

Programmazione Avanzata e parallela

Lezione 23

Luca Manzoni

Python e funzioni

Generatori, decoratori

- Generatori
- Iteratori
- Funzioni e closures
- Decoratori

Generatori

Interrompere e riprendere l'esecuzione

- Quando iteriamo a volte non è necessario avere tutta la lista/array su cui iteriamo (soprattutto se è potenzialmente infinito)
- È sufficiente poter generare gli oggetti uno alla volta
- Serve un costrutto in grado di darci:
 - Il modo di ritornare un valore
 - Il modo di proseguire l'esecuzione dopo che aveva ritornato quel valore

Generatori

Interrompere e riprendere l'esecuzione

- Possiamo utilizzare i **generatori** un caso particolare di **coroutine**
- Una **coroutine** ha due principali caratteristiche:
 - Lo stato delle variabili locali viene preservato tra chiamate successive
 - La coroutine è sospesa quando il controllo esce al di fuori della coroutine
 - Quando la coroutine riprende il controllo l'esecuzione riprende da dove si era interrotta

Coroutine

Ideazione

Le coroutine furono scoperte alla fine degli anni '50 e formalmente definite nel 1963:

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.¹ There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

Conway, Melvin E., *Design of a Separable Transition-diagram Compiler*
Communications of the ACM. ACM. 6 (7): 396–408, 1963

Generatori

Interrompere e riprendere l'esecuzione

- Nel caso particolare dei generatori in Python usiamo la keyword “**yield**” per:
 - Rendere disponibile (o richiedere un valore) all'esterno di una funzione
 - L'esecuzione si interrompe quando chiamiamo `yield...`
 - ...e riprenderà da quel punto quanto chiederemo di proseguire

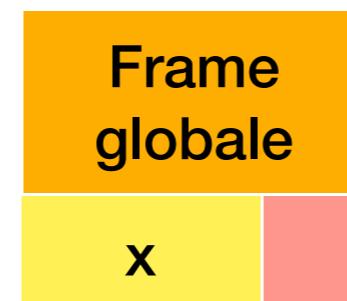
Generatori

Interrompere e riprendere l'esecuzione

- Usare un generatore:
 - Costruiamo un generatore (e.g., chiamiamo una funzione che usa `yield`), e.g., `x = f()`
 - Per ottenere il primo valore con `yield` chiamiamo la funzione `next` con argomento il generatore, e.g., `next(x)`
 - La prossima volta che chiamiamo `next` l'esecuzione di `f` riprenderà dello `yield` in cui si era fermata...
 - ...per proseguire fino allo `yield` successivo (o, se non c'è, produrre una eccezione di tipo `StopIteration`)

Generatori

Interrompere e riprendere l'esecuzione

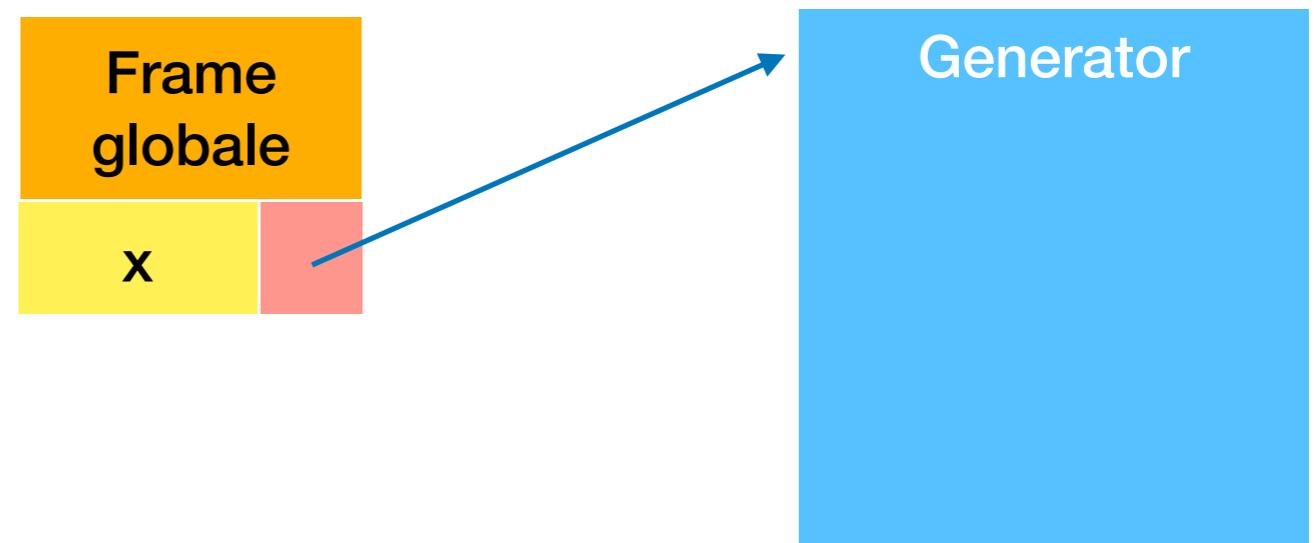


```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

→ `x = f()
print(next(x))
print(next(x))`

Generatori

Interrompere e riprendere l'esecuzione



