Liste, dizionari, set e tuple

Andrea Passerini
passerini@disi.unitn.it

Informatica

Liste

Descrizione

- Una lista è una **sequenza** di oggetti qualunque (anche di tipo diverso, anche altre liste)

```
>>> l = ["AG01", 857, ["PAZ","Piwi"]]
```

- Essendo una sequenza, condivide le operazioni su sequenza viste per le stringhe

```
>>> len(l)
3
```

- La lista è un tipo **mutabile**: può essere allungata ed accorciata, e si possono modificare i suoi elementi

```
>>> l[0] = "AGO1_HUMAN"
>>> l
["AG01_HUMAN", 857, ["PAZ","Piwi"]]
```

Operazioni su liste

Operatori

```
>>> l = ["AGO1","AKAP1"]
>>> l + ["ACO1","AGO3"]
["AGO1","AKAP1","ACO1","AGO3"]

>>> l + []       # [] indica una lista vuota
 ["AGO1","AKAP1"]

>>> l * 2
["AGO1","AKAP1","AGO1","AKAP1"]

>>> "AKAP1" in l
True
>>> ["AGO1","AKAP1"] in l # vale solo per singoli e
False
```

Operazioni su liste

Indicizzazione e sottolista

```
>>> l = [1,2,3,4,5,6]
>>> l[2]
3
>>> l[-1]
6
>>> l[:]
[1, 2, 3, 4, 5, 6]
>>> l[3:]
[4, 5, 6]
>>> l[:-2]
[1, 2, 3, 4]
```

Operazioni su liste

Matrici

- Una lista i cui elementi siano tutte liste implementa una *matrice*
- Si possono recuperare tramite uno o due indici righe o singoli elementi della matrice

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix[2]
[7, 8, 9]
>>> matrix[2][0]
7
>>> matrix[2][0:2]
[7, 8]
```

Modifica di liste

Modifica con assegnazione ad indici

- Essendo oggetti mutevoli, è possibile modificare il contenuto di una lista

```
>>> l = [0,0,0,0]
>>> p = l
>>> l[1] = 1
>>> l
[0,1,0,0]
>>> p
[0,1,0,0]
>> l[2] += 1
>>> p
[0,1,1,0]
```

- Poiché l'oggetto cambia, tutte le variabili che vi si riferiscono "vedono" il cambiamento

Modifica di liste

Modifica tramite la funzione `del`

- La funzione `del` permette di cancellare elementi o sottoliste specificando posizione o gamma di posizioni

```
>>> l = [1,2,3,4,5,6]
>>> del(l[2])
>>> l
[1, 2, 4, 5, 6]
>>> del(l[-1])
>>> l
[1, 2, 4, 5]
>>> del(l[2:4])
>>> l
[1, 2]
>>> del(l[:])
>>> l
[]
```

Metodi di lista

Metodi di modifica

- Come i tipi stringa, anche le liste hanno una serie di metodi da eseguire
- Essendo oggetti mutabili, molti di questi metodi modificano direttamente l'oggetto

append aggiunge un elemento in fondo alla lista

```
>>> l = [1,2,3,4,5,6]
>>> l.append(7)
>>> l
[1, 2, 3, 4, 5, 6, 7]
```

extend estende la lista attaccando in fondi tutti gli elementi di una lista

```
>>> l.extend([8,9])
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Metodi di lista

Metodi di modifica

insert Inserisce un elemento prima di una certa posizione

```
>>> l = [1,2,3,4,5,6]
>>> l.insert(3,3.5)
>>> l
[1, 2, 3, 3.5, 4, 5, 6]
```

Metodi di lista

Metodi di modifica

sort Ordina la lista (assume l'esistenza di una funzione di ordinamento tra elementi, si usa per liste di elementi dello stesso tipo)

```
>>> l = ["basso","medio","alto"]
>>> l.sort()
>>> l
['alto', 'basso', 'medio']
```

reverse Riordina la lista dall'ultimo al primo elemento

```
>>> l.reverse()
>>> l
['medio', 'basso', 'alto']
```

Documentazione

dir e help

- E' sempre possibile visualizzare l'elenco dei metodi di lista disponibili con la funzione `dir`

```
>>> dir([])
[..............., 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

- E recuperare informazioni su uno specifico metodo con la funzione `help`

