Lazy Evalution

Bas Terwijn

February 16, 2026

1 Introductie

Dit document behandelt lazy evaluation.

1.1 Doelen

Bekend raken met:

- generator expressions
- generator functions

2 List Comprehension vs. Generator Expression

Een list comprehension schrijf je met vierkante haken `[ ... ]` en een generator expression met ronde haken `( ... )`. Beide constructies lijken erg op elkaar, maar het verschil is dat een list comprehension direct een lijst in het geheugen aanmaakt (eager evaluation), terwijl een generator expression een generator object aanmaakt dat pas waarden aanmaakt als deze nodig zijn (lazy evaluation). We kijken naar enekele voorbeelden om dit verschil te illustreren.

list comprehension (eager):

src/values_eager.py

```python
n = 5
values = [i for i in range(5)]

print(values)  # [0, 1, 2, 3, 4]
print(values)  # [0, 1, 2, 3, 4]
print(values)  # [0, 1, 2, 3, 4]
```

Met een list comprehension maken we direct een lijst in het geheugen aan. We kunnen deze waarden meerder keren uit de lijst lezen.

generator expression (lazy):

src/values_lazy.py

```python
n = 5
values = (i for i in range(5))
print(values)  # <generator object ...>
print(list(values))  # [0, 1, 2, 3, 4]
print(list(values))  # []   generator is used up
print(list(values))  # []   generator is used up
```

Met een generator expression maken we een generator object aan. Als we die printen zien we geen waarden maar zoiets als `<generator object <genexpr> at 0x7484eef45a80>`. We kunnen met `list(values)` wel de waarden uit de generator lezen en in een lijst stoppen, maar als we eenmaal een waarde uit de generator hebben gelezen, dan is deze opgebruikt. Vandaar dat we een lege lijst krijgen als we `list(values)` een tweede keer aanroepen.

3 Lazy Evaluatie

Ook de volgorde waarop dingen gebeuren is anders bij lazy evaluation. Om dit te zien definieren we de print functie `pr(tag, v)` functie. Deze functie print de tag en de waarde, en returned deze waarde. Hiermee kunnen we gaan zien wanneer waarden worden aangemaakt en wanneer ze worden gebruikt.

src/helpers.py

```python
def pr(tag, v):
    """ Print 'tag' and 'v' and return 'v' """
    print(tag, v)
    return v

if __name__ == '__main__':
    a = pr('test:', 123)  # prints 'test: 123' and also returns 123
    print('a:', a)
```

```
test: 123
a: 123
```

list comprehension (eager):

src/values_eager_pr.py

```python
from helpers import pr

n = 5
values = [pr('create:', i) for i in range(5)]
print('list is created')
for i in values:
    pr('use:', i)
```

```
create: 0
create: 1
create: 2
create: 3
create: 4
list is created
use: 0
use: 1
use: 2
use: 3
use: 4
```

Als we met `pr` printen wanneer we een waarde aanmaken en wanneer we een waarde gebruiken, dan zien we dat bij de list comprehension eerst alle waarden worden aangemaakt. Als dat is gebeurt is daarna ook de complete lijst aangemaakt. Deze lijst kan vervolgens worden gebruikt in bijvoorbeeld een for-loop.

generator expression (lazy):

src/values_lazy_pr.py

```python
from helpers import pr

n = 5
values = (pr('create:', i) for i in range(5))
print('generator is created')
for i in values:
    pr('use:', i)
```

```
generator is created
create: 0
use: 0
create: 1
use: 1
create: 2
use: 2
create: 3
use: 3
create: 4
use: 4
```

Als we ditzelfde doen met een generator expression, dan zien we, met enige verassing, dat de generator al is aangemaakt voordat één waarden is aangemaakt. Een generator is lazy (letterlijk lui) en maakt een waarde pas aan op het moment dat deze echt nodig is. De for-loop vraagt herhaaldelijk een waarde op waardoor de generator steeds op het laatste moment de volgende waarde aanmaakt net voordat deze waarde gebruikt wordt.

4 Lazy Evaluatie Voordelen

Als we voor `n` een hele grote waarde kiezen, bijvoorbeeld `10**9` of `10**12`, dan krijgen we problemen bij eager evaluation:

- Er is veel geheugen nodig om eerst alle waarden op te slaan, misschien wel meer dan er beschikbaar is in een computer, crash!
- Het duurt heel lang om eerst de hele lijst aan te maken voordat we de waarden gaan gebruiken.

Het voordeel van lazy evaluation is dat er zeer weinig geheugen nodig is, omdat een waarde pas wordt aangemaakt als deze nodig is. Ook kunnen we de eerste waarden meteen gebruiken zonder te hoeven wachten tot opvolgende waarden zijn aangemaakt. Lazy evaluation is dus vooral nuttig voor programma's die met grote hoeveelheden data werken.