```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

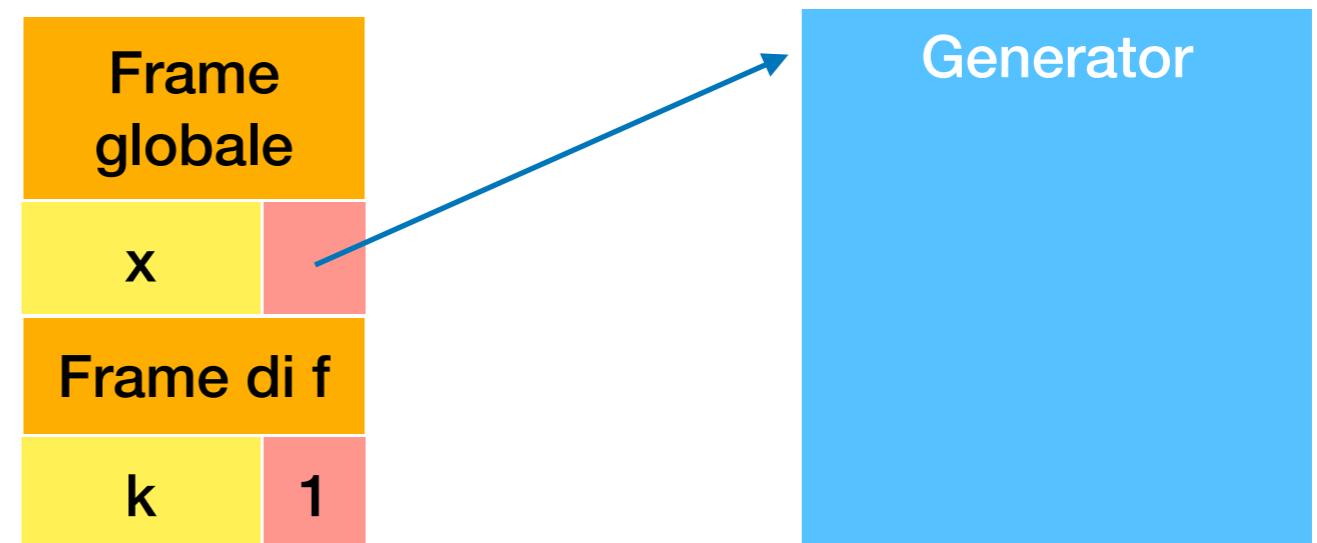
→ `x = f()
print(next(x))
print(next(x))`

Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

```
x = f()
→ print(next(x))
print(next(x))
```

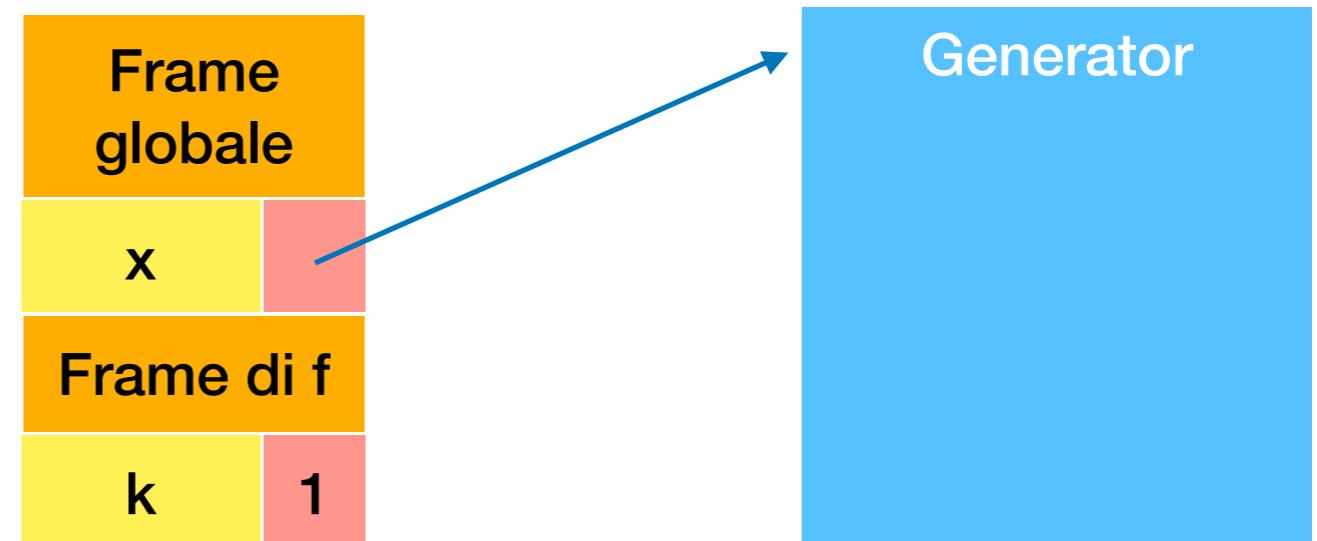


Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

```
x = f()
print(next(x))
print(next(x))
```



