PYCON IT 2026 · TALK

I generatori: da `yield` ad `asyncio`.

> *"Strade? Dove andiamo... non servono strade."*
>
> — Ritorno al Futuro · 1985

Fiscozen

Carlo Bertini · Francesco Panico

```python
async def main():
    await asyncio.sleep(1)
```

Cosa succede *davvero* quando scrivete `await`?

TIME CIRCUITS ON • IL PIANO DI VOLO

Oggi facciamo un viaggio nel tempo.
25 anni di evoluzione, da una PEP all'altra.

DESTINATION TIME	DIC · 2026 · PY 3.16
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	OTT · 2001 · PEP 255

Partiamo dal **2001** con la nascita di `yield`, e arriviamo fino a oggi — passando per ogni PEP che ha portato a `async/await`.

Due Python developer di Fiscozen.
Backend, infra, e troppe coroutine.

Francesco Panico

Tech Team Leader @ Fiscozen

github.com/panicofr

Francesco & Carlo

Carlo Bertini

Platform Leader @ Fiscozen

github.com/WaYdotNET

Il protocollo che sta dietro al `for`.

Iterable — implementa `__iter__()`, restituisce un Iterator.

Iterator — implementa `__next__()`, produce il valore successivo. A fine corsa solleva `StopIteration`.

```
it = iter([1, 2, 3])

next(it)   # 1
next(it)   # 2
next(it)   # 3
next(it)   # raise StopIteration
```

Sotto al cofano: un `while` con `next()`.

QUELLO CHE SCRIVI

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

QUELLO CHE PYTHON ESEGUE

```
_iter = iter([1, 2, 3])     # __iter__()
while True:
    try:
        x = next(_iter)     # __next__()
    except StopIteration:
        break
    print(x)
```

Il **for** è solo un consumatore. Il problema interessante è scrivere il **produttore**.

— DOMANDA —

E se l'iterator *non conoscesse ancora* i valori che dovrà produrre?

DESTINATION TIME	OTT · 2001 · PEP 255
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	· · · — — — · · ·

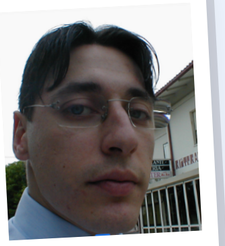
ATTO 2 · LA SOSPENSIONE · SPEAKER: FRANCESCO

CAPITOLO I

Simple Generators.

Una funzione che impara a mettersi in pausa.
Il punto di partenza di tutto.

Francesco
2001



Carlo
2001

PEP
255

2001 Python 2.2

Un generatore è una funzione capace di sospendersi.

```
def count():
    yield 1
    yield 2
    yield 3
```

```
c = count()
next(c)    # 1
next(c)    # 2
next(c)    # 3
next(c)    # raise StopIteration
```

Chiamare `count()` **non** restituisce 1.
Restituisce un oggetto generatore che prende vita ad ogni `next()`.

Variabili locali, posizione nel codice, stack frame: tutto viene congelato.

```
def demo():
    x = 10
    yield x     # x = 10, pause here
    x += 1
    yield x     # x = 11, restart from here
```

```
d = demo()
next(d)    # 10
next(d)    # 11  ← x NON riparte da 10
next(d)    # raise StopIteration
```

Quando rientriamo nel generatore, lo **scongeliamo** al punto esatto in cui si era fermato.

`return` non torna: solleva `StopIteration`.
Il valore viaggia come attributo dell'eccezione.

```
def demo():
    x = 10
    yield x     # x = 10, paused here
    return 42
```

```
d = demo()
next(d)                 # 10
next(d)                 # raise StopIteration: 42
sys.last_exc.value      # 42
```

Tenete a mente questo dettaglio: **ci tornerà utile** fra qualche slide, con `yield from`.

— MODELLO MENTALE —

Un generatore ~~non è una funzione che ritorna più valori~~.
È una funzione che può essere sospesa.

> *"Grande Giove! Una funzione che si ferma... e riparte dal futuro."*
>
> — ispirato a Ritorno al Futuro

— DOMANDA —

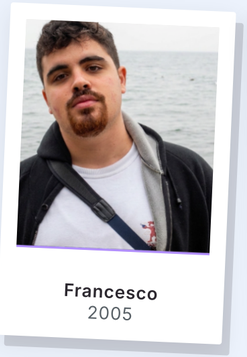
Aspetta... non è così che funziona una *coroutine*?

