Python

Iterator, Generator, Decorator

Ing. Ovidiu Daniel Barba - Università degli Studi di Roma Tor Vergata
Corso di Ingegneria degli Algoritmi (Parte Pratica)
A.A. 2018/2019

1.

Iterator

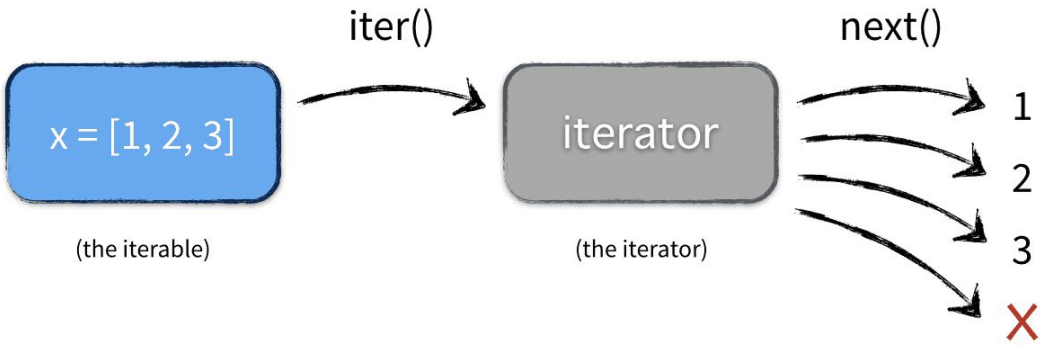


Image from https://nvie.com/posts/iterators-vs-generators/

Iterator

- Oggetti che permettono di attraversare tutti gli elementi di una collection, indipendentemente dall'implementazione
- Metodo ***next()*** che ritorna il prossimo valore della sequenza oppure solleva l'eccezione *StopIteration* se non ce ne sono altri
- Liste, tuple, dizionari, set, stringhe, ecc sono tutti container di oggetti dai quali si può ottenere un iterator
  - Hanno tutti il metodo *iter()* che ritorna un iterator
  - >>> l = [1, 2]
  - >>> iterator = iter(l)
  - >>> print(next(iterator)) # 1
  - >>> print(next(iterator)) # 2
  - >>> print(next(iterator)) # StopIteration error
- Può essere visto come una 'fabbrica' lazy di valori. E' inattiva finchè non le chiedi un valore, calcola e ritorna quel valore e si rimette nello stato di inattività

N.B.: I valori di un iterator possono essere iterati una sola volta, non si può 'tornare indietro'

Iterator - Ciclo for

- Si può usare il ciclo **for** per attraversare tutti gli elementi di un iterator
- La struttura **for … in <collection>** internamente crea un iterator chiamando il metodo *iter()* sulla collection e ritorna il prossimo elemento ad ogni iterazione usando *next()*
  - >>> tup = (‘sono’, ‘una’, ‘tupla’)
  - >>> for x in tup: # crea l’iterator i
  - >>> print(x) # x = next(i) ad ogni iterazione

Iterator Custom

- Si può creare il proprio iterator
- Basta creare una classe e implementare i metodi __iter__() e __next__():
  - in __iter__() avviene l'inizializzazione dell'iterator e il suo return (l'iterator che viene ritornato di solito è l'oggetto stesso)
  - __next__() ritorna il prossimo elemento della sequenza e solleva StopIteration quando sono terminati

Iterator Custom - Fibonacci

```
class FibIter:
    def __init__(self):
        self.prev = 0
        self.curr = 1

    def __iter__(self):
        return self

    def __next__(self):
        value = self.curr
        self.curr += self.prev
        self.prev = value
        return value
```

```
>>> f = FibIter()
>>> for i in range(10):
        print(next(f))
>>> # primi 10 numeri di Fibonacci
>>> for i in range(10):
        print(next(f))
>>> # i numeri di fibonacci da 11 a 20 (l'iterator
ricorda l'ultimo numero generato da next)
>>> for i in f:
        print(i)   # sequenza infinita di numeri di Fib
```

