yield: Χρήση παραδείγματα.

Επιμέλεια: Α.Δρακόπουλος

Πίνακας περιεχομένων

Τι ακριβώς κάνει η εντολή `yield` στον παραπάνω κώδικα

Στον κώδικα:

```
yield nr, nc
```

η εντολή:

- **Δεν τερματίζει** τη συνάρτηση όπως κάνει το `return`
- **Παράγει μία τιμή** (ένα tuple `(nr, nc)`)
- **Σταματά προσωρινά την εκτέλεση** της συνάρτησης
- **Επιτρέπει στο πρόγραμμα να συνεχίσει** από το σημείο όπου σταμάτησε όταν ζητηθεί το επόμενο αποτέλεσμα

Έτσι η `neighbors(world, r, c)` **δεν επιστρέφει λίστα**, αλλά **γεννήτρια (generator)** που παράγει **μία-μία τις έγκυρες γειτονικές θέσεις**.

Αυτό επιτρέπει:

- **Χαμηλή κατανάλωση μνήμης**
- **Τεμπέλικη αξιολόγηση (lazy evaluation)**
- Ιδανικότητα για BFS, όπου δεν χρειάζονται **όλοι** οι neighbors ταυτόχρονα, αλλά **ένας-ένας**.

Συνοπτική Θεωρία για `yield`

1. Το `return` επιστρέφει *μία* τιμή και τερματίζει τη συνάρτηση

Το `yield` επιστρέφει *πολλαπλές* τιμές **χωρίς τερματισμό**.

2. Το `yield` μετατρέπει αυτόματα τη συνάρτηση σε generator function

Δηλαδή:

```
def f():
    yield 1
    yield 2
```

ισοδυναμεί με συνάρτηση που μπορεί να επαναληφθεί:

```
for x in f():
    print(x)
```

και θα εκτυπώσει:

```
1
2
```

3. Η συνάρτηση διατηρεί την κατάσταση της (state) ανάμεσα στα `yield`

Χρησιμοποιείται πολύ σε:

- BFS (neighbors)
- Traversal αλγορίθμους
- Επεξεργασία streams
- Load-on-demand data

4. Δεν δημιουργεί ολόκληρες λίστες — παράγει τιμές όταν χρειάζονται

Εξαιρετικά χρήσιμο για BFS γιατί:

- Ο BFS queue παίρνει neighbors **έναν-έναν**,
- Δεν χρειάζεται να περιμένουμε να φτιαχτεί ολόκληρη λίστα.

5. Πότε είναι απαραίτητο

- Όταν επιστρέφουμε *πολλά* στοιχεία *σταδιακά*
- Όταν δεν θέλουμε να επιβαρύνουμε τη μνήμη (π.χ. χιλιάδες neighbors σε γράφο)
- Όταν μια διαδικασία είναι “ροή” (streaming)
- Σε BFS/DFS για traversal δέντρων και grids

Πλεονεκτήματα

Πλεονέκτημα	Εξήγηση
Χαμηλή μνήμη	Δεν δημιουργεί λίστες, δίνει 1 τιμή κάθε φορά
Lazy evaluation	Υπολογίζει μόνο όταν χρειάζεται
Απλότητα	Γράφεις loops σαν να είναι λίστες
Ιδανικό για BFS	Γείτονες παράγονται διαδοχικά

Μειονεκτήματα

Μειονέκτημα	Εξήγηση
Δεν μπορεί να γίνει index	Δεν είναι λίστα
Δεν αποθηκεύει όλες τις τιμές	Αν θες όλες τις τιμές → list(generator)
Μία χρήση μόνο	Δεν μπορείς να το επαναχρησιμοποιήσεις αν καταναλωθεί

5 ΠΑΡΑΔΕΙΓΜΑΤΑ ΧΡΗΣΗΣ ΤΟΥ `yield`

Παράδειγμα 1 — Generator για φυσικούς αριθμούς

Εκφώνηση

Γράψτε συνάρτηση που παράγει τους φυσικούς αριθμούς 0–4.

Λύση

```
def numbers():
    for i in range(5):
        yield i

for x in numbers():
    print(x)
```

Σχόλια

- Η συνάρτηση **παράγει** 0, 1, 2, 3, 4 ένα-ένα.
- Δεν δημιουργείται λίστα → οικονομία μνήμης.
- Ο χρήστης παίρνει κάθε τιμή όταν τη χρειάζεται.

Παράδειγμα 2 — Generator για ανάγνωση αρχείου γραμμή-γραμμή

Εκφώνηση

Να γραφτεί συνάρτηση που διαβάζει ένα αρχείο *τεμπέλικα*.