DESTINATION TIME	DIC · 2005 · PEP 342
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	OTT · 2001 · PEP 255

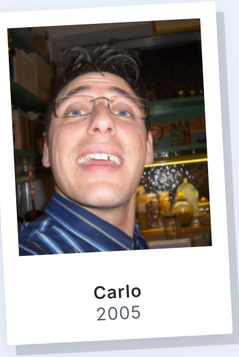
ATTO 3 · LA COMUNICAZIONE · SPEAKER: CARLO
CAPITOLO II

Coroutines via Enhanced Generators.

Stesso costrutto, tre nuovi superpoteri:
`send()` · `throw()` · `close()`.



Francesco
2005



Carlo
2005

PEP
342

2005 Python 2.5

`yield` non è più uno statement: è un'espressione.

← BEFORE · SOLO STATEMENT

```
yield value
```

→ AFTER · ESPRESSIONE

```
received = yield value
# emette value · riceve da fuori
```

"A yield-expression is like an inverted function call." — PEP 342

```python
def accumulator():
    total = 0
    while True:
        value = yield total
        total += value
```

```python
acc = accumulator()
next(acc)          # 0
acc.send(10)       # 10
acc.send(5)        # 15
acc.send(100)      # 115
```

La coroutine non si limita a produrre: **riceve dei dati** dal chiamante.
È una **conversazione**.

`close()` — terminazione pulita.
`throw()` — iniezione di eccezioni.

```python
def worker():
    try:
        while True:
            yield "working"
    except GeneratorExit:
        print("cleanup!")
```

```python
w = worker()
next(w)                 # "working"
w.close()               # "cleanup!"

w = worker()
next(w)
w.throw(ValueError("boom"))
```

Non solo dialogo: anche **controllo del ciclo di vita**.

01 GENERATORE — PRODUCE

```
def countdown(days):
    for d in range(days, 0, -1):
        yield d
```

Flusso: **unidirezionale**
Interfaccia: **next() · for**

02 COROUTINE — RICEVE & ELABORA

```
def fermentation(days):
    for d in range(days, 0, -1):
        command = yield
        if command == "stop":
            return d
```

Flusso: **bidirezionale**
Interfaccia: **send() · throw() · close()**

— MULTITASKING COOPERATIVO —

La coroutine decide DOVE sospendersi. Il chiamante decide QUANDO riprenderla.

`yield` · `next() / send()`

> *"Se ti impegni davvero, puoi ottenere tutto quello che vuoi."*
>
> — Ritorno al Futuro Parte III

🍺 BREWMASTER

Mettiamo insieme tutto con un esempio: facciamo la birra.

```python
def fermentation(days):
    """Fermentation in days"""
    for d in range(days, 0, -1):
        yield d
```

```python
def brew():
    """start → fermentation → ready"""
    yield "Start brewing"
    for v in fermentation(3):    # delega naïve
        yield v
    yield "🍺 Beer ready!"
```

In esecuzione: 5 `next()`, 5 valori.

```
def fermentation(days):
    for d in range(days, 0, -1):
        yield d

def brew():
    yield "Start brewing"
    for v in fermentation(3):
        yield v
    yield "🍺 Beer ready!"
```

```
b = brew()
next(b)   # → "Start brewing"
next(b)   # → 3
next(b)   # → 2
next(b)   # → 1
next(b)   # → 🍺 Beer ready!
```

— DOMANDA —

Come possiamo *interrompere* la fermentazione?

Il chiamante può mandare un comando dentro.

```
def fermentation(days):
    """Fermentation, can be stopped with send('stop')"""
    for d in range(days, 0, -1):
        command = yield d
```

`command = yield d` — emette `d` e riceve da fuori.