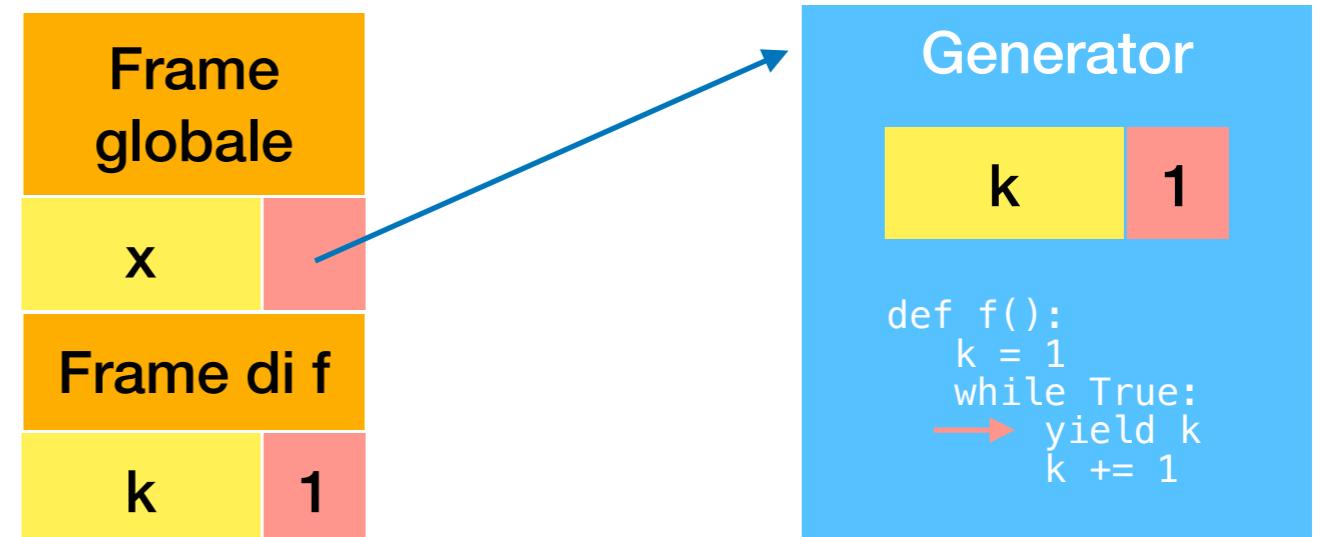
Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```



```
x = f()
→ print(next(x))
print(next(x))
```

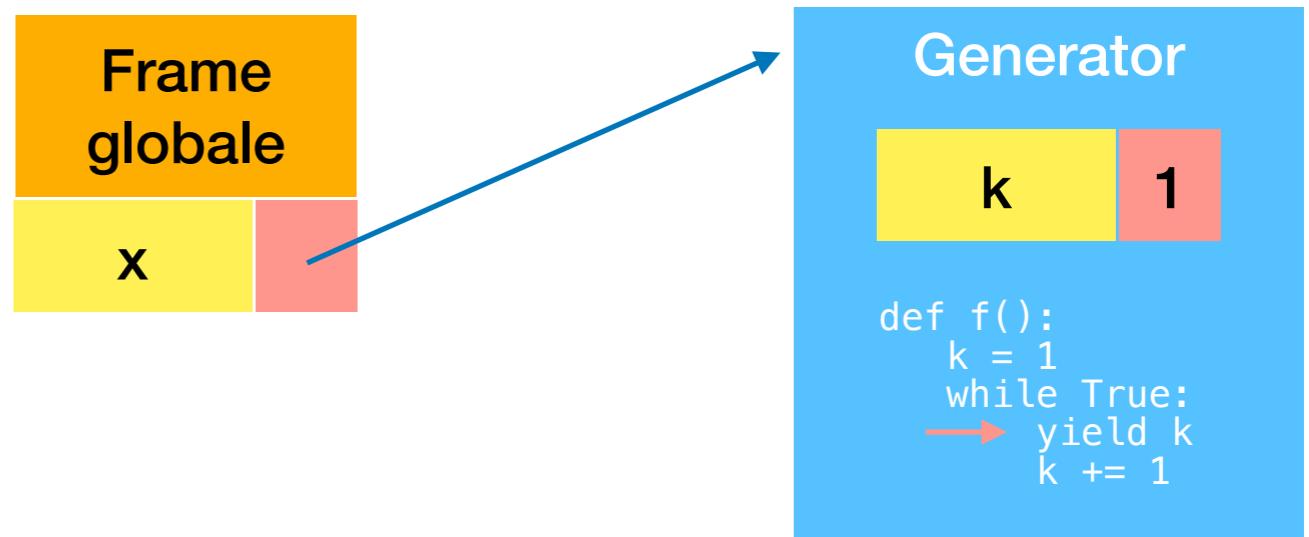


Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

```
x = f()
→ print(next(x))
print(next(x))
```