Λύση

```
def read_lines(path):
    with open(path) as f:
        for line in f:
            yield line.strip()
```

Σχόλια

- Το αρχείο δεν φορτώνεται όλο στη μνήμη.
- Χρήσιμο για πολύ μεγάλα logs.

Παράδειγμα 3 — Generator που παράγει μόνο ζυγούς αριθμούς

Εκφώνηση

Να παραχθούν όλοι οι ζυγοί αριθμοί μέχρι το 10.

Λύση

```
def even_numbers(n):
    for i in range(n+1):
        if i % 2 == 0:
            yield i

for x in even_numbers(10):
    print(x)
```

Σχόλια

- Επιστρέφει *μόνο* τα κατάλληλα στοιχεία.
- Δεν φτιάχνει λίστα `[0,2,4,6,8,10]`.

Παράδειγμα 4 — Generator που παράγει γείτονες (όπως στο robot)

Εκφώνηση

Γράψτε generator που δίνει τις 4 κατευθύνσεις γύρω από `(r,c)`.

Λύση

```
def nbrs(r, c):
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        yield (r+dr, c+dc)

for p in nbrs(5, 5):
    print(p)
```

Σχόλια

- Ακριβώς η λογική που χρησιμοποιεί το BFS στο robot project.
- Η yield επιτρέπει επεξεργασία *καθώς παράγονται* οι neighbors.

Παράδειγμα 5 — Infinite generator (άπειρη ακολουθία)

Εκφώνηση

Γράψτε generator που δίνει άπειρη ακολουθία φυσικών αριθμών.

Λύση

```
def infinite_counter():
    i = 0
    while True:
        yield i
        i += 1

for x in infinite_counter():
    if x > 5:
        break
    print(x)
```

Σχόλια

- Το yield μπορεί να υποστηρίξει άπειρες ακολουθίες γιατί δεν χρειάζεται μνήμη.
- Αυτό θα ήταν *αδύνατο* με μια κανονική λίστα.

Σύνοψη

- Το `yield` κάνει τη συνάρτηση **γεννήτρια**.
- Η συνάρτηση εκτελείται *σταδιακά*, όχι όλη μαζί.
- Είναι ιδανικό για δόμηση **neighbor iterators**, όπως στο BFS.
- Είναι απαραίτητο όταν δεν θέλουμε να παράγουμε ολόκληρες λίστες.
- Εξοικονομεί μνήμη, είναι καθαρό και εκφραστικό.

Παράδειγμα 6 — Yield σε φίλτρο λίστας (filtering generator)

Εκφώνηση

Γράψτε generator που δέχεται μια λίστα αριθμών και παράγει *μόνο* εκείνους που είναι μεγαλύτεροι από ένα όριο.

Λύση

```
def filter_greater_than(nums, threshold):
    for n in nums:
        if n > threshold:
            yield n

for x in filter_greater_than([1,5,3,10,2], 4):
    print(x)
```

Έξοδος

```
5
10
```

Σχόλια

- Η yield επιτρέπει να επιστρέφουμε ένα-ένα τα “έγκυρα” στοιχεία χωρίς να κατασκευάζουμε νέα λίστα.
- Εξαιρετικό για μεγάλα datasets ή continuous streams.

Παράδειγμα 7 — Yield σε παραγωγή χαρακτήρων μίας συμβολοσειράς

Εκφώνηση

Γράψτε συνάρτηση που παράγει κάθε χαρακτήρα ενός string, έναν κάθε φορά.

Λύση

```
def characters(s):
    for ch in s:
        yield ch

for c in characters("robot"):
    print(c)
```

Σχόλια

- Αποδεικνύει ότι `yield` μπορεί να χρησιμοποιηθεί σαν “iterator builder”.
- Μπορεί να χρησιμοποιηθεί για parsing, tokenizing κειμένου ή streams.

Παράδειγμα 8 — Generator που επιστρέφει ζεύγη (pairs) από λίστα

Εκφώνηση

Γράψτε generator που επιστρέφει ζεύγη στοιχείων από μια λίστα, π.χ. `(nums[i], nums[i+1])`.

Λύση

```
def pairwise(nums):
    for i in range(len(nums)-1):
        yield (nums[i], nums[i+1])

for p in pairwise([10, 20, 30, 40]):
    print(p)
```

Έξοδος

```
(10, 20)
(20, 30)
(30, 40)
```

Σχόλια

- Αυτό το μοτίβο χρησιμοποιείται σε αλγορίθμους που απαιτούν γειτονικά στοιχεία (π.χ. smoothing, compression, path analysis).
- Η yield επιτρέπει “ροή” ζευγών χωρίς να φτιάξουμε μια λίστα με όλα τα pairs.