Se il comando è `"stop"`, `return d`.
Il valore di ritorno viaggia dentro la `StopIteration`.

```
def fermentation(days):
    """Fermentation, can be stopped with send('stop')"""
    for d in range(days, 0, -1):
        command = yield d
        if command == "stop":
            return d   # remaining days
```

Ricordate slide 10? **Eccolo il dettaglio.**

Il vecchio `brew` consuma i valori… ma il `send` non arriva a `fermentation`.

```
def brew():
    yield "Start brewing"
    for v in fermentation(3):
        yield v                          # ← il send si perde nel for
    yield "🍺 Beer ready!"
```

Per propagarlo, dobbiamo scrivere a mano tutto il **plumbing**…

Quindici righe. Per propagare un `send`.

```
def brew():
    yield "Start brewing"
    sub = fermentation(3)
    value = next(sub)
    while True:
        try:
            command = yield value        # cede al chiamante, riceve comando
            value = sub.send(command)    # inoltra al sub-generator
        except StopIteration as exc:
            yield f"🛑 Stopped at day {exc.value}" if exc.value else "🍺 Beer ready!"
            break
```

← Quindici righe... per *una sola idea*:
"inoltra tutto al sub".

Funziona. Ma a che prezzo?

```
def brew():
    yield "Start brewing"
    sub = fermentation(3)
    value = next(sub)
    while True:
        try:
            command = yield value
            value = sub.send(command)
        except StopIteration as exc:
            yield f"🛑 Stopped at day {exc.value}" if exc.value else "🍺 Beer ready!"
            break
```

```
b = brew()
next(b)          # → "Start brewing"
next(b)          # → 3
next(b)          # → 2
next(b)          # → 1
b.send("stop")   # → 🛑 Stopped at day 1
```

— DOMANDA —

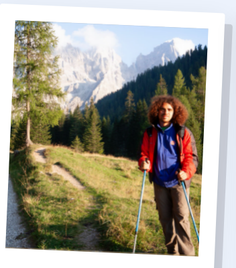
Come possiamo *evitare* il codice boilerplate?

DESTINATION TIME	GEN · 2012 · PEP 380
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	DIC · 2005 · PEP 342

CAPITOLO III

Syntax for Delegating to a Subgenerator.

Una keyword per delegare tutto: `yield from`.
Una riga al posto di quindici.



Francesco
2012

Carlo
2012

PEP
380

2012 Python 3.3

15 righe di plumbing.
Per fare una sola cosa: "inoltra tutto al sub-generator".

```
def brew():
    yield "Start brewing"
    sub = fermentation(3)
    value = next(sub)
    while True:
        try:
            command = yield value
            value = sub.send(command)
        except StopIteration as exc:
            yield f"🛑 Stopped at day {exc.value}" if exc.value else "🍺 Beer ready!"
            break
```

`yield from` — una riga al posto di quindici.

```python
def brew():
    yield "Start brewing"
    # delega COMPLETA: send, throw, close, return value
    remaining = yield from fermentation(3)
    yield f"🛑 Stopped at day {remaining}" if remaining else "🍺 Beer ready!"
```

```python
b = brew()
next(b)           # → "Start brewing"
next(b)           # → 3
next(b)           # → 2
next(b)           # → 1
b.send("stop")    # → 🛑 Stopped at day 1
```

Stesso comportamento. Zero plumbing.

PRIMA

```
def brew():
    ...
    for v in fermentation(3):
        yield v
```

DOPO

```
def brew():
    ...
    yield from fermentation(3)
```

OPERAZIONE	LATO CHIAMANTE	LATO SUB-GENERATOR
`next(g)`	avanza	fino al prossimo yield
`g.send(v)`	invia v	v diventa il valore dell'espressione yield
`g.throw(e)`	inietta eccezione	**`raise e`** al punto di yield
`g.close()`	chiude	**`GeneratorExit`** nel sub
`return v`	**`result = yield from sub`**	v risale al chiamante

I/O sincrono incapsulato in un generatore.

```
def check(name, ingredient):
    time.sleep(random.uniform(0.1, 1.5))   # simulates I/O, blocks
    return f"[{name}] ✓ {ingredient}"

def quality_check(name):
    for ingredient in ["malt", "hops", "water", "yeast"]:
        yield check(name, ingredient)
```

Ogni `yield` è un punto di sospensione: il chiamante riceve un check, può farci quello che vuole, e poi chiede il prossimo.

