

Tuple in Python



Tuple

Un secondo tipo di collezione in Python è dato dalle tuple

Una **tupla** è una sequenza immutabile di oggetti

Si può definire una tupla con la notazione:

```
<tupla> ::= ( <espressioni> ) | <espressioni>  
<espressioni> ::= | <espressione> | <espressione> {, <espressione>}
```

- La notazione è la stessa delle liste, ma si usano le **parentesi tonde**
- ...O addirittura nessuna parentesi!



Tuple

Vediamo qualche esempio di definizione di tupla

```
In [1]: (2, 4, 6) # con le parentesi tonde
```

```
Out[1]: (2, 4, 6)
```

```
In [2]: 2, 4, 6 # senza parentesi
```

```
Out[2]: (2, 4, 6)
```

È (ovviamente) possibile assegnare una tupla ad una variabile:

```
In [3]: t = (2, 4, 6)  
print(t)
```

```
(2, 4, 6)
```



Accesso a Tuple

Si può accedere ad una tupla con le stesse modalità di una lista

```
In [4]: t = (2, 4, 6, 9)
print(t[1])
print(t[-1])
print(t[1:3])
```

```
4
9
(4, 6)
```

Si può accedere con un indice singolo (anche negativo) o con slice

Come una lista, una tupla può contenere dati eterogenei:

```
In [5]: t = (1, 3.5, False)
print(t)
```

```
(1, 3.5, False)
```



Immutabilità delle Tuple

Le tuple sono sequenze immutabili

...Questo è la differenza principale con le liste

- Non è possibile ri-assegnare un elemento di una tupla:

```
In [6]: t = (1, 4, 7)
        t[0] = 2
```

```
-----
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_58/1674606074.py in <module>
      1 t = (1, 4, 7)
----> 2 t[0] = 2

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

- Non è neppure possibile aggiungere o rimuovere elementi
- Con le liste si può, anche se lo abbiamo solo accennato



Operazioni su Tuple

Le tuple supportano gli stessi operatori delle liste

Si può usare l'operatore +

```
In [7]: t1, t2 = (1, 2), (3, 4)
        t1 + t2
```

```
Out[7]: (1, 2, 3, 4)
```

...L'operatore *

```
In [8]: t1 * 3
```

```
Out[8]: (1, 2, 1, 2, 1, 2)
```

...E l'operatore in:

```
In [9]: print(2 in t1)
```



True

Tuple Unpacking

È possibile assegnare gli elementi di una tupla ad altrettante variabili

In Python si chiama "spacchettamento" di tuple (**tuple unpacking**)

```
In [10]: a, b, c = (2, 4, 6)
         print(a)
         print(b)
         print(c)
```

```
2
4
6
```

- Gli elementi della tupla a dx dell'uguale
- ...Vengono assegnate alle variabili a sx dell'uguale

Si procede per posizione:

- Il primo elemento va nella prima variabile
- ...Il secondo elemento nella seconda variabile e così via



Tuple Unpacking

Se qualche variabile non è di interesse...

...È possibile assegnarla alla variabile speciale "_" (underscore)

```
In [11]: a, _, _ = (3, 7, 9)
```

Il tuple unpacking è una funzionalità molto comoda

■ Viene spesso usata per definire molte variabili in un'unica linea

```
In [12]: a, b, c = 4, 8, 12
```

■ ...E a dispetto del nome funziona anche con le liste:

```
In [13]: a, b, c = [1, 2, 3]
print(f'a: {a}, b: {b}, c: {c}')
```

```
a: 1, b: 2, c: 3
```



Insieme



Insiemi

Un terzo tipo di collezioni è dato dagli insiemi:

Un **insieme** è una collezione di elementi mutabile, non-ordinata e senza replicazioni

Si può definire un insieme con la notazione:

```
<insieme> ::= "{" <espressioni> "}"  
<espressioni> ::= | <espressione> | <espressione> {, <espressione>}
```

- La notazione è la stessa delle liste, ma si usano le **parentesi graffe**

Vediamo un semplice esempio:

```
In [14]: {1, 2, 3}
```

```
Out[14]: {1, 2, 3}
```



Insiemi

Un insieme non può contenere duplicati

Se si prova ad inserire lo stesso elemento due volte, il duplicato viene scartato:

```
In [15]: {1, 1, 2, 2, 3, 3}
```

```
Out[15]: {1, 2, 3}
```

Un insieme è una collezione non ordinata

...Quindi accedere con un indice non ha senso:

```
In [16]: s = {1, 2, 3}
s[0]
```

```
-----
TypeError                                Traceback (most recent call last)
/tmp/ipykernel_58/3287493502.py in <module>
      1 s = {1, 2, 3}
----> 2 s[0]
```

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



Operazioni su Insiemi

Anche per gli insiemi sono disponibili alcuni operatori specifici

```
In [17]: s1, s2 = {1, 2}, {2, 3}
print('s1 | s2:', s1 | s2)
print('s1 & s2:', s1 & s2)
print('s1 - s2:', s1 - s2)
print('s1 ^ s2:', s1 ^ s2)
```

```
s1 | s2: {1, 2, 3}
s1 & s2: {2}
s1 - s2: {1}
s1 ^ s2: {1, 3}
```

- L'operatore `|` restituisce l'unione di due insiemi
- L'operatore `&` restituisce l'intersezione di due insiemi
- L'operatore `-` restituisce la differenza di due insiemi
- L'operatore `^` restituisce l'unione degli elementi esclusivi



Operazioni su Insiemi

L'operatore `in` si comporta nel solito modo

```
In [18]: s1 = {1, 2}
         print('3 in s2:', 3 in s1)

3 in s2: False
```

Per aggiungere e rimuovere elementi

- Si possono usare gli operatori `|` e `-`
 - ...Ma ci sono anche metodi più appropriati
- Li vedremo più avanti nel corso



Conversione di Collezioni

È possibile **convertire** il tipo di una collezione usando speciali funzioni

- Si usa il costruttore di liste `list` per ottenere una lista:

```
In [19]: t = (1, 2, 3)
         list(t)
```

```
Out[19]: [1, 2, 3]
```

- Si usa il costruttore di tuple `tuple` per ottenere una tupla:

```
In [20]: tuple([1, 2, 3])
```

```
Out[20]: (1, 2, 3)
```

- Si usa il costruttore di insiemi `set` per ottenere un insieme:

```
In [21]: set([1, 1, 2, 2, 3, 3])
```

```
Out[21]: {1, 2, 3}
```

Dizionari in Python



Dizionari in Python

Un quarto tipo di collezione in Python è dato dai dizionari

Un **dizionario** è una collezione di oggetti, non ordinata, mutabile, indicizzata mediante valori specificati dall'utente

Si può definire un dizionario con la notazione:

```
<dizionario> ::= dict() | "{" <coppia> {, <coppia>} "}"  
<coppia> ::= <espressione chiave> : <espressione valore>
```

- Un dizionario vuoto lo si indica con `dict()` (costruttore di dizionari)
- Altrimenti si racchiudono, tra parentesi graffe...
- ...Una o più coppie chiave-valore



Dizionari in Python

Vediamo qualche esempio

```
In [22]: d = {'nome': 'Mario', 'cognome': 'Pinto'}  
d
```

```
Out[22]: {'nome': 'Mario', 'cognome': 'Pinto'}
```

- Un dizionario con due chiavi (e quindi due valori)
 - Alla chiave 'nome' corrisponde il valore 'Mario'
 - Alla chiave 'cognome' corrisponde il valore Pinto'

```
In [23]: d = dict()  
d
```

```
Out[23]: {}
```

- Un dizionario vuoto



Dizionari in Python

Vediamo qualche esempio

```
In [24]: stato = {0: 'ottimo', 1: 'ammissibile', 2: 'non ammissibile', 3: 'indefinito'}
```

- Le chiavi non devono essere necessariamente stringhe

