



Tuple, Liste, Dizionari



Tuple

- E' una **sequenza ordinata di elementi** con (eventualmente) **tipi diversi**
- Non è possibile cambiare i valori degli elementi (la tupla è **immutabile**)
- Le tuple sono allocate tramite **parentesi tonde**

t = () *empty tuple*

t = (2, "mit", 3)

t[0] → ritorna 2

(2, "mit", 3) + (5, 6) → ritorna (2, "mit", 3, 5, 6)

t[1:2] → slice tuple, ritorna ("mit",)

t[1:3] → slice tuple, ritorna ("mit", 3)

len(t) → ritorna 3




t[1] = 4 → gives error, can't modify object

*extra comma
means a tuple
with one element*



Tuple

- Di solito usate per eseguire uno **swap** tra valori di variabili

<code>x = y</code> <code>y = x</code>		<code>temp = x</code> <code>x = y</code> <code>y = temp</code>		<code>(x, y) = (y, x)</code>	
--	---	--	---	------------------------------	---







- Utilizzate per **ritornare più di un valore** da una funzione

```
def quotient_and_remainder(x, y):  
    q = x // y  
    r = x % y  
    return (q, r)  
  
(quot, rem) = quotient_and_remainder(4, 5)
```

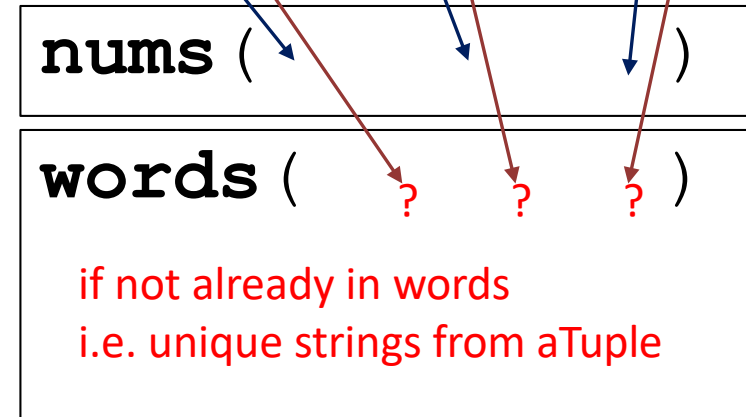
integer division



Manipolazione delle tuple: esempio







aTuple: ((^{ints}  ) , ( ) , ( ))

- **aTuple** è tupla di tuple di tipo (int, string)
- **Esercizio:** Sviluppare una funzione **get_data(aTuple)** che
 - Estrae tutti gli interi da aTuple e li setta come elementi in una nuova tupla.
 - Estrae tutte le stringhe uniche da aTuple e le setta come elementi in una nuova tupla.
 - Ritorna la tuple del minimo e del massimo nella tupla di interi, e il numero di stringhe unique.







Manipolazione delle tuple

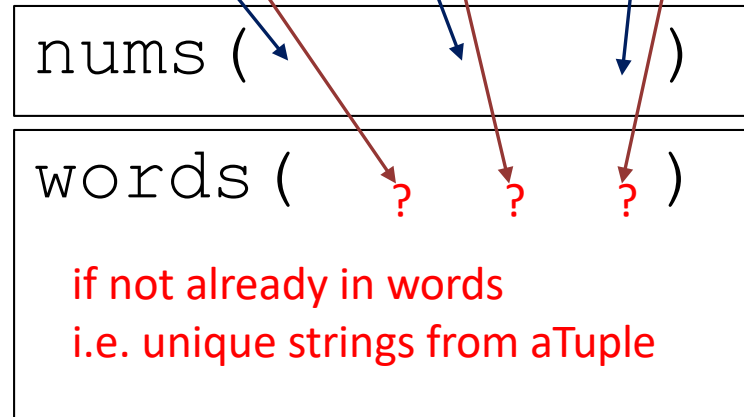
aTuple: ((^{ints}  ) , ( ) , ( ))

- E' possibile **iterare** sulle tuple

```
def get_data(aTuple):  
    nums =   
    words = ()  
    for t in aTuple:  
        nums = nums +   
        if t[1] not in words:  
            words = words + (t[1],)  
    min_n = min(nums)  
    max_n = max(nums)  
    unique_words = len(words)  
    return (min_n, max_n, unique_words)
```

empty tuple

singleton tuple





Liste

- Una **sequenza ordinata** di informazioni, accessibile tramite un *indice*
- Una lista è denotata da **parentesi quadre**, []
- Una lista contiene **elementi**
 - **di solito omogenei** (i.e., tutti interi)
 - può contenere valori di tipi misti (**non comune**)
- A differenza di una tupla, gli elementi di una lista possono cambiare
 - Una lista è **mutabile**