Una coroutine che riceve comandi.

```
def fermentation(name, days):
    """Can be stopped with send('stop')"""
    for d in range(days, 0, -1):
        time.sleep(1)
        command = yield f"[{name}] fermenting, {d} days left"
        if command == "stop":
            return d   # remaining days
```

`command = yield ...` — PEP 342 applicato al dominio.

`yield from` orchestra tutto.

```
def brew(name):
    yield from quality_check(name)
    remaining = yield from fermentation(name, 3)
    yield f"🛑 Stopped at day {remaining}" if remaining else "🍺 Beer ready!"
```

Quattro righe. **Tutto il flusso passa automaticamente**: next, send, throw, close, return value.

```python
def brew(name):
    yield from quality_check(name)
    remaining = yield from fermentation(name, 3)
    yield f"🛑 Stopped at day {remaining}" if remaining else "🍺 Beer ready!"

for step in brew("Pilsner"):
    print(step)
```

```
[Pilsner] ✓ malt
[Pilsner] ✓ hops
[Pilsner] ✓ water
[Pilsner] ✓ yeast
[Pilsner] fermenting, 3 days left
[Pilsner] fermenting, 2 days left
[Pilsner] fermenting, 1 days left
🍺 Beer ready!
```

Funziona. **Sequenziale, però.** Una birra alla volta.

— DOMANDA —

Come possiamo eseguire in *"parallelo"* più preparazioni?

DESTINATION TIME	2012 · ESPERIMENTO
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	GEN · 2012 · PEP 380

ATTO 7 · L'ORCHESTRAZIONE · SPEAKER: FRANCESCO

CAPITOLO IV

Mini event loop.

Multitasking cooperativo, costruito a mano con quello che abbiamo già.

Francesco
2012

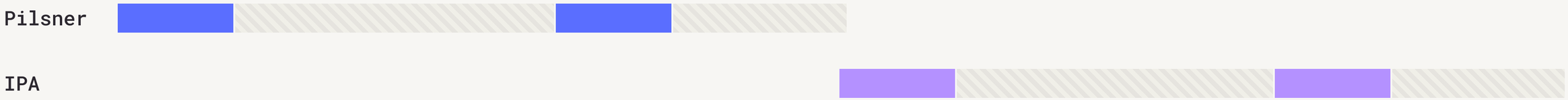
Carlo
2012

ESPERIMENTO
15 righe

`deque` · `next()`

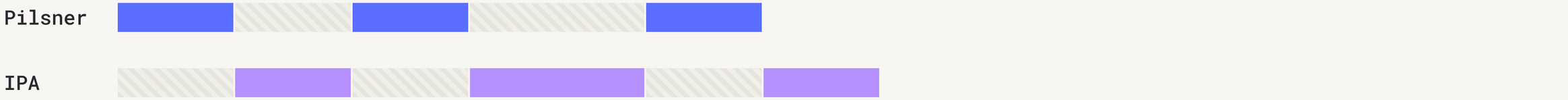
Stesso single thread.
Tempi morti riempiti.

✗ SEQUENZIALE — UN LOTTO ALLA VOLTA



tempo totale: 2×

✓ MULTITASKING COOPERATIVO



tempo totale: ~1×

Pilsner IPA in attesa di I/O

Un solo thread, mille task in attesa.

1

Coda dei task

deque, heap, priority queue.

→

2

Esegui 1 task

single thread, una coroutine alla volta.

→

3

Cede su `yield`

cooperazione esplicita → prossimo pronto.

→

4

Ripeti

finché la coda non è vuota.

Lo scriviamo noi. Quindici righe di Python.

```python
from collections import deque

coroutines = deque()

def enqueue_coro(coro):
    coroutines.append(coro)

def run():
    while coroutines:
        coro = coroutines.popleft()
        try:
            value = next(coro)          # avanza fino al prossimo yield
            print(value)
            coroutines.append(coro)     # rimette in coda
        except StopIteration:
            pass                        # coroutine finita
```

```
enqueue_coro(brew("Pilsner"))
enqueue_coro(brew("IPA"))
run()
```

```
[Pilsner]  ✓ malt
[IPA]      ✓ malt
[Pilsner]  ✓ hops
[IPA]      ✓ hops
...
[Pilsner]  ✓ yeast
[IPA]      ✓ yeast
[Pilsner]  fermenting, 3 days left
[IPA]      fermenting, 3 days left
...
🍺 Beer ready!
🍺 Beer ready!
```

Single thread, le due birre **avanzano insieme**. Niente magia: ognuna fa yield, il loop alterna.