```
>>> help([].remove)
Help on built-in function remove:

remove(...)
    L.remove(value) -- remove first occurrence
                       of value
```

List comprehensions

Operazioni sugli elementi di una lista

- Il costrutto nominato *list comprehension* è un tipo di espressione caratteristica del Python che permette di creare una lista come risultato di un'operazione da fare sugli elementi di un'altra lista

```
>>> L = [0,1,2,3,4]
>>> [i+1 for i in L]
[1, 2, 3, 4, 5]
```

- Il costrutto è composto da:
  - Un oggetto di tipo sequenza su cui iterare (e.g. la lista `L`)
  - Una variabile (o più , vedremo) che raccolga di volta in volta gli elementi (e.g. `i`)
  - Una espressione che faccia qualche operazione che coinvolga l'elemento cui la variabile si riferisce (e.g. `i+1`)
  - Le parole riservate `for` ed `in`, e le parentesi quadre a delimitazione

List comprehensions

Operazioni sugli elementi di una lista

- Ad esempio può essere usato per estrarre una colonna da una matrice:

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> [ row[2] for row in matrix]
[3, 6, 9]
```

- L'operazione sugli elementi può anche essere una qualsiasi funzione o metodo compatibile con il loro tipo

```
>>> [ str.upper() for str in ["a","b","c"]]
['A', 'B', 'C']
```

- E' possibile selezionare gli elementi della lista da processare tramite una condizione (parola chiave `if`)

```
>>> [i for i in [1,3,4,5,6,2,8,9] if i % 2 != 0]
[1, 3, 5, 9]    # solo numeri dispari
```

List comprehensions

Operazioni sugli elementi di una lista

- Il costrutto può essere usato con oggetti che siano *sequenze*, non solo con liste

```
>>> [ r for r in "AHHDCCEDGGTA" if r in "HC"]
['H', 'H', 'C', 'C']
```

- Può ovviamente essere combinato con metodi o funzioni che processino liste

```
>>> "".join([ r for r in "AHHDCCEDGGTA"
... if r in "HC"])
'HHCC'
```

- E' uno dei costrutti più utili del linguaggio, ne vedremo molti esempi

Dizionari

Descrizione

- Un dizionario è una collezione *non ordinata* di oggetti indirizzati tramite *chiavi* (invece che per posizione come nelle mappe)

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
```

- Un dizionario è quindi una **mappa** da chiavi a valori, e i suoi oggetti si recuperano specificandone la chiave corrispondente

```
>>> D["medio"]
1
```

Dizionari

Creazione

- Un dizionario può essere creato specificandone l'insieme di coppie `chiave:valore` separate da virgole e delimitato da parentesi graffe

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
```

- E' comune creare un dizionario vuoto per poi riempirlo inserendo coppie di volta in volta

```
>>> D = {}
```

Dizionari sparsi

Le chiavi di un dizionario non devono necessariamente essere stringhe, basta che siano oggetti *immutabili* (e.g. numeri o tuple). E' così possibile creare ad esempio vettori o array multidimensionali *sparsi* (con pochi indici con valore specificato, assumendo ad es. una valore di default come 0 per gli altri)

```
>>> D = {10 : 2.3, 50 : 1.5, 223 : -3}
>>> D = {(0,0) : "rosso", (0,1) : "verde" ,
... (100,200) : "blu"}
```

Dizionari

Ricerca

- Verifica della presenza di una certa chiave (operatore `in`)

```
>>> "alto" in D
True
>>> "altissimo" in D
False
```

- metodo `get`: recupero di un oggetto specificando chiave e valore di default da restituire se non presente (`None` se non specificato)

```
>>> D.get("alto")
2
>>> D.get("altissimo") # None non viene stampato
>>> D.get("altissimo", "alto")
'alto'
```

Dizionari

Metodi di esplorazione

len restituisce il numero di elementi (coppie chiave,valore) contenuti nel dizionario

```
>>> D = {"a" : 0, "b" : 1, "c" : 2}
>>> len(D)
3
```

keys restituisce una vista sulle chiavi nel dizionario

```
>>> D.keys()
dict_keys(['a', 'c', 'b'])
```

values restituisce una vista sui valori nel dizionario

```
>>> D.values()
dict_values([0, 2, 1])
```

items restituisce una vista delle coppie chiave,valore (come tuple) nel dizionario

```
>>> D.items()
dict_items([('a', 0), ('c', 2), ('b', 1)])
```

