

# Generatori e coroutine

Programmazione Avanzata

Annalisa De Bonis

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

1

## L'espressione yield

- L'espressione `yield` e' usata per definire un generatore e quindi puo` essere solo usata nel corpo della definizione di una funzione.
- L'uso di `yield` nel corpo di una funzione trasforma la funzione in generatore
- Quando viene invocata una funzione generatore viene restituito un iteratore chiamato generatore
- L'esecuzione del generatore comincia quando viene invocato uno dei metodi del generatore. L'esecuzione continua fino alla prima espressione di `yield` dove l'esecuzione e' sospesa e viene restituito il valore prodotto da `yield` al codice che ha invocato il metodo.
  - Viene restituito il valore dell'`expression_list` nell'espressione `yield` (`expression_list`: singola espressione o tupla di espressioni) Ad esempio (`yield 2`) restituisce 2, (`yield 3, 4`) restituisce (3, 4). Il valore dell'espressione `yield` e' un'altra cosa (vedi sotto)
- Quando l'esecuzione e' ripresa invocando uno dei metodi del generatore, l'espressione `yield` assume un valore che dipende dal metodo che ha fatto riprendere l'esecuzione. Se viene usato `__next__()` allora il valore dell'espressione `yield` e' `None` altrimenti se viene usata una `send()`, il valore dell'espressione e' il valore passato a `send()`.

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

2

## I metodi del generatore

- `generator.__next__()` : comincia l'esecuzione di un generatore o la riprende dall'ultima espressione `yield` eseguita
- Quando l'esecuzione di un generatore riprende per mezzo dell'invocazione di `__next__()` , l'espressione `yield` corrente assume sempre valore `None`. L'esecuzione poi continua fino alla prossima espressione `yield` dove l'esecuzione viene nuovamente sospesa e il valore dell'`expression_list` è restituita al programma che ha invocato `__next__()`
  - se invece l'esecuzione del generatore termina senza produrre un altro valore allora viene lanciata `StopIteration`.

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

3

## I metodi del generatore

- Il metodo `send()` riprende l'esecuzione del generatore e invia un valore al generatore. Il valore passato come argomento a `send` diventa il risultato dell'espressione `yield` da cui riprende l'esecuzione.
- `send()` restituisce il prossimo valore fornito dall'espressione `yield` raggiunta dopo che è ripresa l'esecuzione o lancia `StopIteration` se si esce dal generatore senza ottenere un altro valore.
- Quando `send()` è invocata per avviare il generatore essa deve essere invocata con `None` come argomento. perché non è stata ancora raggiunta un'espressione `yield` in grado di ricevere il valore.

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

4

```

def raddoppio():
    while True:
        x=yield #tipicamente raggiunto mediante un next.
                #Prima della sospensione non fornisce niente al programma che invoca next
                #perche' la expression_list e` vuota
                # Se l'esecuzione e` poi ripresa per via di una send
                #allora l'espressione yield assume valore
                #uguale all'argomento di send che viene assegnato a x

        print("stampa tra un yield e l'altro del while. x=" ,x)

        x=yield x*2 #!eventuale send (di cui al commento precedente)
                   # fa in modo che venga raggiunto questo yield.
                   #La send restituisce x*2 e l'esecuzione
                   #si sospende a questo yield.
                   #Se il programma invoca poi next()

                   #!esecuzione di next riprende l'esecuzione del while e va alla prossima iterazione
                   #qui l'assegnamento e` inutile ma è stato messo per far vedere che x e` None dopo l'assegnamento

        print('stampa fine corpo while x=',x)

```

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

5

```

g=raddoppio()
r=next(g) raggiunta linea 1, in r viene messo None restituito da yield
if r==None:
    print("next(g) non ha restituito niente")
r=g.send(5) ripresa esecuzione da 3, in x viene messo 5, in r viene messo 10
print ("send ha restituito ",r)
r=next(g) ripresa esecuzione da 5, in x viene messo None, in r None
if r==None:
    print("next(g) non ha restituito niente")
r=g.send(8) ripresa esecuzione da 3, in x viene messo 8, in r viene messo 16
print ("send ha restituito ",r)

```

1. def raddoppio():
2. while True:
3. x=yield
4. print("stampa tra "\
- "un yield e l'altro"\
- "del corpo del while: x ha valore" ,x)
5. x=yield x\*2
6. print("stampa fine corpo while. x=",x)

```

next(g) non ha restituito niente
stampa tra un yield e l'altro del corpo del while. x= 5
send ha restituito 10
stampa fine corpo while. x= None
next(g) non ha restituito niente
stampa tra un yield e l'altro del corpo del while. x= 8
send ha restituito 16

```

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

6

```
print("un' altra versione di raddoppio")
def raddoppio():
    x=2 #occorre inizializzare x perche' altrimenti errore alla prima esecuzione dell'istruzione contenente yield

    while True:
        x=(yield x) *2
        print("stampa tra un yield e l'altro del corpo del while: x ha valore",x)
```

```
g=raddoppio()
r=next(g)
print("next(g) ha restituito il {} prodotto dalla prima esecuzione di yield".format( r ))
r=g.send(5)
print ("send ha restituito ",r)
r=g.send(8)
print ("send ha restituito ",r)
```

```
un' altra versione di raddoppio
next(g) ha restituito il 2 prodotto dalla prima esecuzione di yield
stampa tra un yield e l'altro del corpo del while: x ha valore 10
send ha restituito 10
stampa tra un yield e l'altro del corpo del while: x ha valore 16
send ha restituito 16
```

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

7

## I metodi del generatore

- `throw()` lancia un'eccezione del tipo che gli è stato passato come argomento nel punto in cui l'esecuzione del generatore è stata sospesa.
- Il metodo `throw()` restituisce il prossimo valore prodotto dal generatore se questo ne produce uno nuovo
- Se il generatore non cattura l'eccezione "inviata" da `throw` o lancia un'eccezione differente allora l'eccezione immessa da `throw` si propaga al codice che ha invocato `throw`.