— DOMANDA —

Mi devo implementare a mano l'event loop?

DESTINATION TIME	DIC · 2013 · PEP 3156
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	2012 · MINI LOOP

CAPITOLO V

Asynchronous IO, Rebooted.

La standard library ti dà tutto.
Si chiama `asyncio`.

Francesco
2013

Carlo
2013

PEP
3156

2013 — Python 3.3

Tre famiglie di metodi.
Ricordatevi queste, e avete il modello mentale dell'API.

Lifecycle

- `run_until_complete()`
- `run_forever()`
- `stop()` · `close()`
- `is_running()`

Tempo

- `time()`
- `call_later()`
- `call_at()`

Scheduling

- `call_soon()`
- `create_task()`
- `gather()`

Wrappa anche socket, file descriptor, pipe, signal — non li vediamo qui.

Un Task guida la coroutine.
Chiama `coro.send()`, riceve quando lei yielda.

01 · PENDING

Creato

Schedulato sull'event loop ma non ancora avviato. Nasce da `create_task()`.

→

02 · RUNNING

In esecuzione

Il loop sta eseguendo la coroutine. Quando lei yielda, il controllo torna al loop. Single thread: uno alla volta.

→

03 · DONE

Terminato

La coroutine ha fatto `return` o sollevato un'eccezione. `task.result()` restituisce o solleva.

↻ yield → cede al loop · un solo Task in RUNNING in ogni istante.

Un placeholder che yield-a se stesso e ritorna il risultato quando arriva.

L'event loop **non sa nulla di coroutine**. Il suo unico lavoro è schedulare callback.

Una coroutine asyncio sospende, restituisce, o solleva.
Niente altro.

SOSPENSIONE

```
result = yield from future
```

Sospende. Riceve il result o propaga l'eccezione.

DELEGA

```
result = yield from coroutine
```

Delega ad un'altra coroutine. Propaga risultato o eccezione.

RISULTATO

```
return expression
```

Produce il valore per chi ti aspetta con `yield from`.

ECCEZIONE

```
raise exception
```

Propaga un'eccezione a chi ti aspetta con `yield from`.

× Le coroutine possono fare yield solo di `Future`, non di valori generici.

`@asyncio.coroutine` + `yield from asyncio.sleep` non bloccano l'event loop.

```
import asyncio

@asyncio.coroutine
def quality_check(name):
    for ingredient in ["malt", "hops", "water"]:
        print(f"[{name}] ✓ {ingredient}")
        yield from asyncio.sleep(0.5)

@asyncio.coroutine
def fermentation(name, days):
    for d in range(days, 0, -1):
        print(f"[{name}] fermenting, {d} days left")
        yield from asyncio.sleep(1)
```

❌ BEFORE · BLOCCA IL LOOP

```
def fermentation(name, days):
    for d in range(days, 0, -1):
        time.sleep(1)                # ❌ blocca
        command = yield f"[{name}] {d} days left"
        if command == "stop":
            return d
```

✅ AFTER · CEDE AL LOOP

```
@asyncio.coroutine
def fermentation(name, days):
    for d in range(days, 0, -1):
        print(f"[{name}] {d} days left")
        yield from asyncio.sleep(1)    # ✅ cede
```

Ogni `yield from asyncio.sleep` dice al loop: **"fai pure altro mentre aspetto"**.

```python
@asyncio.coroutine
def brew(name):
    yield from quality_check(name)
    yield from fermentation(name, 3)
    print(f"[{name}] 🍺 Beer ready!")

loop = asyncio.get_event_loop()
batches = asyncio.gather(
    brew("Pilsner"),
    brew("IPA"),
)
loop.run_until_complete(batches)
loop.close()
```

```
[Pilsner]  ✓ malt
[IPA]      ✓ malt
[Pilsner]  ✓ hops
[IPA]      ✓ hops
[Pilsner]  ✓ water
[IPA]      ✓ water
[Pilsner]  fermenting, 3 days left
[IPA]      fermenting, 3 days left
...
[Pilsner]  🍺 Beer ready!
[IPA]      🍺 Beer ready!
```

— DOMANDA —

Come possiamo distinguere una *coroutine* da un *generator*?

