Collections

Corso Programmazione Python 2024

Modulo 3

Luca Di Pietro Martinelli

Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)

Tipi di dato semplici e strutturati

I tipi di dato possono essere classificati in:

- tipi semplici
- tipi strutturati

Un **tipo semplice** è composto da valori che non possono essere "scomposti" in valori più semplici ai quali sia possibile accedere attraverso operatori o funzioni del linguaggio.

Esempi di tipi semplici del linguaggio Python (e di altri linguaggi) sono i numeri interi, i numeri frazionari e i valori Booleani.

Un **tipo strutturato** è invece composto da valori che sono a loro volta collezioni o sequenze di valori più semplici.

Le stringhe sono un esempio di tipo strutturato: sono infatti composte da sequenze ordinate di caratteri a cui è possibile accedere individualmente.

Tipi di dato strutturati: le Collections

Oltre alle stringhe, quattro dei principali tipi strutturati del linguaggio Python sono:

- le liste, che consentono di rappresentare collezioni ordinate e modificabili di valori qualsiasi
- le tuple, che consentono di rappresentare collezioni ordinate e non modificabili di valori qualsiasi
- i set, che permettono di utilizzare collezioni non ordinate ma modificabili di soli valori immutabili
- i dizionari, che permettono di lavorare con collezioni ordinate e modificabili di valori qualsiasi.

Tipi di dato mutabili e immutabili

Stringhe, liste, tuple, set e dizionari, pur gestendo tutte collezioni di elementi (sono infatti definite ***Collections***) presentano dunque un'importante differenza tra loro.

Mentre attraverso l'istruzione di assegnamento e l'operatore di indicizzazione è possibile modificare i singoli elementi di una lista e quelli di un dizionario, non è invece possibile modificare né i singoli caratteri delle stringhe né i singoli elementi di una tupla. Discorso a parte è invece quello dei set, che offre una mutabilità limitata.

Per tale motivo:

- le stringhe e le tuple sono dette collezioni immutabili
- le liste, i set e i dizionari sono dette collezioni mutabili

Il tentativo di modificare un elemento di una stringa o di una tupla produce di fatto un errore nell'esecuzione del programma!

Operazioni comuni alle Collections

Alcuni operatori e alcune funzioni built-in che operano su collezioni ordinate possono essere applicati a stringhe, liste, tuple e dizionari, ovvero nello specifico:

Funzioni built-in:

- ✓ `len`
- ✓ `max`
- ✓ `min`
- ✓ `sum`

Operatori:

- ✓ `in`
- ✓ `not in`
- ✓ Indicizzazione
- ✓ Slicing
- ✓ Concatenazione

Lista

Il tipo di dato Lista di Python, come il nome suggerisce, ci permette di raggruppare più elementi tra di loro.

Le liste in Python sono molto potenti: possono infatti contenere al loro interno diversi tipi di dato anche tutti assieme, e i vari elementi vengono ordinati in base a un indice proprio della lista, in modo da semplificarne l'accesso.

Volendo ad esempio rappresentare in modo compatto le coordinate di un punto in uno spazio a tre dimensioni, è possibile utilizzare una singola variabile di tipo `list` invece che tre variabili distinte di tipo `float`.

Tipo di dato list

Una lista si rappresenta nei programmi Python con dei valori:

- inseriti all'interno di una sequenza ordinata
- scritti tra parentesi quadre
- separati da virgole

Esempi:

```
[7, -2, 4]
[-5.3, 6, True]

[]
```

→ lista composta da tre numeri interi

→ lista composta da un numero reale, un numero intero e un valore booleano

→ lista vuota

Tipo di dato list

Le liste, come i valori di qualsiasi tipo di dato Python, sono espressioni e possono quindi essere scritte in qualsiasi punto di un programma nel quale possa comparire una espressione.

È dunque possibile:

- stampare una lista con l'istruzione `print`
  - Esempio: `print([7, -2, 4])`
- assegnare una lista a una variabile
  - Esempio: `v = [7, -2, 4]`

Tipo di dato list

I valori degli elementi di una lista possono a loro volta essere indicati attraverso espressioni.