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

8

## I metodi del generatore

- `close()` lancia una `GeneratorExit` nel punto dove la funzione generatore è sospesa.
- Se accade che il generatore termini subito dopo senza lanciare eccezioni, o se è già chiuso, o ancora se lancia `GeneratorExit` (in quanto non la cattura), `close` restituisce il controllo al codice che l'ha invocato.
- Se il generatore produce un valore, viene lanciata una `RuntimeError`. Se lancia una qualsiasi altra eccezione, questa viene propagata al codice che ha invocato `close`.
- `close()` non fa niente se il generatore ha già terminato la sua esecuzione normalmente o a causa di un'eccezione.

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

9

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is called.")
...
>>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

**esempio tratto dal manuale**

**next: yield nella prima iterazione restituisce valore di value passato in input**

**next: riprende esecuzione della prima iterazione del while a partire da yield; viene posto value a None; viene raggiunto yield nella seconda iterazione del while e viene restituito value (cioè None)**

**send: riprende esecuzione della seconda iterazione del while a partire da yield; viene posto value a 2; viene raggiunto yield nella terza iterazione e viene restituito value (cioè 2)**

**throw: lancia eccezione mentre generatore sospeso a yield nella terza iterazione; eccezione catturata e messa in value; viene raggiunto yield nella quarta iterazione e viene restituito value (cioè l'eccezione);**

**close: lancia GeneratorExit mentre generator è sospeso ad yield nella quarta iterazione; l'eccezione non è catturata da except perché GeneratorExit non è sottoclasse di Exception. GeneratorExit non viene rilanciata dopo il finally perché il controllo viene restituito al caller (si veda dslice precedente).**

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

10

## Yield from

- `yield from` permette ad un generatore di delegare parti delle sue operazioni ad un altro generatore.
- `yield from <expr>` tratta l'espressione fornita come un subiteratore. Tutti i valori prodotti dal subiteratore sono passati direttamente al codice che ha invocato i metodi del generatore corrente.
- I valori passati da `send()` e qualsiasi eccezione passata da `throw()` sono passati all'iteratore sottostante se ha i metodi appropriati. Se non è questo il caso, `send()` lancia `AttributeError` o `TypeError` mentre `throw()` lancia l'eccezione passata come argomento a `throw`.

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

11

## yield from

da <https://docs.python.org/3/whatsnew/3.3.html>

- Per iteratori semplici, **`yield from iterable`** è essenzialmente equivalente a **`for item in iterable: yield item`**

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

12

## yield from

da <https://docs.python.org/3/whatsnew/3.3.html>

```
def accumulate():
    """se viene inviato un valore con send(), somma il valore a tally; si interrompe se non viene inviato niente"""
    tally = 0
    while 1:
        next = yield
        if next is None:
            return tally
        tally += next

def gather_tallies(tallies):
    """delega il lavoro ad accumulate per ottenere una somma che poi appende alla lista tallies"""
    while 1:
        tally = yield from accumulate()
        tallies.append(tally)
    tallies = []
    acc = gather_tallies(tallies)
    next(acc) #Assicura che acc sia pronto a ricevere valori (esecuzione si sospende allo yield)
    for i in range(4):
        acc.send(i) #i valori vengono inviati ad acc e quindi ad accumulate
    acc.send(None) # Fa terminare l'esecuzione di accumulate richiesta da gather tallies print(tallies)
    print(tallies)
```

[6]

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

13

## Yield from

- Scrivere una funzione generatrice `myGenerator(n)` che prende in input un intero  $n \geq 1$  e restituisce un iteratore dei primi  $n$  fattoriali. In altre parole, la prima volta che viene invocato `next` viene restituito  $1!$ , la seconda volta  $2!$ , la terza volta  $3!$ , e così via fino ad  $n!$ .
- **Bonus se la funzione generatrice è definita ricorsivamente. In questo caso è consentito scrivere una funzione generatrice ricorsiva che prende in input più parametri e che poi viene opportunamente invocata da `myGenerator`.**
  - Soluzione nella prossima slide

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

14

## yield from

generatore dei primi n fattoriali

```
def myGenerator(n):
    return myGeneratorAux(n,1,1)

def myGeneratorAux(n,c,p):
    if n==1: yield p
    else:
        yield p
        c=c+1
        p=c*p
        yield from myGeneratorAux(n-1,c,p)

if __name__=="__main__":
    print("I primi 6 fattoriali sono:")
    for x in myGenerator(6):
        print(x)
```

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

15

## yield from

visita inorder di alberi binari

```
def inorderV(tree):
    if tree is not None:
        try:
            value, left, right = tree
        except ValueError: # wrong number to unpack
            print("Bad tree:", tree)
        else: # The following is one of 3 possible orders.
            yield from inorderV(left)
            yield value # Put this first or last for different orders.
            yield from inorderV(right)

tree = ('a', ('b', ('c',None,None), ('d', ('e', None, None), ('f', ('g', None, None), None))),
        ('h', None, ('i', ('l', ('m', None, None), None), None)))

print([x for x in inorderV(tree)])
```

```
['c', 'b', 'e', 'd', 'g', 'f', 'a', 'h', 'm', 'l', 'i']
```

Programmazione Avanzata a.a. 2021-22  
A. De Bonis

16