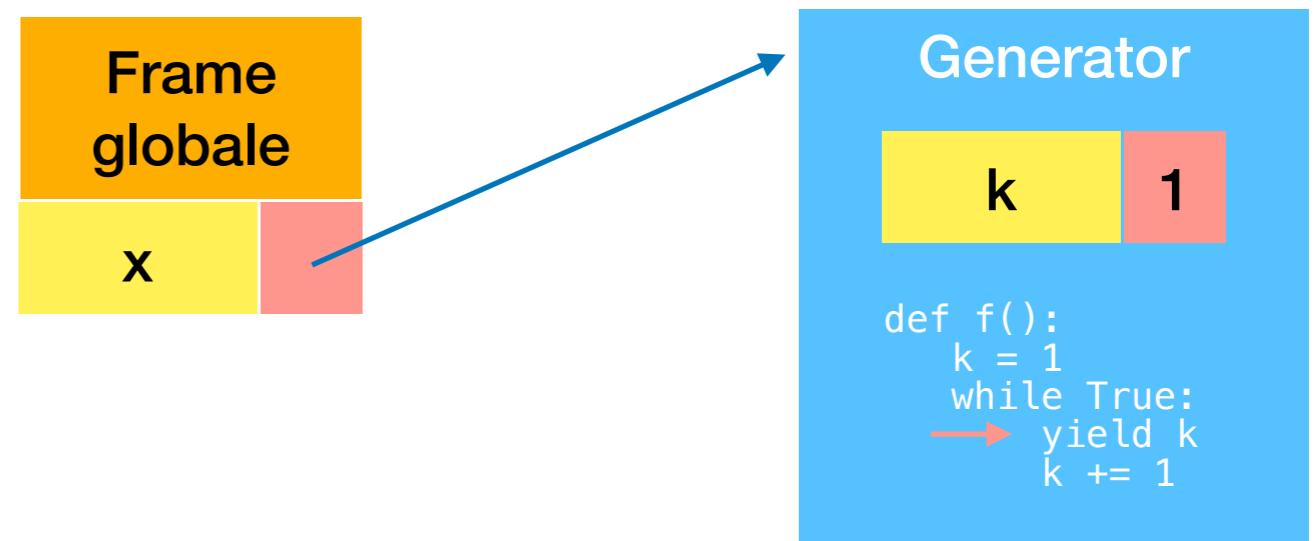


Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

```
x = f()
print(next(x))
print(next(x))
```



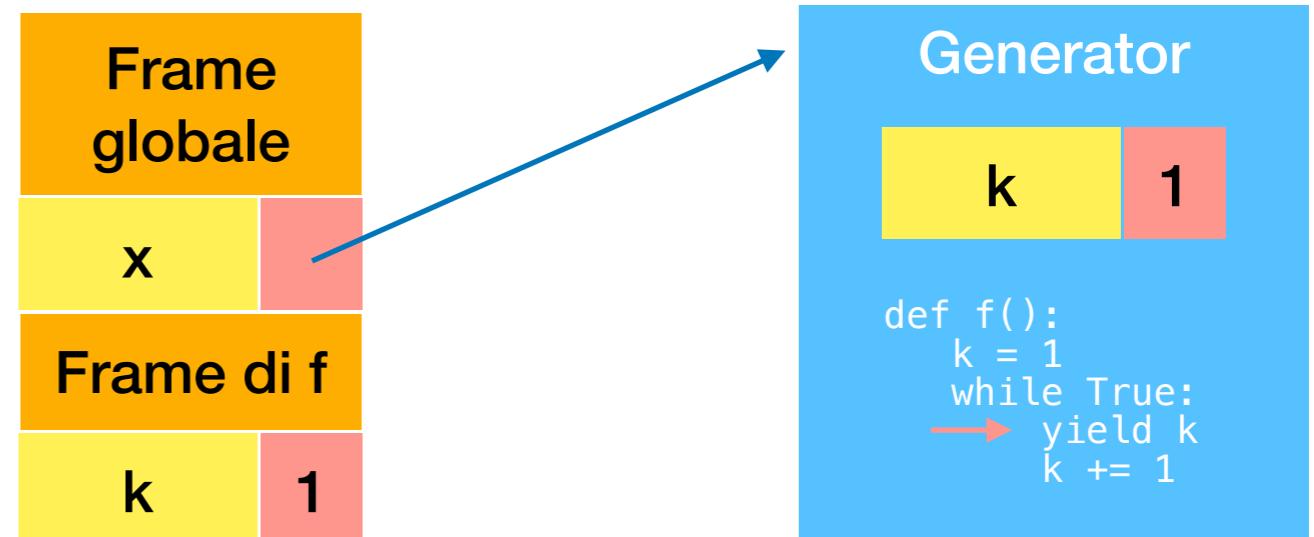
Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```



```
x = f()
print(next(x))
→ print(next(x))
```