Παράδειγμα 9 — Generator με state που συνεχίζεται μεταξύ yield

Εκφώνηση

Υλοποιήστε generator που κρατάει άθροισμα που αυξάνεται συνεχώς κάθε φορά που καλείται.

Λύση

```
def cumulative_sum(nums):
    total = 0
    for n in nums:
        total += n
        yield total

for x in cumulative_sum([1, 2, 3, 4]):
    print(x)
```

Έξοδος

```
1
3
6
10
```

Σχόλια

- Δείχνει καθαρά το μεγάλο πλεονέκτημα των generators: **διατηρούν state ανάμεσα στα yield**.
- Ο μηχανισμός αυτός χρησιμοποιείται σε streaming statistics, online algorithms κ.λπ.

Παράδειγμα 10 — Generator που παράγει υπολίστες (sliding window)

Εκφώνηση

Γράψτε generator που δίνει όλα τα “παράθυρα” μήκους 3 μιας λίστας.

Λύση

```
def sliding_window(nums, k=3):
    for i in range(len(nums) - k + 1):
        yield nums[i:i+k]

for w in sliding_window([1,2,3,4,5], 3):
    print(w)
```

Έξοδος

```
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
```

Σχόλια

- Εξαιρετικό παράδειγμα για data processing (time-series, sensor data).
- Η yield επιτρέπει streaming windows αντί να τα κατασκευάζουμε όλα ταυτόχρονα.

ΜΕΡΟΣ Α’ — Πώς λειτουργούν οι Generators εσωτερικά (State Machines)

Οι **γεννήτριες (generators)** της Python δεν είναι απλά “ειδικές συναρτήσεις”.
Εσωτερικά είναι **state machines** — μικρές μηχανές που:

- διατηρούν την κατάστασή τους (local variables, position στο code, execution state),
- σταματούν και συνεχίζουν την εκτέλεση,
- παράγουν τιμές χωρίς να τερματίζουν.

Ας δούμε **ακριβώς** πώς δουλεύουν.

1. Όταν μια συνάρτηση περιέχει `yield`, δεν είναι πλέον “κανονική συνάρτηση”

Παράδειγμα:

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

Όταν καλώ:

```
g = counter()
```

Δεν εκτελείται τίποτα ακόμα.

Η Python δημιουργεί **generator object**.
Αυτό το αντικείμενο έχει:

- internal instruction pointer
- local variables
- hidden state frame

2. Η πρώτη εκτέλεση γίνεται όταν ζητήσεις την πρώτη τιμή

Με:

```
next(g)
```

η Python:

1. μπαίνει στη συνάρτηση
2. τρέχει μέχρι το πρώτο `yield`
3. επιστρέφει την τιμή στο `yield`
4. ΠΑΓΩΝΕΙ την εκτέλεση (όλα τα locals αποθηκεύονται)

Η κατάσταση μοιάζει ως εξής:

Στοιχείο	Τι περιέχει
instruction pointer	Σε ποια γραμμή σταμάτησε
local variables	Ό,τι είχαν μέχρι το yield
stack frame	Πλήρης κατάσταση εκτέλεσης

3. Με το επόμενο `next(g)` συνεχίζει από το ΣΗΜΕΙΟ που σταμάτησε

Δεν ξεκινά από την αρχή.
Δεν μηδενίζει μεταβλητές.

Μοιάζει με **paused coroutine**.

4. Όταν φτάσει τέλος συνάρτησης → `StopIteration`

Αυτό σημαίνει ότι ο generator ολοκληρώθηκε.

Παραλληλισμός με Μηχανή Καταστάσεων (State Machine)

Ας πούμε ότι ο generator είναι:

```
def gen():
    x = 10               # state init
    yield x              # state 1
    x += 1
    yield x              # state 2
    x *= 2
    yield x              # state 3
```

Εσωτερικά αναπαρίσταται περίπου ως:

```
STATE 0: x = 10 → goto STATE 1
STATE 1: yield x → goto STATE 2
STATE 2: x += 1 → yield x → goto STATE 3
STATE 3: x *= 2 → yield x → END
```

Με κάθε `next()` η κατάσταση μετάβασης αλλάζει.