Per esempio, dopo l'esecuzione della seguente sequenza di istruzioni:

```
x = 2
y = -5
z = [x, y ** 2 + 1, x == 3]
```

la variabile `z` sarà associata alla lista `[2, 26, False]`.

È possibile anche inserire liste come elementi di una lista (liste annidate):

```
[-3, "abcd", ["a", 'b'], 10]
```

Componendo una lista con tutte liste annidate come suoi elementi avremo realizzato una matrice:

$$\begin{pmatrix} -3 & 1 & 4 \\ 2 & 5 & -1 \end{pmatrix}$$ => `[[-3, 1, 4], [2, 5, -1]]`

List: operatori

Sintassi	Descrizione
lista1 == lista2	Uguale a
lista1 != lista2	Diverso da
espressione in lista	Verifica della presenza di un valore in lista
espressione not in lista	Verifica dell'assenza di un valore in lista
lista1 + lista2	Concatenazione
lista[indice]	Indicizzazione: accesso a un elemento
lista[indice1:indice2]	*Slicing* (sezionamento): accesso a un sottoinsieme di elementi di una lista

List: operatori di confronto

Gli operatori == e `!=` consentono di scrivere espressioni condizionali (il cui valore sarà `True` o `False`) consistenti nel confronto tra due liste.

Sintassi:

```
lista_1 == lista_2
lista_1 != lista_2
```

dove `lista_1` e `lista_2` indicano espressioni che abbiano come valore una lista.

Semantica:
Due liste sono considerate identiche se sono composte dallo stesso numero di elementi e se ogni elemento ha valore identico a quello dell'elemento che si trova nella stessa posizione nell'altra lista.

List: operatori in e not in

Gli operatori `in` e `not in` consentono di scrivere espressioni condizionali che hanno lo scopo di verificare se un certo valore sia presente o meno all'interno di una lista.

Sintassi:

```
espressione in lista
espressione not in lista
```

dove espressione indica una qualsiasi espressione Python.

Semantica:

- Se il valore di espressione è presente tra gli elementi di lista, allora l'operatore `in` restituisce il valore `True`; in caso contrario, restituisce `False`.
- Il comportamento dell'operatore `not in` è l'opposto di quello per `in`.
- La ricerca non viene estesa agli elementi di eventuali liste annidate all'interno di lista.

List: operatore di concatenazione

L'operatore di concatenazione per le liste è analogo al corrispondente operatore del tipo di dato stringa.

Sintassi:

```
lista_1 + lista_2
```

Semantica:

La concatenazione restituisce una nuova lista composta dagli elementi di `lista_1` seguiti da quelli di `lista_2`, disposti nello stesso ordine in cui si trovano nelle due liste. Le liste originali non vengono modificate.

List: operatore di indicizzazione

L'operatore di indicizzazione consente di accedere a ogni singolo elemento di una lista, per mezzo dell'indice corrispondente.

Sintassi:

```
lista[indice]
```

dove `indice` deve essere una espressione il cui valore sia un intero compreso tra 0 e la lunghezza della lista meno uno. Indicando un valore negativo la lista viene scorsa in senso contrario (ad esempio con `-1` viene restituito l'ultimo elemento della lista).

Semantica:
il risultato è il valore dell'elemento di lista il cui indice è pari al valore di indice. Se il valore di `indice` non corrisponde a una delle posizioni della lista si otterrà un messaggio di errore.

List: operatore di indicizzazione

Gli elementi di una lista possono essere valori di tipi qualsiasi, quindi anche strutturati, come stringhe o altre liste.

L'operatore di indicizzazione consente di accedere anche agli elementi di strutture annidate.

Se `s` è una variabile a cui è stata assegnata una lista e l'elemento di indice `i` della lista è a sua volta una sequenza (lista o stringa), sarà possibile accedere all'elemento di indice `j` di quest'ultima con la seguente sintassi: `s[i][j]`.

List: operatore di indicizzazione

L'operatore di indicizzazione consente anche di modificare i singoli elementi di una lista attraverso un'istruzione di assegnamento.