```
In [25]: d = {1+1: 'b40878', 'voto': 30 - 3 - 1}
d
```

```
Out[25]: {2: 'b40878', 'voto': 26}
```

- Sia chiavi che valori possono essere specificati mediante espressioni



Accesso a Dizionari

Per accedere ad un dizionario si usa l'operatore di indicizzazione

...E si passa come argomento la chiave desiderata:

```
In [26]: d = {'nome': 'Mario', 'cognome': 'Pinto'}  
print(d['nome'])  
print(d['cognome'])
```

```
Mario  
Pinto
```

Se si prova ad usare una chiave mancante si ottiene un errore:

```
In [27]: d['età']
```

```
-----  
KeyError                                Traceback (most recent call last)  
/tmp/ipykernel_58/2959126869.py in <module>  
----> 1 d['età']  
  
KeyError: 'età'
```



Chiavi Valide

Non tutti i tipi di dato possono esse usati come chiavi

Tendenzialmente, qualunque tipo di dato **immutabile** è valido

- Per esempio si possono usare i **tipi di dato semplici**

```
In [28]: d = {1: 'tizio', 1.3: 'caio', True: 'sempronio', '4': 'Franco'}  
d
```

```
Out[28]: {1: 'sempronio', 1.3: 'caio', '4': 'Franco'}
```

- ...Oppure le **tuple**

```
In [147]: distanze = {(0, 1): 1.3, (0, 2): 4.1, (1, 2): 2.2}  
distanze
```

```
Out[147]: {(0, 1): 1.3, (0, 2): 4.1, (1, 2): 2.2}
```

- Si può sfruttare questa proprietà per memorizzare matrici "sparse"

- ...ossia in cui la maggior parte delle celle vale 0 o non è definita

Accesso a Dizionari

È possibile sia **ottenere** che **assegnare** valori

...Esattamente come per le liste

- Se l'indizzazione appare al di fuori di un assegnamento, o a dx del segno "="
- Allora l'accesso è una espressione e come tale denota un valore:

```
In [148]: d = {'nome': 'Mario', 'cognome': 'Pinto'}  
print(d['nome'] + ' ' + d['cognome'])
```

Mario Pinto

- Se l'indicizzazione appare a sx del segno "=", indica la cella in cui scrivere

```
In [149]: d['nome'] = 'Vario'  
print(d['nome'] + ' ' + d['cognome'])
```

Vario Pinto



Accesso a Dizionari

Quando si assegna un valore ad un elemento

- ...Se la chiave usata per l'accesso non esiste, questa viene creata
- Quindi l'assegnamento procede come al solito

```
In [150]: d = {'nome': 'Mario', 'cognome': 'Pinto'}  
          d['età'] = 34  
          print(d)  
  
          {'nome': 'Mario', 'cognome': 'Pinto', 'età': 34}
```

- In questo modo è possibile aggiungere valori al dizionario

È possibile rimuovere valori da un dizionario

- Si può usare l'istruzione "del", come per le liste
- ...E come per le liste, è meglio evitare di farlo



Operazioni su Dizionari

Anche per i dizionari sono disponibili alcuni operatori specifici

```
In [158]: d1, d2 = {'nome': 'Mario', 'voto': 27}, {'nome': 'Vario', 'età': 20}
print('d1 | d2:', d1 | d2)
print('d2 | d1:', d2 | d1)

d1 | d2: {'nome': 'Vario', 'voto': 27, 'età': 20}
d2 | d1: {'nome': 'Mario', 'età': 20, 'voto': 27}
```

- L'operatore " | " unisce due dizionari
- Se ci sono chiavi identiche nei due, prevale il valore del **secondo** dizionario

L'operatore " in " si applica alle chiavi

```
In [160]: d1 = {'nome': 'Mario', 'voto': 27}
print("'nome' in d1:", 'nome' in d1)
print("'Mario' in d1:", 'Mario' in d1)

'nome' in d1: True
'Mario' in d1: False
```