La distinzione avviene a runtime, non a sintassi.

✓ YIELD FROM DI UN FUTURE

```
@asyncio.coroutine
def fetch():
    data = yield from http_get()
    return data
```

Il Task driver vede un Future yieldato → registra wakeup → tutto funziona.

✗ YIELD DI UN VALORE GENERICO

```
@asyncio.coroutine
def oops():
    yield 42          # yield valore!
```

Il Task driver riceve 42, non un Future → `RuntimeError: yielded non-future value`.

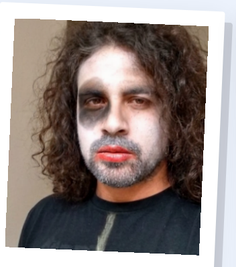
Sintatticamente identiche. **Tutta l'ambiguità che PEP 492 viene a risolvere.**

DESTINATION TIME	MAG · 2015 · PEP 492
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	DIC · 2013 · PEP 3156

ATTO 9 · LA SINTASSI DISAMBIGUA · SPEAKER: FRANCESCO
CAPITOLO VI

Coroutines with async and await.

Zucchero sintattico dedicato.
Niente più overload con i generatori.



Francesco
2015

Carlo
2015

PEP
492

2015 Python 3.5

Funzionalmente identiche. `await` è esplicito.

PYTHON ≥ 3.4, < 3.5

```
@asyncio.coroutine
def fetch():
    result = yield from http_get()
    return result
```

PYTHON ≥ 3.5

```
async def fetch():
    result = await http_get()
    return result
```

1. È una coroutine, non un generatore.
2. Accetta solo awaitable, non qualsiasi iterabile.
3. `next()` su coroutine nativa → `TypeError`.

Tre tipi di oggetti possono essere `await`-ati.

01

Coroutine object

Ritornato dalla chiamata di una coroutine function (definita con `async def`).

02

Generator-based

Una coroutine basata su generatori, decorata con `@types.coroutine`.

03

Awaitable object

Un oggetto che implementa il metodo `__await__`.

Tre tipi, una sintassi.

01

Coroutine nativa

Definita con `async def`. La forma standard.

```
async def fetch():
    return 42

result = await fetch()     # 42
```

02

Generator-based

Decorata con `@types.coroutine`. Legacy.

```
@types.coroutine
def legacy():
    yield "hi"

msg = await legacy()       # "hi"
```

03

Awaitable

Classe con `__await__` che yielda e ritorna.

```
class MyFuture:
    def __await__(self):
        yield
        return 42

v = await MyFuture()       # 42
```

Quattro buoni motivi.

🎯

Sintassi disambigua

`await` ≠ `yield` — niente più overload con i generatori.

🔍

Diagnostica

`"coroutine 'X' was never awaited" — warning automatico al GC.`

🛡️

Errori al parser

`await` fuori contesto → `SyntaxError` in compilazione, non a runtime.

⚡

Performance

`PyCoroObject` dedicato in CPython, più veloce di un generator puro.

Sintassi dedicata per I/O bound
a livello di blocco (`async with`) e iterazione (`async for`).

ASYNC WITH – GESTIRE RISORSE ASYNC

```python
# DB connection: __aenter__ / __aexit__
async with aiopg.connect(...) as conn:
    await conn.execute(...)

# Session HTTP
async with aiohttp.ClientSession() as s:
    async with s.get(url) as resp:
        data = await resp.json()
```

ASYNC FOR – ITERARE OGGETTI ASYNC

```python
# __aiter__ + async __anext__
class AsyncCounter:
    def __init__(self, n):
        self.n, self.i = n, 0
    def __aiter__(self): return self
    async def __anext__(self):
        if self.i >= self.n:
            raise StopAsyncIteration
        self.i += 1
        return self.i

async for x in AsyncCounter(3):
    print(x)
```

I dunder: `__aenter__` / `__aexit__` (with) · `__aiter__` / `__anext__` (for).

```
async def main():
    await asyncio.sleep(1)
```

Eccoci.
Abbiamo chiuso il cerchio.