Sintassi:

```
lista[indice] = espressione
```

dove `lista` indica il nome di una variabile alla quale sia stata in precedenza assegnata una lista, mentre espressione indica una qualsiasi espressione Python

Semantica:

l'elemento di `lista` nella posizione corrispondente a indice viene sostituito dal valore di espressione.

List: operatore di *slicing*

L'operatore di *slicing* restituisce una lista composta da una sottosequenza della lista a cui viene applicato.

Sintassi:
`lista[indice_1:indice_2]`

dove `indice_1` e `indice_2` sono espressioni i cui valori devono essere numeri interi compresi tra 0 e la lunghezza di lista.

Utilizzi particolari:
`lista[indice_1:]` => lista con elementi da `indice_1` all'ultimo
`lista[:indice_2]` => lista con elementi dal primo a `indice_2 - 1`
`lista[:]` => lista intera (copia della lista originale)

Semantica:

il risultato è una lista composta dagli elementi di `lista` aventi indici da `indice_1` a `indice_2 - 1` (si noti che l'elemento avente indice pari a `indice_2` non viene incluso nel risultato). Anche in questo caso la lista originale non viene modificata.

List: aggiunta e rimozione elementi

Data la mutabilità delle liste è anche possibile modificarle aggiungendo singoli elementi alla fine della struttura dati tramite la funzione `append`.

Sintassi:

```
lista.append(elemento)
```

dove `lista` è la lista su cui operare ed `elemento` è l'elemento che viene aggiunto al termine degli elementi già presenti all'interno della lista.

È possibile poi anche rimuovere elementi da una lista attraverso i seguenti modi:

- `lista.remove(valore_elemento)` => rimuove il primo elemento in base al suo valore
- `lista.pop(indice_elemento)` => rimuove un elemento in base al suo indice
- `del lista[indice_elemento]` => rimuove un elemento in base al suo indice
- `del lista[indice_a:indice_b]` => rimuove elementi tramite lo *slicing*

Funzione range

La funzione `range` è molto utile in quanto restituisce una sequenza immutabile di numeri tra il numero intero di partenza dato e il numero intero finale (non compreso).

Sintassi:

```
range(start, stop, step)
```

- `start` indica il valore di partenza della sequenza (opzionale, se non indicato di default è a 0)
- `stop` indica il valore a cui si interrompe la generazione della sequenza (e non è compreso)
- `step` indica l'incremento tra ogni numero intero della sequenza (opzionale, di default è a 1).

List: funzioni built-in

Sintassi	Descrizione
`len(lista)`	Restituisce la lunghezza di una lista
`min(lista)`	Restituisce l'elemento più piccolo in una lista di numeri e il primo in ordine alfabetico con stringhe
`max(lista)`	Restituisce l'elemento più grande in una lista di numeri e il primo in ordine alfabetico con stringhe
`sum(lista)`	Restituisce la somma in una lista di numeri
`list(range(a))`	a deve essere un intero; se a > 0 restituisce la lista [0, 1,. . . ,a-1], altrimenti restituisce una lista vuota
`list(range(a, b))`	a e b devono essere interi: se a < b, restituisce la lista [a,a+1,. . . ,b-1], altrimenti restituisce una lista vuota

Tipo di dato tuple

Come il tipo di dato `list` visto precedentemente, anche il tipo `tuple` di Python rappresenta un sequenza di elementi che però a differenza del precedente è immutabile.

Una tupla si rappresenta nei programmi Python con dei valori:

- inseriti all'interno di una sequenza ordinata
- scritti tra parentesi tonde oppure senza alcuna parentesi
- separati da virgole

Esempi:

```
(7, -2, 4)
-5.3, 6, True
```

→ tupla composta da tre numeri interi

→ tupla composta da un numero reale, un numero intero e un valore booleano

Tuple: funzioni built-in

Sintassi	Descrizione
`len(tupla)`	Restituisce la lunghezza di una tupla
`min(tupla)`	Restituisce l'elemento più piccolo in una tupla di numeri e il primo in ordine alfabetico con stringhe
`max(tupla)`	Restituisce l'elemento più grande in una tupla di numeri e il primo in ordine alfabetico con stringhe
`sum(tupla)`	Restituisce la somma in una tupla di numeri
`tuple(range(a))`	a deve essere un intero; se a > 0 restituisce la tupla [0, 1,. . . ,a-1], altrimenti restituisce una tupla vuota
`tuple(range(a, b))`	a e b devono essere interi: se a < b, restituisce la tupla [a,a+1,. . . ,b-1], altrimenti restituisce una tupla vuota

Tipo di dato set

Allo stesso modo dei tipi di dato `list` e `tuple` anche il tipo `set` di Python permette di creare una collezione di elementi. Le caratteristiche particolari di questa struttura dati sono date dal fatto che non è ordinato e che non può contenere duplicati al suo interno.