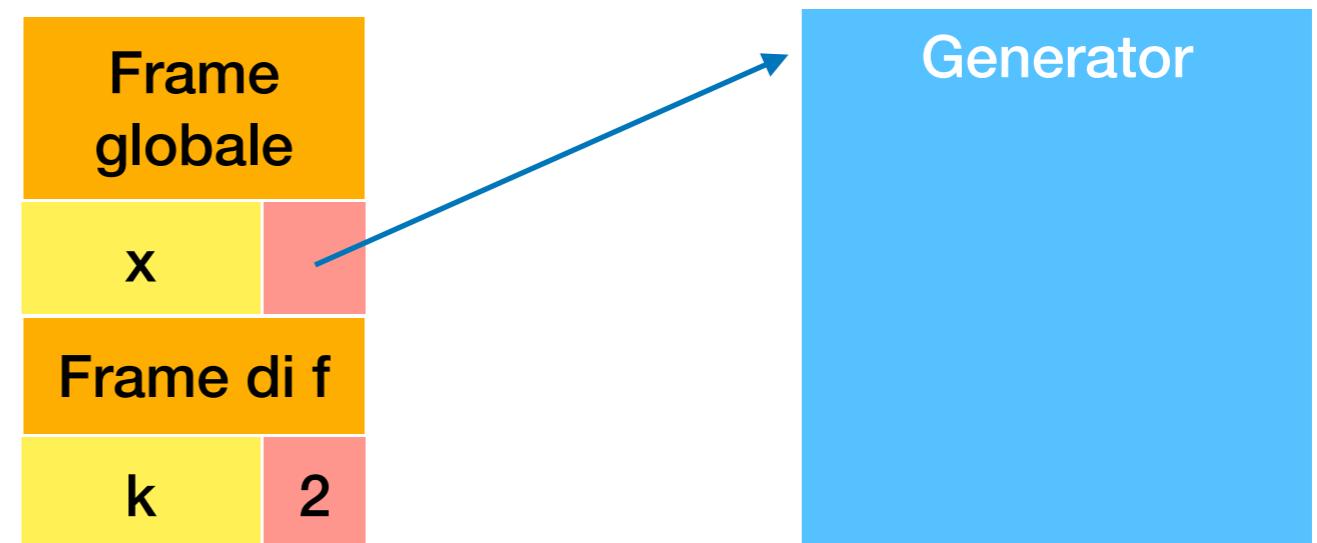
Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```



```
x = f()
print(next(x))
→ print(next(x))
```



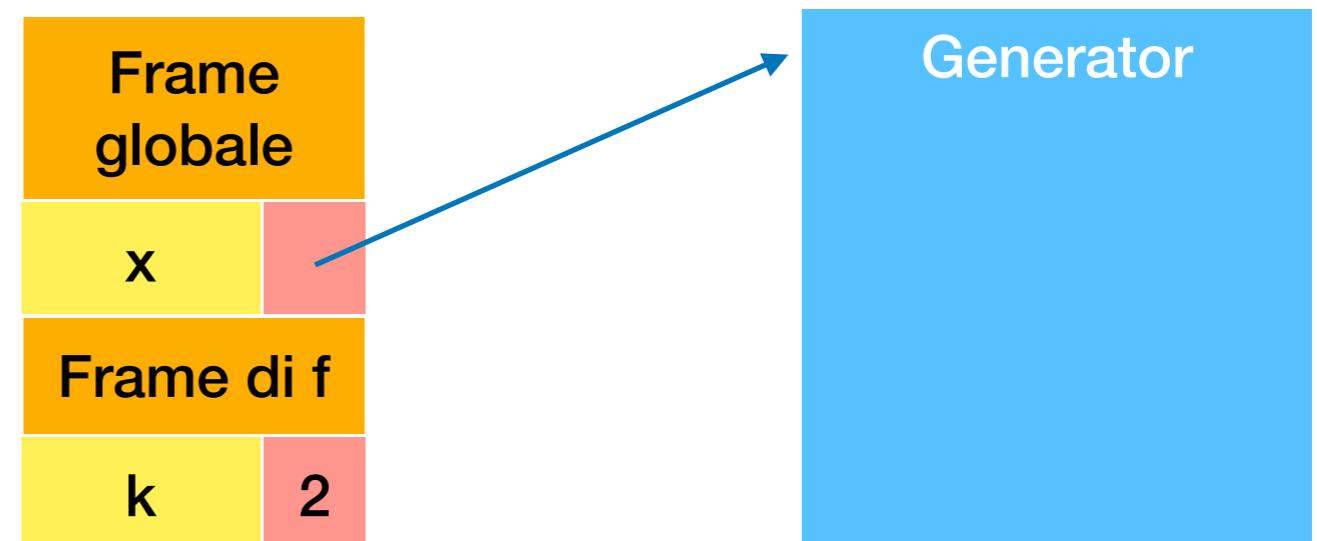
Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```



```
x = f()
print(next(x))
print(next(x))
```



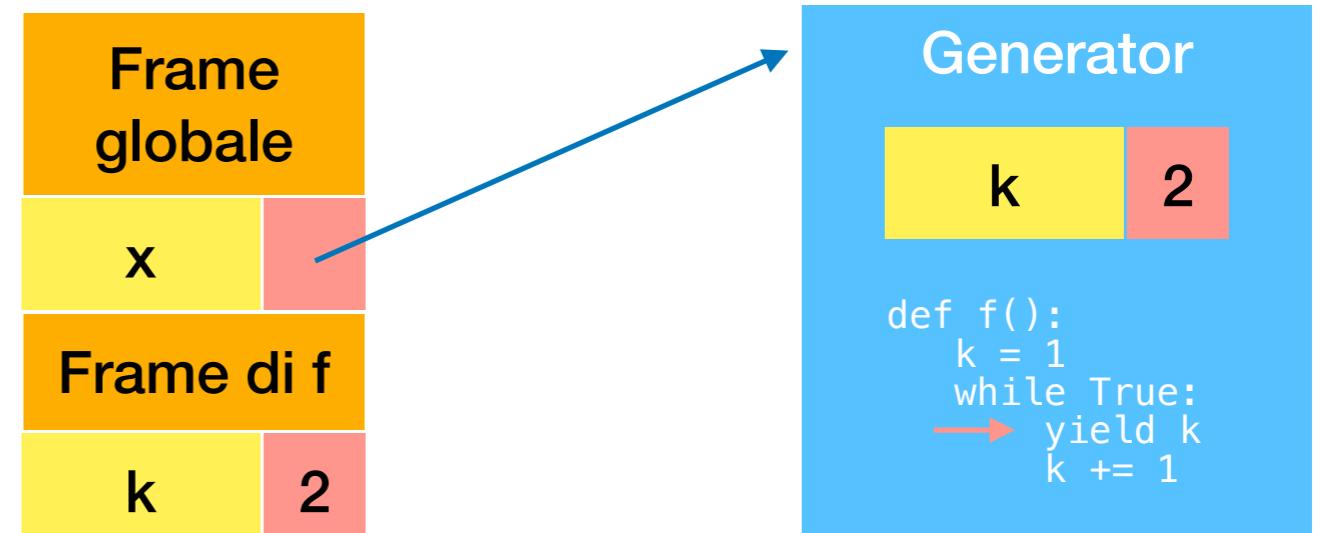
Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```



```
x = f()
print(next(x))
→ print(next(x))
```