Adesso lo sappiamo: cosa succede quando scrivete `await`?

`await` è il send-loop del PEP 342 con una sintassi dedicata.

01

Voi scrivete

La sintassi che vedete nel vostro codice.

```
async def main():
    result = await asyncio.sleep(1)
```

02

Dentro il Task

Il Task chiama `coro.send(None)` — la coroutine sospende e yielda un Future.

```
def __step(self):
    result = self.coro.send(None)
```

03

L'event loop

Registra una callback sul Future. Quando si completa, `__step` ricomincia.

```
future.add_done_callback(task.__step)
# quando il Future risolve:
#   __step ricomincia
```

— LA VERITÀ —

`async`/`await` non è magia.
È lo stesso `send()` del 2005,
la stessa delega di `yield from` del 2012,
lo stesso `Future.__iter__` del 2013.
È diventato uno standard del linguaggio.

> *"1.21 gigawatt!" — non è magia, è solo tanta energia ben incanalata.*
>
> — Doc Brown · Ritorno al Futuro

— DOMANDA —

Posso usare i generatori in un *contesto asincrono*?

DESTINATION TIME	SET · 2016 · PEP 525
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	MAG · 2015 · PEP 492

CAPITOLO VII

Asynchronous Generators.

Due paradigmi che per anni non si potevano combinare,
finalmente uniti in un costrutto solo.

Francesco
2016

Carlo
2016

PEP
525

2016 Python 3.6

Generatori — valori lazy. Coroutine — I/O non bloccante.
Per anni non si potevano combinare.

← PRIMA DEL PEP 525

```
async def fetch_stream(urls):
    for url in urls:
        yield await fetch(url)

# SyntaxError: 'yield' inside async function
```

yield e async def erano costrutti separati: errore durante il parsing.

→ CON PEP 525

```
async def fetch_stream(urls):
    for url in urls:
        yield await fetch(url)

# Valido! È un async generator
```

`yield` in `async def` → async generator. Consumato con `async for`.

— DOMANDA —

Tutto bello…
Ma *come lo usiamo*?

Stessa pipeline, ma con `async def + yield` → async generator.

← BEFORE · SYNC

```
def quality_check(name):
    for ingredient in ["malt", "hops", "water", "yeast"]:
        time.sleep(0.1)        # blocca
        yield f"[{name}] ✓ {ingredient}"
```

→ NOW · ASYNC GENERATOR

```
async def quality_check(name):
    for ingredient in ["malt", "hops", "water", "yeast"]:
        await asyncio.sleep(0.1)   # cede
        yield f"[{name}] ✓ {ingredient}"
```

← BEFORE · COROUTINE SYNC

```python
def fermentation(name, days):
    """Can be stopped with send('stop')"""
    for d in range(days, 0, -1):
        time.sleep(1)
        command = yield f"[{name}] {d} days left"
        if command == "stop":
            return d
```

→ NOW · ASYNC COROUTINE (PEP 525)

```python
async def fermentation(name, days):
    """Can be stopped with asend('stop')"""
    for d in range(days, 0, -1):
        await asyncio.sleep(1)
        command = yield f"[{name}] {d} days left"
        if command == "stop":
            return d
```

Stesso pattern del PEP 342, ma **async**: il driver chiama `asend()` invece di `send()`.

`async for` per produrre, `asend()` per dialogare.

```python
async def brew(name):
    async for status in quality_check(name):      # produce-only → async for
        print(status)

    gen = fermentation(name, 3)                    # bidirezionale → asend
    msg = await gen.asend(None)
    try:
        while True:
            print(msg)
            msg = await gen.asend(None)            # asend("stop") per interrompere
    except StopAsyncIteration:
        pass

    print(f"🍺 [{name}] Beer ready!")

asyncio.run(brew("IPA"))
```

Funziona. Ma **10 righe di plumbing** per delegare a una sub-coroutine. Vi suona familiare?

— DOMANDA —

Vi suona *familiare*?

la storia si ripete.