Un set si rappresenta nei programmi Python con dei valori:

- scritti tra parentesi graffe
- separati da virgole

Esempi:

```
{"abc", "bcd", "cde"}
{"abc", "bcd", "bcd"}
```

→ set composto da tre stringhe
→ set inizializzato con tre elementi ma che ne contiene due (`"bcd"` è duplicato)

Set: funzioni built-in e metodi

Sintassi	Descrizione
`len(set)`	Restituisce la lunghezza di un set
`min(set)`	Restituisce l'elemento più piccolo in un set di numeri e il primo in ordine alfabetico con stringhe
`max(set)`	Restituisce l'elemento più grande in un set di numeri e il primo in ordine alfabetico con stringhe
`sum(set)`	Restituisce la somma in un set di numeri
`set(range(a))`	a deve essere un intero; se a > 0 restituisce il set {0, 1,. . . ,a-1}, altrimenti restituisce un set vuoto
`set(range(a, b))`	a e b devono essere interi: se a < b, restituisce il set {a,a+1,. . . ,b-1}, altrimenti restituisce un set vuota
`set.add(element)`	Aggiunge un elemento al set
`set.remove(element)`	Rimuove un elemento dal set se presente

Tipo di dato dict

Il tipo `dict` (dizionario) è un tipo built-in di Python mutabile e ordinato che contiene un insieme di elementi formati da una coppia chiave-valore.

Una volta definito un dizionario è sufficiente utilizzare la chiave (che deve essere necessariamente univoca) per ottenere il valore a essa associato.

Un dizionario si rappresenta nei programmi Python con una serie di elementi:

- inclusi all'interno di parentesi graffe
- separati da virgole
- ciascuno composto da una chiave e un valore separati dai due punti (:)

Esempi:

```
{'a': 1, 'b': 2, 'c': 3}
{20: ['a', 'b'], 28: ['c', 'd']}
```

→ dizionario composto da tre elementi
→ dizionario composto da due elementi con chiave intera e valore pari a lista

Tipo di dato dict

Una volta creato un dizionario, per ottenere lo specifico valore associato a una chiave è possibile utilizzare la seguente sintassi:

```
d = {'a': 1, 'b': 2, 'c': 3}
print(d['a'])  # stampa il valore associato alla chiave 'a', ovvero 1
```

Nel caso si specifichi una chiave inesistente Python restituisce un KeyError. È tuttavia possibile in questo senso usare l'operatore `in` (o `not in`) per verificare se una chiave è presente nel dizionario:

```
print('a' in d)  # stampa True poiché la chiave 'a' è presente in d
```

È inoltre possibile aggiungere o modificare elementi usando la sintassi

```
dizionario[chiave] = valore
```

e rimuoverne usando la sintassi

```
del dizionario[chiave]
```

Dizionari: funzioni built-in e metodi

Sintassi	Descrizione
`len(`dizionario`)`	Restituisce la lunghezza di un dizionario
dizionario`.items()`	Restituisce gli elementi di un dizionario come un insieme di tuple
dizionario`.keys()`	Restituisce le chiavi di un dizionario
dizionario`.values()`	Restituisce i valori di un dizionario
dizionario`.get(chiave)`	Restituisce il valore corrispondente a chiave se presente
dizionario`.pop(chiave)`	Rimuove e restituisce il valore corrispondente a chiave se presente
dizionario`.update(`dizionario2`)`	Aggiunge gli elementi del dizionario2 a quelli del dizionario
dizionario`.copy()`	Crea e restituisce una copia del dizionario
dizionario`.clear()`	Rimuove tutti gli elementi del dizionario