Modulo itertools

- Contiene funzioni che ritornano iterator particolari di sequenze finite e infinite (permutazioni, combinazioni, ecc)
- **count(n)** ritorna un iterator con tutti i numeri >= n; sequenza infinita
  - >>> count(10) # → 10 11 12 13 14 ….
- **cycle(seq_finita)** produce una sequenza infinita da un sequenza finita
  - >>> cycle('ABC') # → A B C A B C A B C ….
- **islice(seq, start, stop)** produce una sequenza finita da una infinita/finita
  - >>> from itertools import cycle, islice
  - >>> colors = cycle(['red', 'blue']) # seq infinita
  - >>> limitedColors = islice(colors, 0, 3) # seq finita
  - >>> for c in limitedColors:

        print(x) # red blue red

2.
Generator

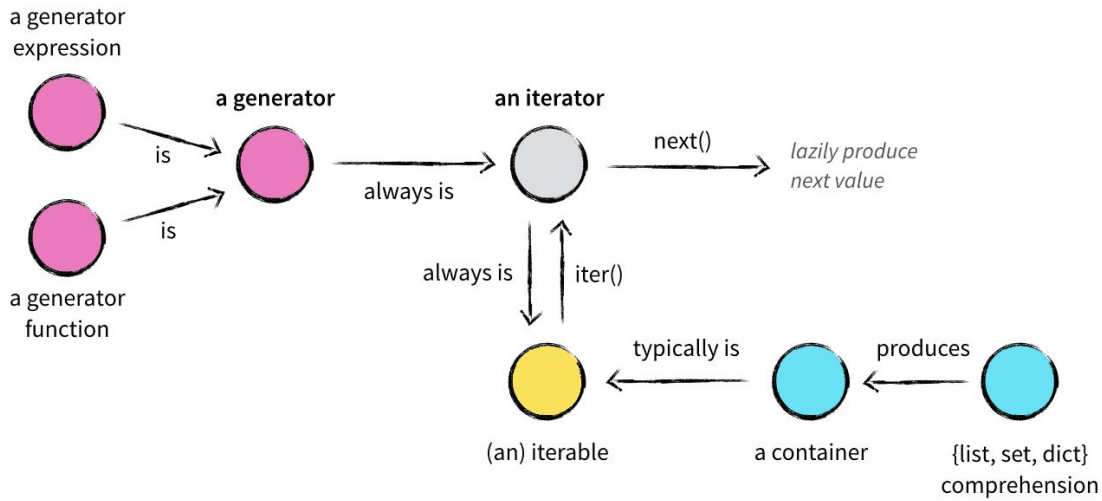


Image from https://nvie.com/posts/iterators-vs-generators/

Generatori

- Tipo speciale di Iterator
- Permette di scrivere iteratori con una sintassi chiara e semplice e senza implementare i metodi __iter__() e __next__()
- Tutti generatori sono iteratori ma non il viceversa
- Definito come una normale funzione Python con la keyword **yield** al posto della return
  - **yield** ritorna un valore al chiamante ma, a differenza di return, non distrugge le variabili locali e la prossima chiamata alla stessa funzione riprende l'esecuzione subito dopo la yield e non a inizio funzione
- Sono efficienti dal punto di vista della memoria e della CPU
  - Usateli il più possibile