Παράδειγμα State Machine (Ολοκληρωμένο)

Κώδικας

```
def simple_gen():
    print("Start")
    yield 1
```

```
print("Middle")
yield 2
print("End")
```

Εκτέλεση βήμα-βήμα

1. `g = simple_gen()`
   → Δεν τρέχει τίποτα.
2. `next(g)`
   - Τυπώνει "Start"
   - Σταματά στο `yield 1`
   - Επιστρέφει `1`
3. `next(g)`
   - Συνεχίζει από τη γραμμή *μετά* το yield
   - Τυπώνει "Middle"
   - Σταματά στο `yield 2`
   - Επιστρέφει `2`
4. `next(g)`
   - Τυπώνει "End"
   - Τελειώνει
   - Εξαίρεση `StopIteration`

ΜΕΡΟΣ Β’ — Σύγκριση `yield` με `return`, iterators & coroutines

1. `yield` vs `return`

Χαρακτηριστικό	`return`	`yield`
Αριθμός τιμών	1	Πολλές
Τερματίζει συνάρτηση;	Ναι	Όχι
Διατηρεί state;	Όχι	Ναι
Μπορεί να παράγει άπειρες τιμές;	Όχι	Ναι
Ιδανικό για	Κανονικές συναρτήσεις	Iterators, BFS, data streams

Παράδειγμα

```
def f():
    return 5
    return 6  # never executed
```

Αντίθετα:

```
def g():
    yield 5
    yield 6
```

2. `yield` vs manual iterator class

Χωρίς yield:

```
class Counter:
    def __init__(self):
        self.x = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.x > 5:
            raise StopIteration
        v = self.x
        self.x += 1
        return v
```

Με yield:

```
def counter():
    for i in range(6):
        yield i
```

Η `yield` **εξοικονομεί ~10 γραμμές κώδικα** και κάνει το ίδιο.

3. `yield` vs coroutines (async/await)

Στοιχείο	Generators	Coroutines
Μηχανισμός	yield	await
Χρήση	Iteration, BFS, streams	Async I/O, concurrency
Pause/resume	ναι	ναι (event loop)
Μοίρασμα δεδομένων	gen.send()	return values, awaits
Σύνθετο;	Απλό	Συνθετότερο

Οι coroutines είναι **γενικευμένη μορφή** generators:
Και τα δύο παγώνουν και συνεχίζουν εκτέλεση.

Παράδειγμα που συγκρίνει generator με coroutine

Generator (yield)

```
def gen():
    print("A")
    yield 1
    print("B")
    yield 2
```

Coroutine (async/await)

```
async def coro():
    print("A")
    await asyncio.sleep(1)
    print("B")
```

Σχόλια (Ελληνικά)

- Ο generator σταματά σε `yield`.
- Το coroutine σταματά σε `await` *μέσα στο event loop*.
- Ο μηχανισμός pause/resume είναι παρόμοιος, αλλά οι **χρήσεις** είναι διαφορετικές.

5 παραδείγματα για κατανόηση

Παράδειγμα 1 — Μετατροπή λίστας σε generator

Εκφώνηση:
Γράψτε συνάρτηση που παράγει τα στοιχεία μιας λίστας ένα-ένα.

```
def to_generator(lst):
    for x in lst:
        yield x

for e in to_generator([10,20,30]):
    print(e)
```

Σχόλια:
Δείχνει τη βασική χρήση: μετατροπή list → streaming iterator.

Παράδειγμα 2 — Yield σε streaming επεξεργασία (map)

Εκφώνηση:
Να προστεθεί 1 σε κάθε αριθμό και να παραχθεί αποτέλεσμα.

```
def add_one(nums):
    for n in nums:
        yield n + 1

for x in add_one([4,5,6]):
    print(x)
```

Σχόλια:

- Παρόμοιο με `map()` αλλά πιο ευέλικτο.

Παράδειγμα 3 — Generator για chunking λίστας

Εκφώνηση:
Παράγετε τμήματα (chunks) μήκους 2.

```
def chunks(nums, k=2):
    for i in range(0, len(nums), k):
        yield nums[i:i+k]

for ch in chunks([1,2,3,4,5], 2):
    print(ch)
```

Σχόλια:
Χρήσιμο σε επεξεργασία δεδομένων, batching.

Παράδειγμα 4 — Generator που αντικαθιστά recursion

Εκφώνηση:
Παράγετε αριθμούς μέχρι το N χωρίς recursion.

```
def upto(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Σχόλια:

- Ο generator λειτουργεί σαν “iterative recursion”.

Παράδειγμα 5 — Generator με input μέσω `.send()`

Εκφώνηση:
Γράψτε generator που δέχεται τιμές από τον χρήστη.

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

gen = accumulator()
print(next(gen))     # start
print(gen.send(5))
print(gen.send(3))
```

Σχόλια:

- Δείχνει ότι generators μπορούν να λειτουργούν σαν **coroutines**.
- Ένα από τα πιο προχωρημένα χαρακτηριστικά της Python.