DESTINATION TIME	MAR · 2026 · DRAFT
PRESENT TIME	MAG · 2026 · PYCON
LAST TIME DEPARTED	SET · 2016 · PEP 525

CODA · IL FUTURO

CAPITOLO VIII · DRAFT

async yield from.

La stessa storia del 2012, ma async.
Una keyword, zero boilerplate.

Francesco
2026

Carlo
2026

PEP · DRAFT
828

2026 Python 3.16 · proposta

`async yield from` — una riga al posto di dieci.

← OGGI · PLUMBING MANUALE

```python
async def brew(name):
    async for status in quality_check(name):
        print(status)

    gen = fermentation(name, 3)
    msg = await gen.asend(None)
    try:
        while True:
            print(msg)
            msg = await gen.asend(None)
    except StopAsyncIteration:
        pass

    print(f"🍺 [{name}] Beer ready!")
```

→ DOMANI · ASYNC YIELD FROM

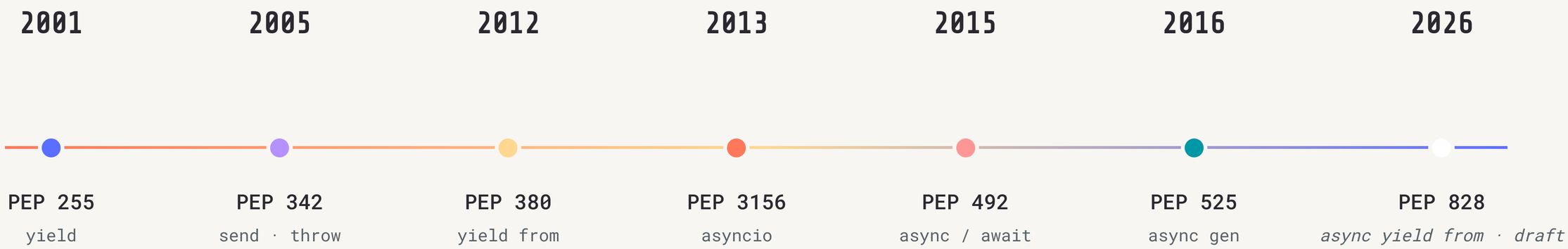
```python
async def brew(name):
    async for status in quality_check(name):
        print(status)

    # delega COMPLETA: asend, athrow, aclose, return
    remaining = async yield from fermentation(name, 3)

    print(f"🍺 [{name}] Beer ready!")
```

⚠ Draft, marzo 2026. Non è ancora nel linguaggio: è la proposta in discussione per Python 3.16. Ma il pattern è chiaro: la **storia** di yield from si ripete nel mondo async.

La timeline.



25 anni. Una sola idea: una funzione che si sospende e riprende.

— IL VIAGGIO IN 4 PAROLE —

I generatori ci hanno dato la sospensione.
`send()` ci ha dato la comunicazione.
`yield from` ci ha dato la composizione.
asyncio ci dà l'orchestrazione.

`async/await` non è magia.
È il risultato di **25 anni di evoluzione** del modello dei generatori.

> *"Il futuro non è ancora scritto: te lo crei tu."*
>
> — Ritorno al Futuro Parte III

Domande?

"Una coroutine non è una funzione che gira in parallelo.
È una funzione che decide quando fermarsi per lasciare spazio alle altre."

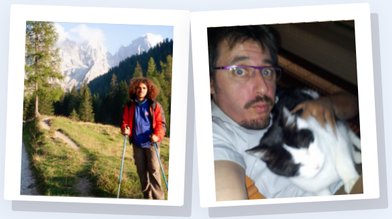
@WaYdotNET · @panicofr

— FINE DEL VIAGGIO —

Grazie per l'attenzione.

2001
PEP 255

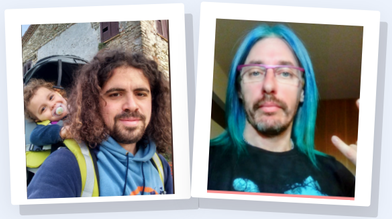
2005
PEP 342



2012
PEP 380 · Mini loop

2013
PEP 3156

2015
PEP 492



2016
PEP 525

2026
PEP 828 · draft

@WaYdotNET @panicofr