Modifica di dizionari

Oggetti mutabili

- Il modo più semplice per recuperare un oggetto è accedendovi per chiave (come si farebbe in una lista con l'indice)

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> D["alto"]
2
```

- Come le liste, i dizionari sono oggetti *mutabili*. E' quindi modificare un oggetto associato ad una certa chiave

```
>>> D["alto"] = 3
>>> D
{'alto': 3, 'basso': 0, 'medio': 1}
```

Modifica di dizionari

Oggetti mutabili

- A differenza delle liste se si assegna un valore ad una chiave non presente, la coppia `chiave:valore` viene aggiunta al dizionario:

```
>>> D["altissimo"] = 3
>>> D
{'alto': 3, 'basso': 0, 'altissimo': 3, 'medio': 1}
```

- In una lista, l'assegnazione di un valore ad un indice fuori dimensione genera invece un errore (si deve usare `append`)

```
>>> L = ["a","b","c"]
>>> L[3] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Modifica di dizionari

Modifica tramite la funzione `del`

- In modo analogo alle liste, la funzione `del` permette di cancellare elementi (ma non sottoinsiemi di elementi) specificandone la chiave

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> del(D["medio"])
>>> D
{'basso': 0, 'alto': 2}
```

Comando `del`

Rimozione di variabili

- La funzione `del` applicata ad una variabile la elimina, ma non cancella l'oggetto a cui si riferisce

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> D2 = D
>>> del(D)
>>> D
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'D' is not defined
>>> D2
{"basso" : 0, "medio" : 1, "alto" : 2}
```

I dizionari sono mappe NON sequenze

Le operazioni su sequenza non funzionano

- Le operazioni che presuppongono un ordinamento degli elementi non funzionano nei dizionari:

```
>>> D = {"a" : 0, "b" : 1, "c" : 2}
>>> D2 = {"d" : 3, "e" : 4}
>>> D + D2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'dict' and 'dict'
```

Documentazione

dir e help

- Al solito, `dir` permette di visualizzare l'elenco dei metodi di dizionario disponibili

```
>>> dir({})
[..................., 'clear', 'copy', 'fromkeys',
'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

- e informazioni su uno specifico metodo si possono recuperare con la funzione `help`

```
>>> help({}.fromkeys)
Help on built-in function clear:

clear(...) method of builtins.dict instance
    D.clear() -> None.  Remove all items from D.
```

Sets

Descrizione

- Un set è una collezione non ordinata di oggetti.

```
>>> S = {"1dsf","3rsf","4rews","1ewae"}
```

- E' come un dizionario che contenga solo chiavi e non valori
- Lo stesso oggetto non può comparire più di una volta, per cui creare un set da un elenco equivale ad eliminarne i duplicati

Sets

Creazione

- Un set può essere creato come elenco di oggetti tra parentesi graffe

```
>>> S = {"1dsf","3rsf","4rews","1ewae"}
```

- Un set può essere creato passando una lista alla funzione `set`

```
>>> S = set(["1dsf","3rsf","1dsf","4rews","3rsf"])
>>> S
{'4rews', '1dsf', '3rsf'}
```

- Un set vuoto può essere creato usando `set` senza argomenti (`{}` crea un dizionario):

```
>>> S = set()
```

Sets

Ricerca

Si usa l'operatore `in`

```
>>> S = set(["1dsf","3rsf","1dsf","4rews","3rsf"])
>>> "3rsf" in S
True
```

Inserimento

- Metodo `add` per aggiungere singoli elementi

```
>>> S = set(["1dsf","3rsf","1dsf","4rews","3rsf"])
>>> S.add("2rsa")
>>> S
{'3rsf', '1dsf', '4rews', '2rsa'}
```

- Metodo `update` per aggiungere elenchi di elementi

```
>>> S.update(["1aaa","1asd","1aaa"])
>>> S
{'2rsa', '4rews', '1aaa', '3rsf', '1asd', '1dsf'}
```

Cancellazione

- Il metodo `discard` *non* da errore se l'elemento da cancellare non c'è

