Python: Contenitori intrinseci (*built-in containers*)

Rights & Credits

Il notebook è stato variamente trasformato, anche con l'aggiunta di contenuti, da Agostino Migliore, a partire dall'originale di:

- Simone Campagna (CINECA)
- Mirko Cestari (CINECA)
- Nicola Spallanzani (CNR-NANO)

Contenitori

Contenitori

Un contenitore è un oggetto (si noti che ogni cosa è definita come un oggetto in un object-oriented programming language come Python) che raccoglie più oggetti: variabili, interi e così via. Gli elementi di un contenitore possono essere anch'essi contenitori. I contenitori di Python sono molto facili da usare, versatili ed efficienti. I principali tipi di contenitori predefiniti sono

- tuple ()
- list []
- dict {}
- set {}, frozenset

tupla

Una tupla è un elenco o collezione **ordinata** (quindi una sequenza) e **immutabile** di valori tipo arbitrario. Anche il numero di valori contenuti nella tupla è arbitrario. Il concetto di *ennupla* usato in matematica è molto simile, anche se usualmente riferito a oggetti dello stesso tipo. Definiamo una tupla di nome `a` come segue:

In [1]:
```
a = (3, 4, 5)
a
```

Out[1]:
```
(3, 4, 5)
```

Le operazioni di *slicing* viste nella lezione precedente si possono effettuare anche sulle tuple.

In [2]:
```
a[1], a[1:], a[2:], a[:2], a[1:2], a[:2][0]
```

Out[2]:
```
(4, (4, 5), (5,), (3, 4), (4,), 3)
```

In [3]:
```
b = 2,3  # Si può definire la lista anche senza usare le parentesi!
b
```

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

Come abbiamo visto, gli elementi di una tupla possono essere di qualsiasi tipo, il che significa che una tupla non è necessariamente omogenea:

In [4]:
```
z = 3 + 1j
c = (4, z, "beta", b)
c
```

Out[4]:
```
(4, (3+1j), 'beta', (2, 3))
```

In [5]:
```
r, i = z.real, z.imag
u = (r, i)
u
```

Out[5]:
```
(3.0, 1.0)
```

In [6]:
```
d, m = divmod(100, 3)
d, m
```

Out[6]:
```
(33, 1)
```

Operazioni con le tuple

In [7]:
```
A = (1, 2, 3)
B = (4, 5, 6)
```

Concatenazione

In [8]:
```
A + B
```

Out[8]:
```
(1, 2, 3, 4, 5, 6)
```

In [9]:
```
A + 2*B
```

Out[9]:
```
(1, 2, 3, 4, 5, 6, 4, 5, 6)
```

Reiterazione

In [10]:
```
A*4
```

Out[10]:
```
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Circa la sintassi delle tuple

Come abbiamo visto, non è necessario usare le parentesi per definire una tupla (esse sono comunque assegnate all'atto della creazione della tupla). La virgola è, invece, importante per definire l'oggetto come tupla.

In [11]:
```
x = 1
type(x)
```

Out[11]:
```
int
```

In [12]:
```
y = 1,
type(y)
```

Out[12]:
```
tuple
```

In [13]:
```
y
```

Out[13]:
```
(1,)
```

Le parentesi sono necessarie solo per definire la tupla vuota:

In [14]:
```
V = ()
V
```

Out[14]:
```
()
```

In [15]:
```
t = (1, 2, 3)
```

In [16]:
```
t[0] = 17
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 t[0] = 17

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

In [17]:
```
T = (10, 5, 7, 19)
```

In [18]:
```
T[3:]
```

Out[18]:
```
(19,)
```

In [19]:
```
T[3]
```

Out[19]:
```
19
```

In [20]:
```
t + T[3:]
```

Out[20]: (1, 2, 3, 19)

In [21]:
```
t + T[3]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 1
----> 1 t + T[3]

TypeError: can only concatenate tuple (not "int") to tuple
```

lista

Una lista e' una sequenza **ordinata**, **mutabile** e di lunghezza arbitraria di oggetti. Le liste sono, in pratica, tuple mutabili e sono definite con parentesi quadre.