Liste: indici e ordinamento

`a_list = []` *empty list*

`L = [2, 'a', 4, [1, 2]]`

`len(L)` → ritorna 4

`L[0]` → ritorna 2

`L[2]+1` → ritorna 5

`L[3]` → ritorna `[1, 2]`, un'altra lista!

`L[4]` → ritorna un errore

`i = 2`

`L[i-1]` → ritorna 'a' perchè `L[1] = 'a'`



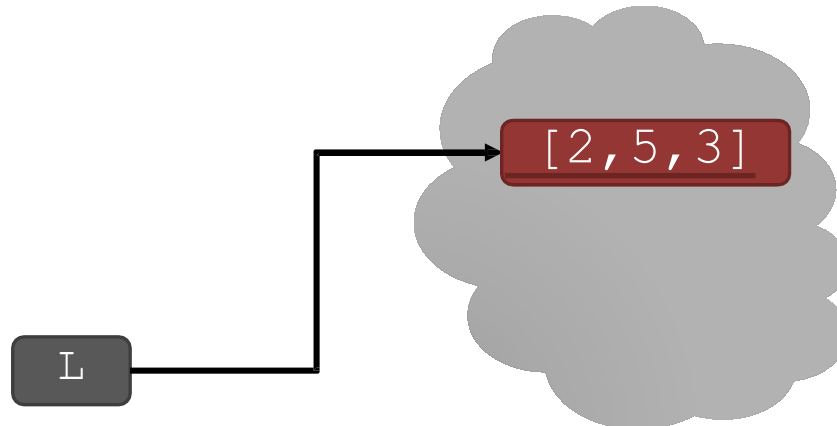
Liste: cambiare gli elementi

- Come abbiamo detto le liste sono **mutabili**
- E' possibile fare l'assegnazione di un elemento ad uno specifico indice cambiando il suo valore

$L = [2, 1, 3]$

$L[1] = 5$

- L sarà $[2, 5, 3]$





Liste: iterare sugli elementi

- **Obiettivo:** eseguire la **somma degli elementi** di una lista
- Il pattern più comune è iterare sugli elementi della lista e calcolare la somma parziale

```
total = 0
for i in range(len(L)):
    total += L[i]
print total
```

```
total = 0
for i in L:
    total += i
print total
```

like strings,
can iterate
over list
elements
directly

- **N.B.:**

- Gli elementi della lista partono da 0 fino a `len(L) - 1`
- `range(n)` va da 0 fino a `n-1`



Operazioni su liste: **append**

- **Aggiungere** elementi alla fine della lista con `L.append(element)`

- Dopo la chiamata ad **append**, la lista **muterà**

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

```
L.append(5) → L is now [2, 1, 3, 5]
```



- **Notare l'operatore punto**

- Le liste sono **oggetti Python** (ricordiamo che **tutto in Python è un oggetto**)
- Gli oggetti hanno *dati*
- Gli oggetti hanno *metodi* e *funzioni*
- L'accesso a tali informazioni avviene tramite (come in Java/C++)
`object_name.do_something()`



Operazioni su liste: extend

- Per combinare più liste insieme si usa il **concatenamento**
- Utilizzando l'operatore **+** otteniamo una nuova lista che è la **concatenazione** delle liste operando
- E' possibile **mutare** una lista con `L.extend(some_list)`

`L1 = [2, 1, 3]`

`L2 = [4, 5, 6]`

`L3 = L1 + L2`

→ `L3` è `[2, 1, 3, 4, 5, 6]`
`L1`, `L2` non cambiano

`L1.extend([0, 6])`

→ `L1` viene mutata in
`[2, 1, 3, 0, 6]`



Operazioni su liste: remove

- E' possibile cancellare un elemento ad uno **specifico indice index** con `del (L[index])`
- Per rimuovere un elemento alla **fine della lista** è possibile usare `L.pop()`, che ritorna l'elemento rimosso
- Per rimuovere uno **specifico elemento** è possibile usare `L.remove(element)`
 - Cerca l'elemento e lo rimuove
 - Se l'elemento è presente più volte, rimuove la prima occorrenza
 - Se l'elemento non è nella lista, ritorna un errore

all these
operations
mutate
the list

```
L = [2, 1, 3, 6, 3, 7, 0] # eseguire in ordine
L.remove(2) → muta L = [1, 3, 6, 3, 7, 0]
L.remove(3) → muta L = [1, 6, 3, 7, 0]
del (L[1])   → muta L = [1, 3, 7, 0]
L.pop()     → ritorna 0 e muta L = [1, 3, 7]
```