Differenze tra strutture dati

	Lista	Tupla	Set	Dizionario
Ordinamento	Ordinata	Ordinata	Non ordinato	Ordinato Non ordinato prima di Python 3.7
Indicizzazione	Indicizzata	Indicizzata	Non indicizzato	Tramite chiavi
Mutabilità	Mutabile	Immutabile	Mutabile Solo aggiunta e rimozione	Mutabile
Duplicati ammessi	Sì	Sì	No	Sì Solo nei valori
Tipi ammessi	Mutabili e immutabili	Mutabili e immutabili	Solo immutabili	Solo immutabili Solo nelle chiavi

Istruzione iterativa for

Python include una versione alternativa dell'istruzione iterativa `while`: l'istruzione `for`.

L'istruzione `for` consente di esprimere un solo tipo di iterazione che consiste nell'accedere a tutti gli elementi di una *Collection* (stringa, lista, tupla, set o dizionario).

L'accesso avviene dal primo all'ultimo elemento e non è possibile modificare tale ordine.

Sintassi:

```
for v in s
    sequenza_di_istruzioni
```

- `v` è il nome di una variabile
- `s` è una espressione avente come valore una *Collection*
- `sequenza_di_istruzioni` è una sequenza di una o più istruzioni qualsiasi che devono rispettare la regola sui rientri già vista per l'istruzione `while`

Confronto tra while e for

Non è possibile usare l'istruzione `for` con lo schema visto in precedenza per eseguire operazioni sugli elementi di una collezione che richiedano l'uso esplicito degli indici, come per esempio:

- la modifica di un elemento di una lista attraverso una istruzione di assegnamento `lista[k] = valore`
- l'accesso a più di un elemento di una lista o di una stringa, ad esempio per confrontare i valori di due elementi adiacenti con un'espressione condizionale come la seguente `lista[k] != lista[k + 1]`

```
k = 0
while k < len(lista):
    lista[k] = 0
    k = k + 1
```

Tutti gli elementi della lista impostati correttamente a 0

```
for elemento in lista:
    elemento = 0
```

Gli elementi della lista non vengono modificati poiché `elemento` è solo una copia del valore

For: funzione enumerate

Grazie alla funzione `enumerate` è possibile usare l'istruzione for per accedere agli elementi di una collezione usando una variabile come indice.

Sintassi:

```
for indice, elemento in enumerate(lista):
    sequenza_di_istruzioni
```

- `indice` è l'indice corretto associato a elemento
- per eseguire l'accesso a un elemento bisogna fare riferimento a `lista` di `enumerate`, non a `elemento`

For: funzione range

La funzione `range` è molto utile nell'esecuzione dei cicli `for` in quanto la sequenza immutabile che genera è utilizzabile per eseguire delle iterazioni.

Esempio:

```
for i in range(2, 10, 2):
    print(i)
```

Output:

```
2
4
6
8
```

Metodo split

Il metodo della classe `string` denominata `split` è molto utile per l'elaborazione di stringhe.

Sintassi:

```
stringa.split()
```

dove `stringa` indica una variabile avente per valore una stringa.

La funzione `split` suddivide una stringa in corrispondenza dei caratteri di spaziatura (incluso il *newline*) e restituisce le corrispondenti sottostringhe all'interno di una lista, senza includere i caratteri di spaziatura. La stringa originale non viene modificata.

Se si desidera suddividere una stringa in corrispondenza di una sequenza di uno o più caratteri specifici, tale sequenza dovrà essere indicata (sotto forma di una stringa) come argomento di `split`

Sintassi:

```
stringa.split(caratteri)
```

Metodo join

Il metodo `join` esegue invece l'inverso di quello che fa la funzione `split`: permette infatti di unire gli elementi di una collezione attraverso l'utilizzo di una stringa che funge da separatore.

Sintassi:

```
separatore.join(collezione)
```

La funzione dunque prende in input una collezione iterabile (list, tuple, set oppure dict) e unisce tutti i suoi elementi applicando tra ciascuno di essi il separatore specificato e ottenendo così una stringa in output.