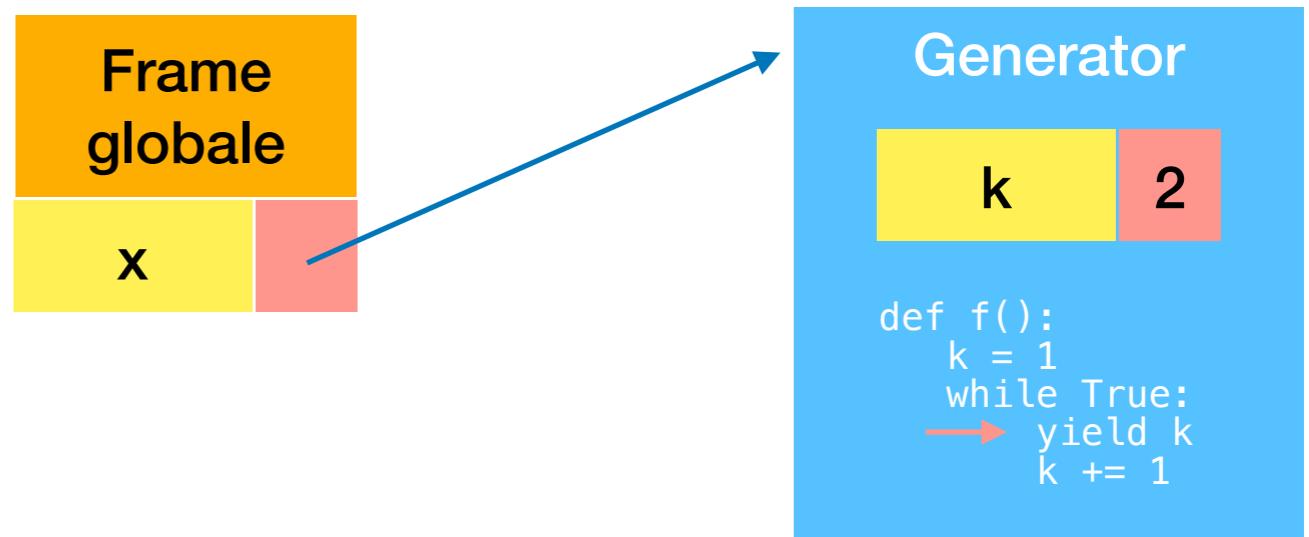


Generatori

Interrompere e riprendere l'esecuzione

```
def f():
    k = 1
    while True:
        yield k
        k += 1
```

```
x = f()
print(next(x))
print(next(x))
```



Generatori

Interrompere e riprendere l'esecuzione

- Quando iteriamo a volte non è necessario avere tutta la lista/array su cui iteriamo (soprattutto se è potenzialmente infinito)
- È sufficiente poter generare gli oggetti uno alla volta
- Serve un costrutto in grado di darci:
 - Il modo di ritornare un valore
 - Il modo di proseguire l'esecuzione dopo che aveva ritornato quel valore

Generatori

Comprehension

- Così come abbiamo list comprehension possiamo effettuare comprehension coi generatori:
- Per liste:
`[x for x in range(10)]`
- Per generatori:
`(x for x in range(10))`
- Mentre la lista deve essere generata tutta (e quindi stare in memoria), il generatore può costruire gli elementi uno alla volta

Iteratori

Generatori e oggetti

- Possiamo iterare sugli argomenti di una collezione usando anche oggetti che rispettano un “iterator protocol”, ovvero si comportano come iteratori
- Ci basta definire due metodi:
 - **`__iter__(self)`** svolge una inizializzazione (se necessaria) e ritorna l'iteratore
 - **`__next__(self)`** ritorna il valore successivo nella sequenza (o genera una eccezione `StopIteration` se non c'è un valore successivo)

Iteratori

Generatori e oggetti

- Un oggetto che implementa quei due metodi può essere usato come un generatore:
 - Nei cicli for
 - Ottenendo l'iteratore con la funzione “**iter**” e chiamando “**next**” su di esso
 - Notate come lo stato interno dell'oggetto “imita” quello che avviene nelle funzioni con “**yield**”

Iteratori

Itertools

- Per lavorare su iteratori esiste la libreria `itertools`:
 - **chain(a, b, ...)**
itera su tutti gli elementi di *a*, poi su tutti quelli di *b*, etc.
 - **product(a, b, ...)**
itera su tutte le combinazioni di valori di *a*, *b*, etc.
È come avere un ciclo innestato
 - **pairwise(a)**
itera su coppie della forma $(a[0], a[1])$, $(a[1], a[2])$, etc.