Conversione di liste

- `L = list(s)` converte una **stringa in lista** e ritorna una lista che include ogni carattere di `s` in un elemento in `L`
- `s.split()` effettua lo **split di una stringa su un carattere specificato** come parametro, utilizzando lo spazio se la chiamata è senza parametri
- `' '.join(L)` converte una **lista di caratteri in una stringa**, dando un parametro tra singolo apice per specificare il carattere separatore tra ogni elemento

```
s = "I<3 cs"  
list(s)  
s.split('<')  
L = ['a', 'b', 'c']  
' '.join(L)  
'_'.join(L)
```

→ `s` è una stringa

→ ritorna `['I', '<', '3', ' ', 'c', 's']`

→ ritorna `['I', '3 cs']`

→ `L` è una lista

→ ritorna `"abc"`

→ ritorna `"a_b_c"`



Altre operazioni su liste

- `sort()`, `sorted()`
- `reverse()`
- e molte altre...
- <https://docs.python.org/3/tutorial/datastructures.html>

```
L=[9,6,0,3]
```

```
sorted(L)      → Ritorna una lista ordinata, senza senza mutare L
```

```
L.sort()       → muta L=[0,3,6,9]
```

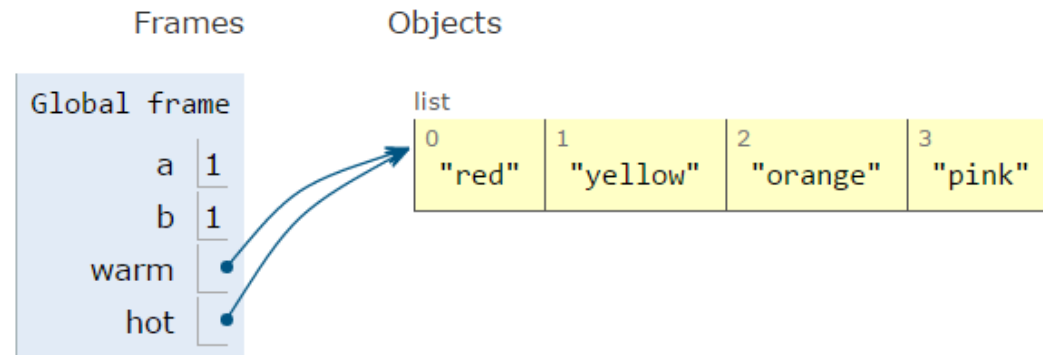
```
L.reverse()    → muta L=[9,6,3,0]
```



Liste e alias

```
1 a = 1
2 b = a
3 print(a)
4 print(b)
5
6 warm = ['red', 'yellow', 'orange']
7 hot = warm
8 hot.append('pink')
9 print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



- **hot** è un **alias** per **warm**
 - Il cambiamento di una variabile impatta sul cambiamento dell'altra!
- L'uso di `append()` ha degli effetti collaterali!



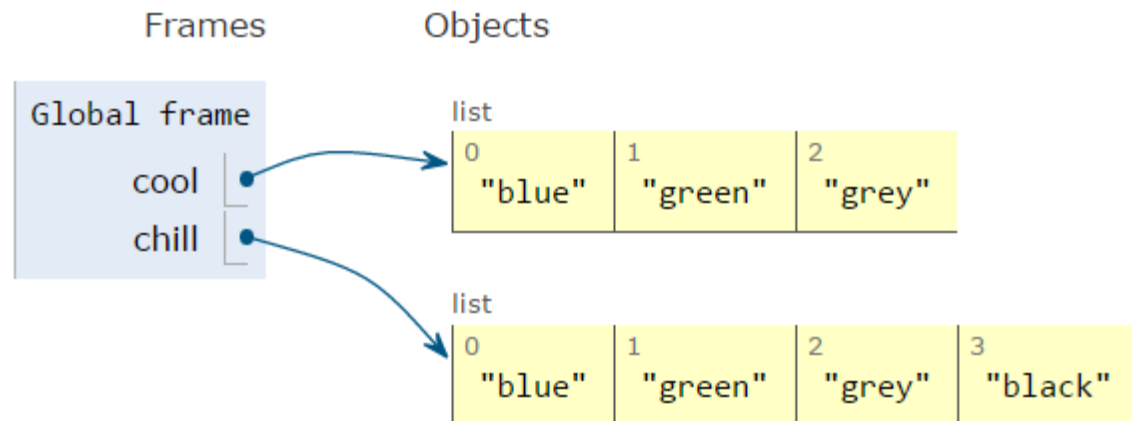
Clonare una lista

- Clonare una lista significa creare una nuova lista e **copiare ogni elemento** usando l'**operatore [:]**