In [22]:
```
R = [1,2,3]
R
```

Out[22]: [1, 2, 3]

Operazioni semplici con le liste

In [23]:
```
2*R
R
```

Out[23]: [1, 2, 3]

In [24]:
```
R *= 3
R
```

Out[24]: [1, 2, 3, 1, 2, 3, 1, 2, 3]

In [25]:
```
S = 2*R
S
```

Out[25]: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

In [26]:
```
R += [4]
R
```

Out[26]: [1, 2, 3, 1, 2, 3, 1, 2, 3, 4]

Possiamo assegnare elementi ad una lista:

In [27]:
```
R[1:4] = ["x","y","z"]
R
```

Out[27]: [1, 'x', 'y', 'z', 2, 3, 1, 2, 3, 4]

In [28]:
```
R[5:12] = [7, 8, 9]
R
```

Out[28]:
```
[1, 'x', 'y', 'z', 2, 7, 8, 9]
```

In [29]:
```
R[3:5] = [10, 11, 12, 13]
R
```

Out[29]:
```
[1, 'x', 'y', 10, 11, 12, 13, 7, 8, 9]
```

Metodi per manipolare le liste

Si può aggiungere un elemento alla fine di una lista (appendere, in inglese *append*) tramite il metodo append() della classe lista (cioè, una funzione definita come attributo di tale classe). Tale funzione prende un solo argomento in ingresso ed è appunto l'oggetto da aggiungere alla lista:

In [30]:
```
l = [1, 2, 3]
l
```

Out[30]:
```
[1, 2, 3]
```

In [31]:
```
l.append(12)
l
```

Out[31]:
```
[1, 2, 3, 12]
```

Si noti che la lista è mutabile; quindi essa è cambiata. Di conseguenza, ripetendo l'operazione di sopra, cambieremo la lista ulteriormente:

In [32]:
```
l.append(12)
```

In [33]:
```
print(l)
```

```
[1, 2, 3, 12, 12]
```

In [34]:
```
l.append([21, 17])
l
```

Out[34]:
```
[1, 2, 3, 12, 12, [21, 17]]
```

Se vogliamo aggiungere due o più elementi separati, usiamo il metodo extend():

In [35]:
```
l.extend([11, 13, 10, 20])
l
```

Out[35]:
```
[1, 2, 3, 12, 12, [21, 17], 11, 13, 10, 20]
```

Il metodo insert(A, p) inserisce l'oggetto specificato come primo argomento (A) alla posizione passata come secondo argomento (p):

In [36]:
```
l.insert(4, "XYZ")
l
```

Out[36]:
```
[1, 2, 3, 12, 'XYZ', 12, [21, 17], 11, 13, 10, 20]
```

Il metodo pop() rimuove l'ultimo elemento di una lista e lo riporta:

In [37]:
```
l.pop()
```

Out[37]:
```
20
```

In [38]:
```
l.pop()
```

Out[38]:
```
10
```

In [39]:
```
l
```

Out[39]:
```
[1, 2, 3, 12, 'XYZ', 12, [21, 17], 11, 13]
```

Il metodo remove() elimina il primo elemento uguale a quello passato come suo argomento:

In [40]:
```
l.remove(12)
l
```

Out[40]:
```
[1, 2, 3, 'XYZ', 12, [21, 17], 11, 13]
```

L'istruzione, o comando, del può rimuovere un elemento o una fetta della lista e persino l'intero contenuto di una lista.

In [41]:
```
del l[3]
l
```

Out[41]:
```
[1, 2, 3, 12, [21, 17], 11, 13]
```

In [42]:
```
del l[4]
l
```

Out[42]:
```
[1, 2, 3, 12, 11, 13]
```

In [43]:
```
del l[:]
l
```

Out[43]:
```
[]
```

In [44]:
```
len(l)
```

Out[44]:
```
0
```

In [45]:
```
l = [3, 1, 4, 7, 21, 12, 17]
l
```

Out[45]:
```
[3, 1, 4, 7, 21, 12, 17]
```

In [46]:
```
len(l)
```

Out[46]:
```
7
```

Slicing di liste

Le liste, come le tuple, possono essere affettate.

In [47]:
```
l[0] # Si noti che il risultato di questa operazione non è un contenitore.
```

Out[47]:
```
3
```

In [48]:
```
l[-2]
```

Out[48]:
```
12
```

In [49]:
```
l
```

Out[49]:
```
[3, 1, 4, 7, 21, 12, 17]
```

In [50]:
```
l[-2:]
```

Out[50]:
```
[12, 17]
```

In [51]:
```
l[4] = 1, 2, 3
l
```

Out[51]:
```
[3, 1, 4, 7, (1, 2, 3), 12, 17]
```

Le operazioni di *slicing* possono anche essere effettuate con un certo passo (*stride*), che può essere anche negativo (*extended slicing*). In tal caso, le parentesi quadre devono contenere informazioni sul range da cui pescare gli elementi e sulla lunghezza del passo.