Iteratori

Itertools

- **permutations(a)**
itera su tutte le permutazioni degli elementi di *a* (attenzione: sono $n!$)
- **combinations(a, m)**
itera su tutte le tuple ordinate di *m* elementi di *a* senza ripetizioni (equivalente a iterare sui tutti i possibili risultati di estrarre *m* elementi da *a* senza reinserimento)
- **repeat(x, m)**
ripete il valore *x* per *m* volte. Se *m* non è specificato l'iteratore è infinito

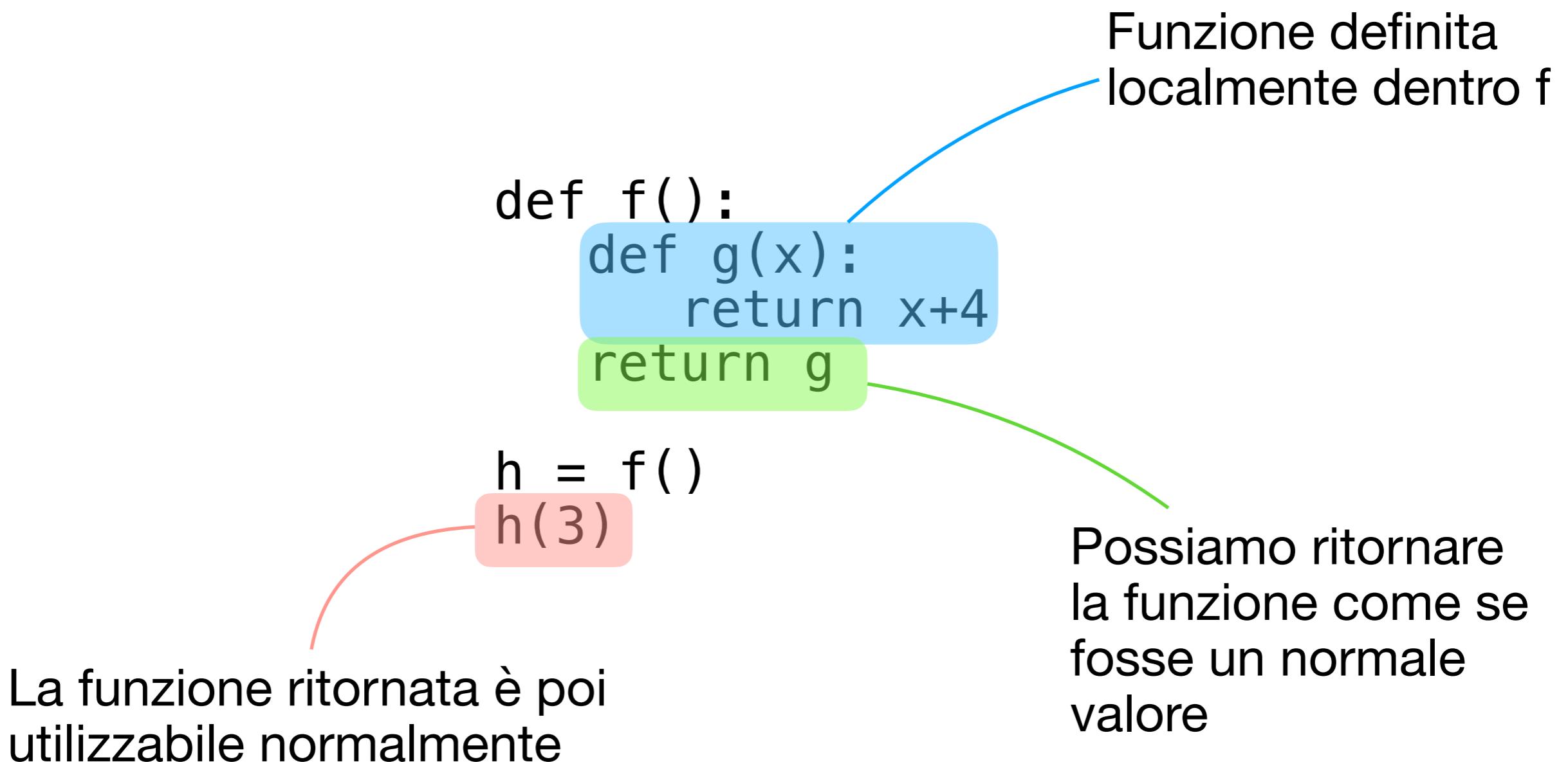
Funzioni che lavorano su funzioni

Funzioni come first-class objects

- È possibile definire funzioni che:
 - Ricevono funzioni come argomenti
 - Ritornano funzioni
- Poter direttamente manipolare funzioni come se fossero valori significa che le funzioni sono “first-class objects”