```
chill = cool[:]
```

```
1 cool = ['blue', 'green', 'grey']  
2 chill = cool[:]  
3 chill.append('black')  
4 print(chill)  
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']  
['blue', 'green', 'grey']
```



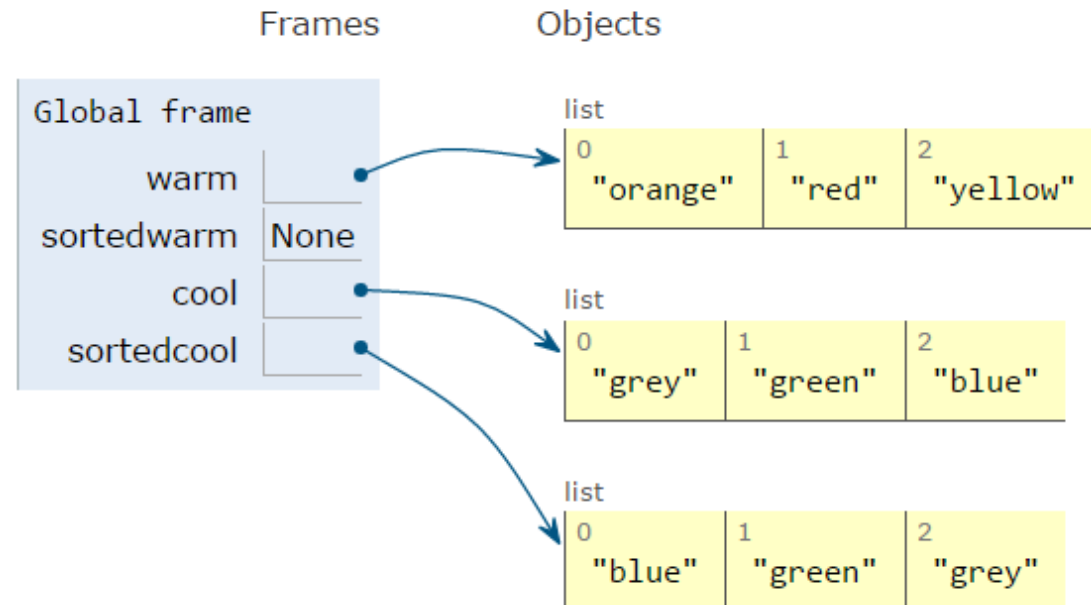


Ordinamento di una lista

- La chiamata a `sort()` **muta** la lista, ritornando `None`
- Invece, la chiamata a `sorted()` **non muta** la lista, è necessario assegnare il valore risultato ad una variable

```
['orange', 'red', 'yellow']  
None  
['grey', 'green', 'blue']  
['blue', 'green', 'grey']
```

