University of Ferrara

Python

Iteratori, mappe

Iteratori

- Finora avrete forse notato che la maggior parte degli oggetti-contenitori può essere sottoposta a loop utilizzando l'istruzione for :

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Iteratori

- Dietro le quinte, l'istruzione for chiama la funzione iter() sull'oggetto contenitore.
- La funzione restituisce un oggetto iteratore che definisce il metodo __next__() il quale accede agli elementi all'interno del contenitore uno alla volta.
- Quando non ci sono più elementi, __next__() lancia l'eccezione StopIteration che comunica la fine del ciclo.
- Il metodo __next__() può essere richiamato utilizzando la funzione integrata next()

Iteratori

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Iteratori

- E’ semplice aggiungere un iteratore alla propria classe.
- Si definisce un metodo __iter__() il quale restituisce un oggetto con il metodo __next__() .
- Se una classe definisce il metodo __next__(), allora è sufficiente che __iter__() restituisca self
- Gli oggetti che hanno il metodo __iter__() si chiamano iterabili

Iteratori

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Iteratori

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

Generatori

- I Generatori sono uno strumento semplice e potente per creare iteratori.
- Vengono scritti come classiche funzioni ma usano l'istruzione yield ogni volta essi vogliano restituire dei dati.
- Ogni volta che next() viene chiamato su un generatore, esso riprende da dove era stato interrotto (ricorda tutti i valori dei dati e quale istruzione è stata eseguita per ultima).

Generatori

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
>>>
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Generatori

- Tutto ciò che può essere fatto con i generatori può essere fatto anche con gli iteratori basati su classi, come descritto nella sezione precedente.
- Quello che rende i generatori così compatti è il fatto che i metodi `__iter__()` e `__next__()` vengono creati in maniera automatica.

Generatori

- Un'altra caratteristica fondamentale è che le variabili locali e lo stato di esecuzione vengono salvati automaticamente tra le chiamate.
- Ciò rende la funzione più semplice da scrivere e molto più chiara di un approccio che utilizza variabili di istanza come self.index e self.data.
- Oltre alla creazione automatica del metodo e al salvataggio dello stato del programma, quando i generatori terminano, attivano automaticamente StopIteration. In combinazione, queste funzionalità semplificano la creazione di iteratori senza alcuno sforzo maggiore rispetto alla scrittura di una normale funzione.

Espressioni Generatore

- Alcuni semplici generatori possono essere codificati succintamente come espressioni usando una sintassi simile alle List Comprehension ma con parentesi tonde, invece di parentesi quadre.
- Queste espressioni sono progettate per situazioni in cui il generatore viene utilizzato immediatamente da una funzione che lo include.
- Le espressioni generatore sono più compatte ma meno versatili delle definizioni complete del generatore e tendono ad avere meno impatto sulla memoria rispetto alle equivalenti List Comprehension.

Espressioni Generatore

```
>>> sum(i*i for i in range(10))          # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec)) # dot product
260
```

La funzione zip prende due o più oggetti iterabili, aggrega gli elementi in tuple e le restituisce

Espressioni Generatore

```
>>> unique_words = set(word  for line in page  for word in
line.split())

>>> valedictorian = max((student.gpa, student.name) for
student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Funzione map

- La funzione **`map`** applica una funzione a tutti gli elementi di un oggetto iterabile

```
map(function, iterable, ...)
```

Funzione map

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
def square(x):
    return x*x

items = [1, 2, 3, 4, 5]
squared = list(map(square, items))
```

Funzione map

```
def myfunc(a):
  return len(a)
x = map(myfunc, ('apple', 'banana', 'cherry'))
print(x)
#convert the map into a list, for readability:
print(list(x))
```

```
<map object at 0x056D44F0>
['5', '6', '6']
```

Funzione map

```
def myfunc(a, b):
  return a + b
x = map(myfunc, ('apple', 'banana', 'cherry'),
('orange', 'lemon', 'pineapple'))
print(x)
#convert the map into a list, for readability:
print(list(x))


<map object at 0x034244F0>
['appleorange', 'bananalemon', 'cherrypineapple']
```

Basato su slide dell'Ing. Riccardo Zese
riccardo.zese@unife.it