Funzioni innestate

Definire funzioni dentro funzioni



Closures

Catturare l'ambiente

- E se dentro una funzione “interna” usassimo una variabile locale della funzione esterna?
- In questo caso la funzione ha delle variabili libere, ovvero che non sono definite in essa ma nell’ambiente esterno
- Si ottiene una *closure*, che è composta da due parti
 - Una funzione f
 - Un ambiente che a ognuna della variabili libere di f associa il valore che aveva al momento della definizione di f

Closures

Catturare l'ambiente

La funzione g “cattura” il valore di x al momento della sua definizione

```
def f(x):  
    def g(y):  
        return x+y  
    return g
```

h = f(5)
h(3)

La funzione ritornata è poi utilizzabile normalmente

Chiamando f con argomento “5” abbiamo che la funzione ritornata avrà “catturato” l’argomento

Funzioni anonime

Funzioni senza nome

- Possiamo creare funzioni senza dar loro un nome usando la keyword `lambda`
- Il nome stesso delle funzioni anonime è “lambda functions” (riferimento al lambda calcolo di Alonso Church)
- Nel caso di Python sono molto più limitate che in altri linguaggi:
 - Limitate ad una singola espressione...
 - ...quindi niente flusso di controllo complesso o espressioni multiple

Funzioni anonime

Come definirle

Qui indichiamo che questa è una funzione anonima

```
lambda x: x + 3
```

Corpo della funzione, è implicito un return

Elenco degli argomenti

Lavorare con funzioni

Map, filter, reduce

- Molte operazioni su collezioni sono riconducibili a particolari applicazioni di funzioni su di esse:
 - **map(f, collection)**
applica la funzione f a tutti gli elementi della collezione
 - **filter(predicate, collection)**
applica il predicato (funzione che ritorna valori Booleani) agli elementi della collezione e mantiene solo quelli per cui il predicato è vero

Lavorare con funzioni

Map, filter, reduce

- **reduce(f, collection)**
usa la funzione f (che ha due argomenti) per ridurre la collezione ad un solo valore:
 - Se f è *lambda x, y: x + y* e la collezione [1,2,3,4] allora:
 $f(f(f(1,2), 3), 4)$ corrisponde a $((1 + 2) + 3) + 4$
 - Per usarlo in Python serve importarlo da *functools*
- Notate come questi risultati siano anche ottenibili usando list/dict comprehension

Lavorare con funzioni

Applicazione parziale

- A volte può essere utile applicare una funzione solo parzialmente:
 - Abbiamo $f(x, y)$ e un valore x
 - Vogliamo definire $f_x(y)$ come $f(x, y)$
- Questa operazione, detta di currying, è il trasformare un funzione che prendere argomenti multipli in una sequenza di funzioni che prendono un argomento alla volta
- Possiamo usare “*partial(f, args)*” importandolo da *functools*

Decoratori

Modificare funzioni e metodi

- Idea di base:
 - Stiamo definendo una funzione **f**
 - Abbiamo una funzione **g** che prende come argomento una funzione e ritorna una funzione
 - Vogliamo sostituire **f** con **g(f)**
- Esempi di applicazione: prendere dati dell'esecuzione, aggiungere una cache, etc.

Decoratori

Brief recap

- Per poter usare in modo efficace i decoratori ci serve ricordare alcune notazioni di Python:
 - ***args** negli argomenti delle funzione andrà a raccogliere una lista con tutti gli argomenti passati (esclusi quelli che precedono ***args**, che hanno già un nome)
 - ***args** come argomento “appiattisce” una lista di argomenti passando gli elementi come singoli argomenti:
`args = [1,2,4]`
`f(*args) # equivale a f(1,2,4)`
 - Per i keyword args abbiamo invece ****kwargs**

Decoratori

Semplice definizione

All'interno del decoratore definiamo la nuova funzione. Se non sappiamo quanti argomenti ha f possiamo usare *args e **kwargs

```
def decorator(f):
    def wrapped(*args, **kwargs):
        return f(*args, **kwargs)
    return wrapped
```

Il decoratore ritorna una nuova funzione che racchiude f (per aggiungere funzionalità)

Decoratori

Modificare funzioni e metodi

- Possiamo poi utilizzare un decoratore prima della definizione della funzione con la notazione `@decoratore`:
- `@decoratore`
`def f(x, y):`
 `...`
- Equivale a definire `f` come `decorator(f)`
- E se il nostro decoratore necessitasse di un argomento (e.g., dove salvare i log, dimensione massima della cache, etc)?

Decoratori

Funzioni che ritornano decoratori

- È possibile definire delle funzioni che ritornano decoratori...
- ...in pratica sfruttiamo le closure per costruire un decoratore diverso a seconda degli argomenti
- Questi sono poi chiamati con:
 - `@decoratore(argomenti)`
`def f(...)`
 - Equivale a
`decoratore = d(argomenti)`
`@decoratore`
`def f(...)`

Decoratori

Piccole note

- Data una funzione possiamo accedere a diverse informazioni come nome, docstring, etc:
 - e.g., `f.__name__` restituisce il nome della funzione
- Usare i decoratori “maschera” il nome vero della funzione
- Usare `@wraps(f)` di `functools` ci permette di mantenere nome e docstring corrette anche usano i decoratori