In [52]:
```
l[0:6:2]
```

Out[52]:
```
[3, 4, (1, 2, 3)]
```

In [53]:
```
L = [1,2,3,4,5,6,7,8,9]
L[7:0:-2]
```

Out[53]:
```
[8, 6, 4, 2]
```

In [54]:
```
M = [1, 2, 4, 7, 3, 8, 5, 6, 17, 10, 12]
M[:8:3]
```

Out[54]:
```
[1, 7, 5]
```

In [55]:
```
M.count(10)
```

Out[55]: 1

Altri metodi per manipolare le liste

Il metodo sort() consente di ordinare una lista omogenea (come impostazione predefinita, cioè *by default*, in ordine crescente).

In [56]:
```
l.sort()
l
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[56], line 1
----> 1 l.sort()
      2 l

TypeError: '<' not supported between instances of 'tuple' and 'int'
```

In [57]:
```
del l[4]
```

In [58]:
```
l.sort()
print(l)
```

```
[1, 3, 4, 7, 12, 17]
```

In [59]:
```
Q = ['a','d','h','k','s', "a", "b", 'r','v','t','u']
Q.sort()
Q
```

Out[59]: ['a', 'a', 'b', 'd', 'h', 'k', 'r', 's', 't', 'u', 'v']

Per ordinare la lista in modo decrescente, specifichiamo l'argomento di `sort()` come segue

In [60]:
```
l.sort(reverse=True)
Q.sort(reverse=True)
l, Q
```

Out[60]: ([17, 12, 7, 4, 3, 1], ['v', 'u', 't', 's', 'r', 'k', 'h', 'd', 'b', 'a', 'a'])

Si noti, da sopra, che il metodo `sort()` accetta il parametro `reverse` con un valore booleano (valore di tipo `bool`) e che l'argomento viene passato al metodo "chiamando per nome" il parametro di input e assegnandogli il valore. Questa si chiama assegnazione via nome o via keyword.

L'ordine degli elementi in una lista può essere invertito usando il metodo reverse():

In [61]:
```
Q.reverse()
Q
```

Out[61]: ['a', 'a', 'b', 'd', 'h', 'k', 'r', 's', 't', 'u', 'v']

Qual'è, allora, la differenza con `l.sort(reverse=True)`?

Introspezione di liste

L'Introspezione consiste nella capacità di Python di esaminare il tipo o le proprietà di un oggetto, in questo caso una lista, all'esecuzione (*at runtime*). Ci sono dei metodi che interrogano le proprietà di una lista e li riportano all'utente. Facciamo pochi esempi. Conosciamo già l'uso della funzione intrinseca `len()`.

In [62]:
```
len(Q)
```

Out[62]: 11

Il metodo count prende un argomento e conta il numero di volte che il valore dell'argomento compare tra gli elementi della lista.

In [63]:
```
Q.count('a')
```

Out[63]: 2

In [64]:
```
Q.count('b')
```

Out[64]: 1

Il metodo index fornisce la posizione della prima volta che un elemento si presenta in una lista. Tale metodo, ma anche `count()` per esempio, funziona pure con le tuple.

In [65]:
```
Q.index("a")
```

Out[65]: 0

In [66]:
```
Q.index('c')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[66], line 1
----> 1 Q.index('c')

ValueError: 'c' is not in list
```

Esercizio

Provate

In [67]:
```
P = [1, 2, 4, 7, 3, 8, 5, 6, 17, 10, 12]
P[1::3]
P[:5:3]
P[8::-3]
```

Out[67]: [17, 8, 4]

e spiegate i risultati.

range

La funzione range può essere usata per costruire una sequenza di interi:

In [68]:
```
range(4)
```

Out[68]: range(0, 4)

In [69]:
```
l = list(range(4))
t = tuple(range(7))
print(l,t,sep="    ")
```

```
[0, 1, 2, 3]    (0, 1, 2, 3, 4, 5, 6)
```

