

[FIM] FONDAMENTI DI INFORMATICA per medicina e chirurgia high tech

P09: Iteratori

Dott. Giorgio De Magistris

demagistris@diag.uniroma1.it

CORSO DI LAUREA IN MEDICINA E CHIRURGIA HIGH TECH



SAPIENZA
UNIVERSITÀ DI ROMA

I3S

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA

DIAG

DIPARTIMENTO DI INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE

TUTTI I DIRITTI RELATIVI AL PRESENTE MATERIALE DIDATTICO ED AL SUO CONTENUTO SONO RISERVATI A SAPIENZA E AI SUOI AUTORI (O DOCENTI CHE LO HANNO PRODOTTO). È CONSENTITO L'USO PERSONALE DELLO STESSO DA PARTE DELLO STUDENTE A FINI DI STUDIO. NE È VIETATA NEL MODO PIÙ ASSOLUTO LA DIFFUSIONE, DUPLICAZIONE, CESSIONE, TRASMISSIONE, DISTRIBUZIONE A TERZI O AL PUBBLICO PENA LE SANZIONI APPLICABILI PER LEGGE

Python For Loop

List items

```
for element in [1, 2, 3]:  
    print(element)
```

Tuple items

```
for element in (1, 2, 3):  
    print(element)
```

For Loop Statement

```
for target in expression:  
    body
```

Keys in a Dictionary

```
for key in {'one':1, 'two':2}:  
    print(key)
```

characters in a String

```
for char in "123":  
    print(char)
```

Lines in a File

```
for line in open("myfile.txt"):  
    print(line)
```

For Loop Statement

- **expression** viene convertita in un **iteratore**
- **body** viene eseguito per ogni elemento fornito dall'iteratore
- quando gli elementi sono esauriti, il ciclo termina

```
for target in expression:  
    body
```

Oggetti Iterabili

- il risultato di expression deve essere iterabile
- un oggetto si dice iterabile se ha un metodo **__iter__**
- il metodo **__iter__** restituisce un iteratore
- l'iteratore è un oggetto che ha il metodo **__next__** che fornisce il prossimo elemento
- quando l'iteratore è esausto (non ci sono più elementi) il metodo **__next__** lancia l'eccezione **StopIteration**

```
>>> l = [1,2,3,4]
>>> l.__iter__()
<list_iterator object at 0x7f72c1af3100>
>>> iter(l)
<list_iterator object at 0x7f72c126f2b0>
>>> it = iter(l)
>>> it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
4
>>> next(it)
Traceback (most recent call last):
  File "/usr/lib/python3.8/idlelib/run.py", line 559, in runcode
    exec(code, self.locals)
  File "<pyshell#8>", line 1, in <module>
StopIteration
```

Iteratori Custom

- Per creare un oggetto iterabile devo creare una classe che implementa il metodo **`__iter__`** che ritorna un iteratore
- Un iteratore è una classe che implementa il metodo **`__next__`**, che ritorna il prossimo elemento
- **ListWithIndex** è sia iterabile che iteratore (**`__iter__`** ritorna l'oggetto stesso), quindi posso usarlo all'interno di un ciclo `for`

```
class ListWithIndex:

    def __init__(self,l:List) -> None:
        self.index = 0
        self.data = l

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            elem = (self.index,self.data[self.index])
            self.index += 1
            return elem
        else:
            raise StopIteration()

l = ["a","b","c","d"]
for e in ListWithIndex(l):
    print(e)
```

Iteratori Custom

- **ListWithIndex** è un iteratore che restituisce gli elementi di una lista insieme al loro indice (ovvero restituisce tuple in cui il primo elemento è l'indice e il secondo è l'elemento corrispondente della lista)
- Il metodo **__init__** inizializza i dati e l'indice di partenza
- il metodo **__next__** ritorna l'elemento corrente insieme al suo indice e incrementa l'indice corrente. Se non ci sono più elementi (l'indice corrente è out of range) **__next__** lancia l'eccezione **StopIteration**

```
class ListWithIndex:

    def __init__(self,l:List) -> None:
        self.index = 0
        self.data = l

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            elem = (self.index,self.data[self.index])
            self.index += 1
            return elem
        else:
            raise StopIteration()

l = ["a","b","c","d"]
for e in ListWithIndex(l):
    print(e)
```

Generatori

- l'istruzione **yield** permette di creare iteratori molto rapidamente
- basta scrivere una normale funzione ed eseguire **yield** per restituire i dati
- una funzione con il comando **yield** crea un generatore (funziona esattamente come un iteratore)
- i metodi **__iter__()** e **__next__()** vengono creati automaticamente
- l'esempio mostra lo stesso iteratore della slide precedente implementato con **yield**

```
def list_with_index(l:List):  
    for i in range(len(l)):  
        yield (i,l[i])  
  
l = ["a", "b", "c", "d"]  
for e in list_with_index(l):  
    print(e)
```

Slides distribuite con Licenza Creative Commons (CC BY-NC-ND 4.0) Attribuzione - Non commerciale - Non opere derivate 4.0 Internazionale

PUOI CONDIVIDERLE ALLE SEGUENTI CONDIZIONI

(riprodurre, distribuire, comunicare o esporre in pubblico, rappresentare, eseguire e recitare questo materiale con qualsiasi mezzo e formato)

Attribuzione*

Devi riconoscere una menzione di paternità adeguata, fornire un link alla licenza e indicare se sono state effettuate delle modifiche. Puoi fare ciò in qualsiasi maniera ragionevole possibile, ma non con modalità tali da suggerire che il licenziante avalli te o il tuo utilizzo del materiale.

Non Commerciale

Non puoi utilizzare il materiale per scopi commerciali.

Non opere derivate

Se remixi, trasformi il materiale o ti basi su di esso, non puoi distribuire il materiale così modificato.

Divieto di restrizioni aggiuntive

Non puoi applicare termini legali o misure tecnologiche che impongano ad altri soggetti dei vincoli giuridici a questa licenza