```
1 warm = ['red', 'yellow', 'orange']  
2 sortedwarm = warm.sort()  
3 print(warm)  
4 print(sortedwarm)  
5  
6 cool = ['grey', 'green', 'blue']  
7 sortedcool = sorted(cool)  
8 print(cool)  
9 print(sortedcool)
```



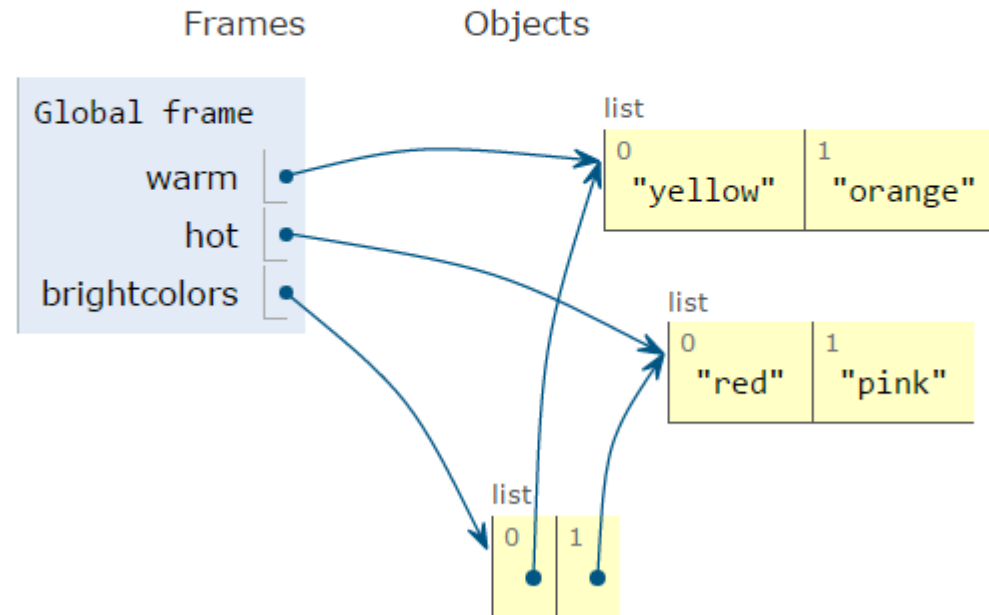


Liste di liste

- Python permette di avere **liste innestate**
- E' possibile incorrere in effetti collaterali dopo la modifica delle liste innestate

```
[['yellow', 'orange'], ['red']]  
['red', 'pink']  
[['yellow', 'orange'], ['red', 'pink']]
```

```
1 warm = ['yellow', 'orange']  
2 hot = ['red']  
3 brightcolors = [warm]  
4 brightcolors.append(hot)  
5 print(brightcolors)  
6 hot.append('pink')  
7 print(hot)  
8 print(brightcolors)
```



Mutazione e iterazione su liste

- **Evitare** la modifica di una lista mentre iteriamo sugli elementi

```
def remove_dups(L1, L2):  
    for e in L1:  
        if e in L2:  
            L1.remove(e)
```



```
def remove_dups(L1, L2):  
    L1_copy = L1[:]  
    for e in L1_copy:  
        if e in L2:  
            L1.remove(e)
```



```
L1 = [1, 2, 3, 4]  
L2 = [1, 2, 5, 6]  
remove_dups(L1, L2)
```

clone list first, note
that `L1_copy = L1`
does NOT clone

- Se usiamo `remove_dups` `L1` è `[2, 3, 4]` non `[3, 4]`, perché?
 - Python utilizza un contatore interno per tenere traccia dell'indice corrente nel ciclo
 - I cambiamenti (e.g., rimozione) **mutano la lunghezza della lista** ma **Python non aggiorna tale contatore!**
 - Nel ciclo dell'esempio, non vedremo mai l'elemento di valore 2 perché dopo la rimozione diventerà l'elemento di indice 0!



Come memorizzare le informazioni di studenti?

- Finora, possiamo memorizzare tali informazioni mantenendo liste separate

```
names = ['Ana', 'John', 'Denise', 'Katy']
```

```
grade = ['B', 'A+', 'A', 'A']
```

```
course = [2.00, 6.0001, 20.002, 9.01]
```

Soluzione:

- Una **lista separata** per ogni elemento (item)
- Ogni lista deve avere la **stessa lunghezza**
- Le info memorizzate per tutte le liste allo **stesso indice** fanno riferimento a info per la stessa persona



Aggiornare/Ottenere info degli studenti