In [70]:
```
list(range(3, 9))
```

Out[70]: [3, 4, 5, 6, 7, 8]

In [71]:
```
list(range(3, 12, 2))
```

Out[71]: [3, 5, 7, 9, 11]

Si noti che la funzione built-in range non costruisce una lista o una tupla. Esso genera una sequenza, che è un oggetto iterabile in una certa direzione, senza immagazzinarla in memoria o riportarla. Tuttavia, al momento della creazione, la sequenza può essere utilizzata da altre funzioni e, in particolare, sopra è usata per creare una lista. A tal fine, abbiamo usato la funzione list, che può convertire `iterabili` in liste.

set e frozenset

Un set è una sequenza **non ordinata** e **mutabile** di valori di tipo e numero arbitrari, senza doppioni. Frozenset è un set **immutabile**. La mancanza di ordine comporta per esempio che, quando inseriamo elementi nel set, l'interprete di Python può cambiarne l'ordine in un modo che poi ne favorisce la ricerca.

In [72]:
```
S1 = {1, 3, 4}
S1
```

Out[72]: {1, 3, 4}

In [73]:
```
S2 = set((1, 2, 1, 3))
S2
```

Out[73]: {1, 2, 3}

In [74]:
```
S3 = set([2, 4, 8])
S3
```

Out[74]:
```
{2, 4, 8}
```

In [75]:
```
S4 = set(range(0,10))
S4
```

Out[75]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

In [76]:
```
S5 = set()
S5.add(10)
S5.add(12)
S5.add(17)
S5
```

Out[76]:
```
{10, 12, 17}
```

In [77]:
```
S5.discard(12)
S5
```

Out[77]:
```
{10, 17}
```

In [78]:
```
S5.clear()
S5
```

Out[78]:
```
set()
```

Operazioni tra sets

In [79]:
```
s = {0, 1, 2, 3, 4, 5, 6}; t = {4, 5, 6, 7, 8, 9, 10}
```

In [80]:
```
u = s.union(t)
u
```

Out[80]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

In [81]:
```
s.intersection(t)
```

Out[81]:
```
{4, 5, 6}
```

In [82]:
```
s.difference(t)
```

Out[82]:
```
{0, 1, 2, 3}
```

In [83]:
```
t.difference(s)
```

Out[83]:
```
{7, 8, 9, 10}
```

In [84]:
```
s.symmetric_difference(t)
```

Out[84]:
```
{0, 1, 2, 3, 7, 8, 9, 10}
```

dict

Il nome di tale contenitore, dict, deriva dall'inglese *dictionary* (*dizionario*). Esso associa un nome chiave (o semplicemente **chiave**) a ciascun **valore** di interesse. Ogni key (arbitrariamente scelta, ma poi unica e immutabile) è associata a un valore arbitrario e generalmente mutabile.

In [85]:
```
atomic_number = {'H': 1, 'He': 2, 'C': 6, 'Fe': 26}
atomic_number
```

Out[85]:
```
{'H': 1, 'He': 2, 'C': 6, 'Fe': 26}
```

In [86]:
```
D = dict(a=1, b=2, c=3, d=4)
D
```

Out[86]:
```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Adesso cerchiamo il numero atomico dell'atomo di carbonio nel dizionario:

In [87]:
```
atomic_number['C']
```

Out[87]:
```
6
```

Tra gli attributi di `dict`, i metodi keys() e values() forniscono (ovvero restituiscono, in inglese *return*), rispettivamente, le chiavi e i valori di un dizionario.

In [88]:
```
list(atomic_number.keys())
```

Out[88]:
```
['H', 'He', 'C', 'Fe']
```

In [89]:
```
list(atomic_number.values())
```

Out[89]:
```
[1, 2, 6, 26]
```

In [90]:
```
list(atomic_number.items())
```

Out[90]:
```
[('H', 1), ('He', 2), ('C', 6), ('Fe', 26)]
```

L'operatore in consente di verificare se una key è contenuta in un dizionario:

In [91]:
```
'He' in atomic_number
```

Out[91]:
```
True
```

In [92]:
```
'Au' in atomic_number
```

Out[92]:
```
False
```

Aggiorniamo un dizionario:

In [93]:
```
D_new = {'e': 5, 'f': 6}
D.update(D_new)
D
```

Out[93]:
```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
```

In [94]:
```
D.clear()
D
```

Out[94]:
```
{}
```