna il valore, altrimenti torna il valore di default
- `D.setdefault(key, default)` se la chiave è presente ne torna il valore altrimenti la inserisce col valore di default (e lo torna)
- `D.pop(key, default)` se la chiave è presente ne torna il valore e la rimuove, altrimenti torna il valore di default (o ERRORE se

no default)

- **D.fromkeys(keys, value)** costruisce un dizionario con le chiavi fornite, tutte associate allo stesso valore indicato

In [9...]

```
# Esempi di operazioni sui dizionari
# estraggo 66 ma se non la trovo torno una stringa di default senza modificare D
print('cerco 66', D.get(66, 'non trovato'))
print(D)

cerco 66 non trovato
{1: 'uno', 2: 'ventidue', 3: 'tre', 11: 'undici', 33: 'trentatre'}
```

In [9...]

```
# cerco 88 ma se non la trovo modifico D e torno il valore (presente o inserito)
print('cerco 88', D.setdefault(88, 'paperoga')) # inserisce paperoga
print(D)
print('cerco 88', D.setdefault(88, 'paperino')) # torna paperoga

cerco 88 paperoga
{1: 'uno', 2: 'ventidue', 3: 'tre', 11: 'undici', 33: 'trentatre', 88: 'paperoga'}
cerco 88 paperoga
```

In [9...]

```
# dizionario con le chiavi prese dai caratteri di una stringa e tutti i valori a 0
dict.fromkeys('ABCDEF', 0)
```

Out[9...]

```
{'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0}
```

In [9...]

```
help(dict)
```

Help on class dict in module builtins:

```
class dict(object)
| dict() -> new empty dictionary
| dict(mapping) -> new dictionary initialized from a mapping object's
|   (key, value) pairs
| dict(iterable) -> new dictionary initialized as if via:
|     d = {}
|     for k, v in iterable:
|       d[k] = v
| dict(**kwargs) -> new dictionary initialized with the name=value pairs
|   in the keyword argument list.  For example:  dict(one=1, two=2)

Built-in subclasses:
StgDict

Methods defined here:

__contains__(self, key, /)
    True if the dictionary has the specified key, else False.

__delitem__(self, key, /)
    Delete self[key].

__eq__(self, value, /)
    Return self==value.

__ge__(self, value, /)
    Return self>=value.

__getattribute__(self, name, /)
    Return getattr(self, name).

__getitem__(self, key, /)
    Return self[key].

__gt__(self, value, /)
```

```
|     Return self>value.  
|  
|__init__(self, /, *args, **kwargs)  
|     Initialize self. See help(type(self)) for accurate signature.  
|  
|__ior__(self, value, /)  
|     Return self|=value.  
|  
|__iter__(self, /)  
|     Implement iter(self).  
|  
|__le__(self, value, /)  
|     Return self<=value.  
|  
|__len__(self, /)  
|     Return len(self).  
|  
|__lt__(self, value, /)  
|     Return self<value.  
|  
|__ne__(self, value, /)  
|     Return self!=value.  
|  
|__or__(self, value, /)  
|     Return self|value.  
|  
|__repr__(self, /)  
|     Return repr(self).  
|  
|__reversed__(self, /)  
|     Return a reverse iterator over the dict keys.  
|  
|__ror__(self, value, /)  
|     Return value|self.  
|  
|__setitem__(self, key, value, /)  
|     Set self[key] to value.
```

```
| __sizeof__(...)  
|     D.__sizeof__() -> size of D in memory, in bytes  
  
| clear(...)  
|     D.clear() -> None. Remove all items from D.  
  
| copy(...)  
|     D.copy() -> a shallow copy of D  
  
| get(self, key, default=None, /)  
|     Return the value for key if key is in the dictionary, else default.  
  
| items(...)  
|     D.items() -> a set-like object providing a view on D's items  
  
| keys(...)  
|     D.keys() -> a set-like object providing a view on D's keys  
  
| pop(...)  
|     D.pop(k[,d]) -> v, remove specified key and return the corresponding value.  
  
|     If the key is not found, return the default if given; otherwise,  
|     raise a KeyError.  
  
| popitem(self, /)  
|     Remove and return a (key, value) pair as a 2-tuple.  
  
|     Pairs are returned in LIFO (last-in, first-out) order.  
|     Raises KeyError if the dict is empty.  
  
| setdefault(self, key, default=None, /)  
|     Insert key with a value of default if key is not in the dictionary.  
  
|     Return the value for key if key is in the dictionary, else default.  
  
| update(...)
```

`D.update([E, ]**F) -> None.` Update D from mapping/iterable E and F.  
If E is present and has a `.keys()` method, then does: `for k in E.keys(): D[k] = E[k]`  
If E is present and lacks a `.keys()` method, then does: `for k, v in E: D[k] = v`  
In either case, this is followed by: `for k in F: D[k] = F[k]`

`values(...)`

`D.values() -> an object providing a view on D's values`

---

Class methods defined here:

`__class_getitem__(...)`

See PEP 585

`fromkeys(iterable, value=None, /)`

Create a new dictionary with keys from iterable and values set to value.

---

Static methods defined here:

`__new__(*args, **kwargs)`

Create and return a new object. See `help(type)` for accurate signature.

---

Data and other attributes defined here:

`__hash__ = None`