range(start, stop, step) è un generator

Generator Custom - Fibonacci

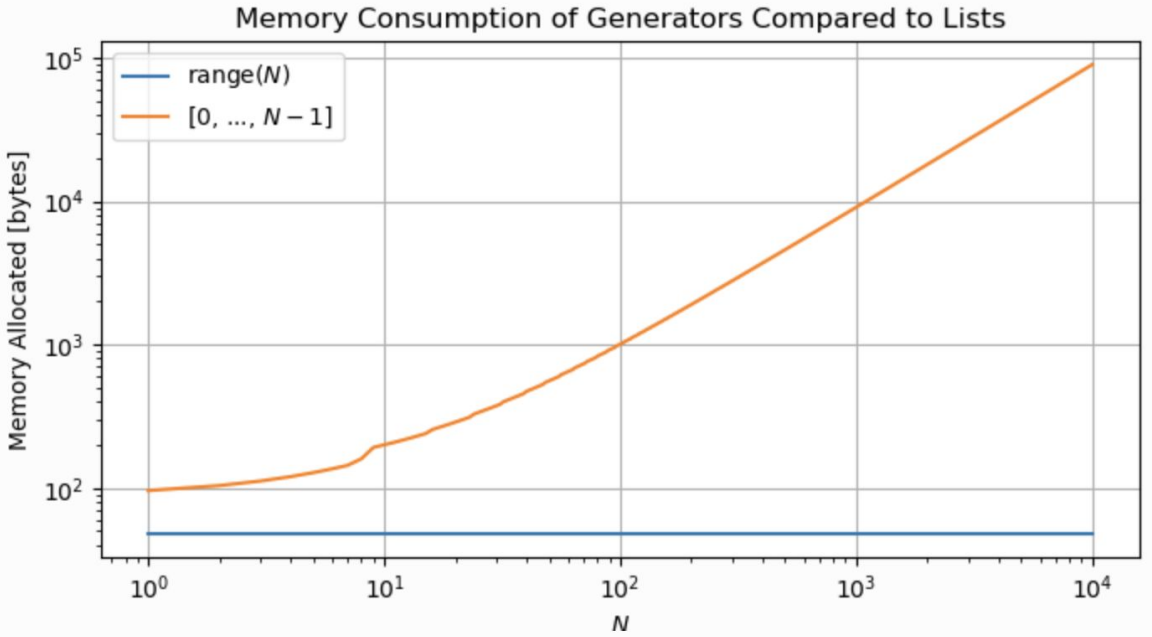
```
def fibGen():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr
```

```
>>> f = fibGen()
>>> print(next(f))  # ritorna e si ferma a yield
>>> print(next(f)) # riprende subito dopo yield, fa un'altra iterazione del ciclo while, ritorna a yield e si ferma di nuovo
……..
```

```
>>> f = fibGen()
>>> firstFibNumbers = list(islice(f, 0, 10)) # seq finita
>>> print(firstFibNumbers)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> for n in f: #  sequenza infinita
….      print(n)
```

Generator vs List: Memoria allocata

- I generatori non allocano memoria per tutti gli elementi che può ritornare ma vengono calcolati appena richiesti
- La memoria allocata dai generatori è indipendente dal numero di elementi da generati



3.

@decorator

Image from http://www.xappsoftware.com/wordpress/2017/12/11/python-decorators-for-dummies-me/

Decorator

- Funzione che prende un'altra funzione *f* e ne estende il comportamento senza modificarla esplicitamente. Inoltre ritorna la funzione che decora *f*
- Reminder : in Python le funzioni possono essere passate come argomento di altre funzioni, possiamo definire funzioni nel corpo di altre funzioni e fare *return* di funzioni
- Se la funzione da decorare ha qualche argomento, devono essere passati alla funzione ritornata dal decorator, altrimenti ci viene ritornato un errore
- Se invece la funzione da decorare non ha argomenti, possono essere omessi nella funzione interna al decorator
  - Gli argomenti della funzione da decorare possono essere passati come **(*args, **kwargs)** indipendentemente dal numero di argomenti

Decorator Custom

```
# definire il decorator (semplice funzione che prende come parametro una funzione e ne ritorna un'altra) con nessun argomento della funzione da decorare

def printDecorator(func):
    def wrapFunction():
        print('Before calling func')
        func()
        print('After calling func')
    return wrapFunction
```

```
# come usare il decorator
@printDecorator
def funcToCall():
    print('Function called')

>>> funcToCall()   # output :
Before calling func
Function called
After calling func
```