5 Generator Expression en Generator Function

Python heeft twee manieren om een generator aan te maken:

- Met een **generator expression**, wat lijkt of een list comprehension, maar dan met ronde haakjes.
- Met een **generator function**, wat een gewone functie is die in plaats van met `return` met `yield` een waarde teruggeeft.

generator expression (lazy):

src/gen_expr.py

```
def generator_expression(n):
    return (i for i in range(n))


n = 5
gen = generator_expression(n)
print(gen) # <generator_expression at 0x7...>
print(list(gen))  # [0, 1, 2, 3, 4]
print(list(gen))  # []   generator is used up
gen = generator_expression(n)
print(list(gen))  # [0, 1, 2, 3, 4] new values
```

generator function (lazy):

src/gen_fun.py

```
def generator_function(n):
    for i in range(n):
        yield i

n = 5
gen = generator_function(n)
print(gen)  # <generator_function at 0x7...>
print(list(gen))  # [0, 1, 2, 3, 4]
print(list(gen))  # []   generator is used up
gen = generator_function(n)
print(list(gen))  # [0, 1, 2, 3, 4] new vales
```

6 Generator Function Voorbeeld

Dit voorbeeld laat duidelijker zien hoe een generator function werkt.

src/generator_fun.py

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

n = 5
gen = generator_function(n)
for i in gen:
    print(i, end=' ')  # 0 1 2 3 4
```

Als we een generator function aanroepen dan krijgen we een generator object terug. We kunnen dit generator object gebruiken in bijvoorbeeld een for-loop om de waarden op te vragen. De generator runt dan t/m het eerste `yield` statement wat een waarde terug geeft aan de for-loop. De volgende keer dat er weer een waarde wordt opgevraagd door de for-loop, gaat het generator object verder waar het gebleven was t/m het eerstvolgende `yield` statement.

De waarde van alle variabelen binnen deze `generator_function()`, zoals `i`, blijft hierbij behouden tussen de verschillende opvragingen. De for-loop is afgelopen wanneer de generator functie returnt, dus nadat `i` niet meer kleiner is dan `n`. Dit wordt gecommuniceerd door een `StopIteration` exception die automatisch door Python wordt afgehandeld in de for-loop.

7 Opdracht `lazy.py`

Er komt een oneindige stroom van pakketjes binnen die we moeten vervoeren met vrachtwagens. De software voor dit proces is al geimplemeneerd met behulp van eager evaluation in function `fill_trucks()` in file `eager.py`. Het voert de volgende stappen uit:

- lees elk pakketje met een (x,y,z) grootte in centimeters uit de oneindige stroom pakketjes
- bereken het volume van elk pakketje in kubieke meters

- alleen pakketjes met een volume groter dan MINIMAL_VOLUME=0.1 m^3 gaan in een vrachtwagen, de andere gaan met de post mee
- vul de huidige vrachtwagen totdat het totale volume groter is dan TRUCK_VOLUME_CAPACITY=20 m^3, begin daarna met vullen een nieuwe lege vrachtwagen
- nadat alle NR_TRUCKS=5 vrachtwagens gevuld zijn rijden zij weg en eindig het programma omdat we moeten wachten tot ze zijn teruggekeerd

Als we dit `eager.py` programma uitvoeren zien we:

```
$ python eager.py
truck1: [2.085, 1.014, 1.697, 0.262, 0.51, 0.258, 3.844, 0.625, 1.481, 0.265, 2.086, 0.926,
1.023, 0.457] sum: 16.533 m3
truck2: [5.04, 1.013, 0.258, 0.408, 0.321, 1.353, 0.31, 0.434, 1.686, 1.158, 1.274, 0.364,
0.752, 0.176, 1.381, 0.257, 2.457] sum: 18.642 m3
truck3: [1.735, 0.458, 2.838, 1.227, 3.562, 0.364, 0.396, 3.058, 0.265, 0.31, 0.363, 0.388,
0.288, 3.716] sum: 18.968 m3
truck4: [1.246, 0.463, 0.442, 0.213, 0.521, 0.277, 1.165, 0.17, 0.716, 0.352, 2.093, 1.15,
1.806, 0.107, 0.503, 0.308, 0.877, 2.008, 1.229, 1.099, 1.02, 0.343, 0.121, 0.425] sum:
18.654 m3
truck5: [1.772, 1.178, 3.939, 0.5, 2.054, 1.369, 0.219, 1.094, 3.054, 0.141, 2.707] sum:
18.027 m3
```

Maar we zijn ontevreden over dit `eager.py` programma omdat veel verschillende taken in de `fill_trucks()` functie worden uitgevoerd wat voor coupling tussen deze taken zorgt. We willen deze taken opsplitsen in aparte functies die elk een enkele taak uitvoeren, en daarbij lazy evaluation gebruiken om geheugen te besparen en sneller te kunnen starten met vullen van de vrachtwagens. Implementeer hiervoor de functies in file `lazy.py` zodat het hetzelfde resultaat geeft als `eager.py`, maar nu met lazy evaluation. Daarbij kun je tijdelijk de `pr()` functie gebruiken om tussenresultaten te printen zodat je kunt zien of deze kloppen, maar verwijder deze print statements weer als het programma werkt.

Test deze opdracht met:

```
$ pytest test_lazy.py
```