```
def get_grade(student, name_list, grade_list, course_list):  
    i = name_list.index(student)  
    grade = grade_list[i]  
    course = course_list[i]  
    return (course, grade)
```

- **Genero confusione** se ho molte info da tenere traccia
- Devo mantenere **molte liste** e utilizzarle come argomento
- Devo **sempre indicizzarle** tramite interi



Il dizionario (dict)

- Tipi di dato utili per **indicizzare elementi di interesse direttamente** (non per forza interi)
- Utili per utilizzare **una struttura dati**, senza usare liste separate

Lista

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
...	...

index

element

Dizionario

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom
index by
label

element



Il dizionario in Python

- Memorizza una coppia di valori
 - **key**
 - **value**

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

custom
index by
label

element

my_dict = {} empty
dictionary

grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}

↑
key1

↑
val1

↑
key2

↑
val2

↑
key3

↑
val3

↑
key4

↑
val4



Il dizionario: lookup

- L'operazione di **lookup** è simile ad indicizzare liste
- **Cerca** la chiave (**key**)
- **Ritorna** il **valore** associato con la chiave
- Se la chiave non esiste, ritorna un errore

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

```
grades['John']      ➔ ritorna 'A+'
```

```
grades['Sylvan']    ➔ Ritorna un KeyError
```




Operazioni su dizionari

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **aggiungere** una entry

```
grades['Sylvan'] = 'A'
```

- **test** se la chiave è nel dizionario

```
'John' in grades    ➔ returns True  
'Daniel' in grades ➔ returns False
```

- **cancellare** una entry

```
del(grades['Ana'])
```



Operazioni su dizionari

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- Ottenere un **iterabile che sia una lista di tutte le chiavi**

```
grades.keys() ➔ ritorna ['Denise', 'Katy', 'John', 'Ana']
```

- Ottenere un **iterabile che sia una lista di tutti i valori**

```
grades.values() ➔ ritorna ['A', 'A', 'A+', 'B']
```



Chiavi e valori di dizionari

- **Valori**

- Possono essere di qualunque tipo (**immutabile e mutabile**)
- Possono essere **duplicati**
- I valori di un dizionario possono essere liste o addirittura altri dizionari!

- **Chiavi**

- Devono essere **uniche**
- Tipo **immutabile** (`int`, `float`, `string`, `tuple`, `bool`)

- **Non c'è ordinamento** nè per le chiavi né per i valori!

```
d = {4:{1:0}, (1,3):"twelve", 'const':[3.14,2.7,8.44]}
```



Liste vs Dizionari

Lista

- Sequenza **ordinata** di elementi
- La ricerca degli elementi è fatta attraverso un indice intero
- Gli indici hanno un **ordinamento**
- Gli indici sono **interi**

Dizionario

- **Associa** chiavi con valori
- La ricerca di un elemento è fatta attraverso un altro elemento
- **Nessun ordinamento** è garantito
- Le chiavi possono essere di qualunque tipo **immutabile**



Esempio dizionario: Analizzatore del testo di una canzone

- 1) Creare un **frequency dictionary** che mappa `str:int`
- 2) Trovare **la parola più frequente** e quante volte occorre nel testo della canzone
 - Usare una lista nel caso in cui ci siano più parole frequenti
 - Ritornare una tupla (`list, int`) per (`words_list, highest_freq`)
- 3) Trovare la **parola che occorre almeno X volte**
 - Lasciare che l'utente specifichi la soglia X
 - Ritornare una lista di tuple, ogni tupla è sia del tipo (`list, int`)
che contiene la lista delle parole ordinata dalla loro frequenza
 - IDEA: Dal dizionario della canzone, trova la parola più frequente, cancellala, ripeti. Funziona perché stiamo mutando il dizionario della canzone.



Analizzatore del testo di una canzone: creazione dizionario

```
def lyrics_to_frequencies(lyrics):  
    myDict = {}  
    for word in lyrics:  
        if word in myDict:  
            myDict[word] += 1  
        else:  
            myDict[word] = 1  
    return myDict
```

can iterate over list
can iterate over keys
in dictionary
update value
associated with key



Analizzatore del testo di una canzone: uso del dizionario

```
def most_common_words(freqs):  
    values = freqs.values()  
    best = max(values)  
    words = []  
    for k in freqs:  
        if freqs[k] == best:  
            words.append(k)  
    return (words, best)
```

this is an iterable, so can
apply built-in function

can iterate over keys
in dictionary



Analizzatore del testo di una canzone: proprietà del dizionario

```
def words_often(freqs, minTimes):  
    result = []  
    done = False  
    while not done:  
        temp = most_common_words(freqs)  
        if temp[1] >= minTimes:  
            result.append(temp)  
            for w in temp[0]:  
                del(freqs[w])  
        else:  
            done = True  
    return result
```

*can directly mutate
dictionary; makes it
easier to iterate*

```
print(words_often(beatles, 5))
```