Decorator Custom - Cache Chiamate Ricorsive

```
# definire il decorator con un numero arbitrario di  argomenti della funzione da decorare

def functionCallCache(func):
    cache = {}   # mantiene i risultati delle chiamate di func
    def wrapFunction(*args, **kwargs):  # argomenti di func
        if args not in cache:
            # calcola il risultato della funzione da decorare e lo salva in cache
            cache[args] = func(*args, **kwargs)
        return cache[args]
    return wrapFunction
```

Decorator Custom - Cache Chiamate Ricorsive

```
# decoriamo Fibonacci ricorsivo

@functionCallCache
def fibonacci(n):
    print(f'Calling fibonacci({n})')
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci(1000)   #   quanto ci mette?
```

Decorator Custom - Timer

```
# Decorator per calcolare il tempo di esecuzione di una funzione
from time import time
def timer(func):
    def wrapping_function(*args, **kwargs):
        start = time()
        value = func(*args, **kwargs)    # chiamiamo da funzione da analizzare
        elapsed = time() - start
        print(f'Function {func.__name__} took {elapsed} seconds')
        return value    # dobbiamo ritornare il valore calcolato dalla funzione
    return wrapping_function

@timer    # applichiamo il decorator
def passa_il_tempo():
    # fai qualcosa
>>> passa_il_tempo()
Function passa_il_tempo took 0.024322 seconds
```

Decorator Custom con Argomenti - Ripetizione funzioni

```
# Decorator che ripete n volte una funzione
def repeat(num_times=2):
    def repeat_decorator(func):    # vero e proprio decorator
        def wrap_func(*args,**kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value    # valore funzione da decorare
        return wrap_func    #  funzione di wrap di func
    return repeat_decorator   #  ritorna il decorator

@repeat(num_times=3)   # applichiamo la funzione che ritorna il decorator
def printer():
    print('Print once')
>>> printer()
Print once
Print once
Print once
```

Class Decorators

- **@staticmethod**
  - Rende un metodo di una classe statico: può essere chiamato come <NomeClasse>.<metodo()> senza istanziare un oggetto
- **@property**
  - Controlla l'assegnamento e il ritorno di un attributo della classe
  - Idea: l'attributo è 'privato' alla classe, la quale può permettere o meno il suo aggiornamento e ritorno all' 'esterno'
- **@abstractmethod**
  - Presente nel modulo *abc*
  - Individua un metodo senza implementazione (non fa nulla)
  - Forza i metodi di una classe astratta (prossima slide) ad essere implementati dalle sue sottoclassi

Classi Astratte

- Classi che definiscono un comportamento ma non l'implementazione
- Hanno metodi astratti
- Presenti nel modulo *abc* (AbstractBaseClass)
- Una classe, per essere astratta deve estendere ABC (*Python 3.4+*) e segnare i metodi con il decorator *@abstractmethod*
- Le sottoclassi devono implementare tutti i metodi astratti, altrimenti non sono istanziabili

Classi Astratte Custom

```
from abc import ABC, abstractmethod
class  Animal(ABC):
        @abstractmethod
        def name(self):
                Pass
>>> a  = Animal() # errore, metodo non implementato
class Cat(Animal):
        pass
>>> c= Cat()  # errore, metodo non implementato
class Dog(Animal):
        def name(self):
                return 'Dog'
>>> d = Dog()  # ok
>>> print(d.name())   # Dog
```

Metodi statici

```
class TypeHelper:
    def __init__(self, max):
        self.max = max
    @staticmethod
    def isInteger(self, n):
        return isInstance(n, int)
    def isGreaterThanMax(self, n):
        return n > self.max
```

```
>>> TypeHelper.isInteger(12)  # True. Non c'è bisogno di istanziare TypeHelper
>>> TypeHelper.isGreaterThanMax(120) # ERROR, metodo NON statico
>>>  th = TypeHelper(100)   # istanziamo per poter usare metodi normali (non statici)
>>> th.isGreaterThanMax(120)  # True;  OK
```