```
>>> S = set(["1dsf","3rsf","1dsf","4rews","3rsf"])
>>> S.discard("1dsf")
>>> S
{'3rsf', '4rews'}
>>> S.discard("1aaa")
```

- Il metodo `remove` *da* errore se l'elemento da cancellare non c'è

```
>>> S.remove("1aaa")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '1aaa'
```

Sets

Operazioni tra Set

- Unione

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}
```

- Intersezione

```
>>> A.intersection(B)
{4, 5}
```

- Differenza

```
>>> A.difference(B)
{1, 2, 3}
```

Attenzione

Le operazioni tra set restituiscono un nuovo set, *non* modificano i set originali

Tuple

Descrizione

- Una tupla è una collezione ordinata di oggetti (racchiusi da parentesi tonde)

```
>>> t = ("C",34)
```

- Come stringhe e liste, è un tipo *sequenza*, accessibile per indice e su cui funzionano le operazione definite su sequenze

```
>>> t[1]
34
```

- Come le stringhe, è *immutabile* e non è possibile modificarla

```
>>> t[1] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
                                        assignment
```

Operazioni su tuple

Operatori

```
>>> t = (1,-5,4)
>>> t + (1,2)
(1, -5, 4, 1, 2)
>>> t + ()
(1, -5, 4)
>>> t * 2
(1, -5, 4, 1, -5, 4)
```

Operazioni su tuple

Tuple con un solo elemento

```
>>> t = (1,-5,4)
>>> t + (1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple
                    (not "int") to tuple
>>> t + (1,)
(1, -5, 4, 1)
```

Nota

- Anche un'espressione può essere racchiusa tra parentesi
- Per distinguere una tupla con un solo elemento da un'espressione, va aggiunta una virgola dopo l'elemento (e.g. `(1,)`)

Operazioni su tuple

Indicizzazione e sottotupla

```
>>> t = (1,2,3,4,5,6)
>>> t[2]
3
>>> t[-1]
6
>>> t[:]
(1, 2, 3, 4, 5, 6)
>>> t[3:]
(4, 5, 6)
>>> t[:-2]
(1, 2, 3, 4)
```

Immutabilità di tuple

Oggetti immutabili

- Come le stringhe, le tuple sono oggetti immutabili

```
>>> t = (12, "abc", [1,2,3])
>>> t[2] = "f"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
                                        assignment
```

- Tale immutabilità non si applica ai contenuti della tupla, per cui i suoi oggetti possono essere modificati *se mutabili*

```
>>> t[2].append(4)
>>> t
(12, 'abc', [1, 2, 3, 4])
>>> t[1][2] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
                                      assignment
```

Perché le tuple?

Utilità delle tuple

- Le tuple sono semplicemente liste immutabili
- Non hanno quindi nessun metodo (ma possono essere manipolate per creare nuove tuple con gli operatori)
- La loro utilità è data dal fatto di essere immutabili: possono essere passate tra parti diverse di un programma essendo sicuri che non verranno modificate (e.g. in una certa posizione ci sarà sempre lo stesso oggetto)
- Possono essere usate dove c'è bisogno di oggetti immutabili, ad esempio come indici di dizionari

Perché le tuple?

Utilità delle tuple

- Le tuple sono spesso usate per semplificare le operazioni di assegnazione a più variabili

```
>>> (x, y, z) = (1.2, 3, 4.5)
>>> y
3
```

- Dove non ambiguo, e' possibile specificare una tupla anche senza parentesi, semplificando ulteriormente la notazione

```
>>> x, y, z = 1.2, 3, 4.5
>>> y, z
(3, 4.5)
```

Perché le tuple?

Utilità delle tuple

- Vari metodi e funzioni restituiscono tuple in uscita, ad esempio il metodo `items` dei dizionari

```
>>> D = {"C" : 8, "H" : 12}
>>> D.items()
dict_items([('C', 8), ('H', 12)])
```

- Il costrutto list comprehension può gestire tuple di variabili invece di variabili singole:

```
>>> [ "%s = %d" % (k,v) for (k,v) in
... D.items()]
['C = 8', 'H = 12']
```