Python Advanced

Inhoudsopgave

1. Inleiding

In deze reader wordt dieper op Python ingegaan en wordt achtergrond-informatie gegeven.

De volgende onderwerpen komen aan bod:

- iterator
- generator
- onderwerpen uit functionele talen
- closure
- decorator

2. Iterator.

Het afdrukken van de elementen van een lijst a kan als volgt:

```
for i in range(len(a)):
    print(a[i], end = ' ')
```

De code lijkt sterk op de C-code

```
for(int i=0; i<size ; i++) // size: aantal elementen van a
{
    printf("%d ",a[i]);
}
```

In Python kan minder omslachtig een lijst a worden doorlopen:

```
for e in a:
   print(e,end = ' ')
```

Ook sets, strings en files kunnen op deze manier doorlopen worden.
Dergelijke klassen worden 'iterable' genoemd.

Technisch is een klasse 'iterable' als het de methode '__iter__()' bevat.

Bovenstaande code heeft de volgende 'low_level'-equivalent:

```
it = iter(a) # a.__iter__()

end_of_iteration = False

while not end_of_iteration:
   try:
      print(next(it), end = ' ' )
   except StopIteration:
     end_of_iteration = True
```

De variabele 'it' is een iterator-object.

Meer informatie staat op de sites:

- https://docs.python.org/3.6/glossary.html#term-iterator
- https://docs.python.org/3.6/library/stdtypes.html#typeiter

Een iterator heeft de volgende methoden:

- __iter__() : geeft zichzelf terug. Dit geeft aan dat een iterator zelf iterable is.
- __next__(): geeft het volgende element terug (de eerste aanroep geeft het eerste element terug). Als er geen volgend element is, wordt een StopIteration-exception opgeworpen

De aanroep ' it.__iter__() ' is hetzelfde als ' iter(it) '
De aanroep ' it.__next__() ' is hetzelfde als ' next(it) '

3. 'all' en 'any'.

De functies 'all' en 'any' zijn ingebouwde functies.

- all(iterable) : geeft True terug als alle elementen waar of ongelijk aan nul zijn.
- any(iterable) : geeft True terug als tenminste één element waar of ongelijk aan nul is

Zie: https://docs.python.org/3/library/functions.html#all en
https://docs.python.org/3/library/functions.html#any

Het volgende voorbeeld gaat na of alle elementen uit de lijst deelbaar door 3 zijn.

```
a = [9,0,6,3,27,18]
print(all([not e%3 for e in a]))
```

Uitvoer:

```
True
```

De volgende opdracht gaat na de lijst minstens één drievoud bevat:

```
a = [10,1,6,2,28,17]
print(any([not e%3 for e in a]))
```

4. Oneindigheid.

Interessant is het begrip oneindigheid. In de wiskunde wordt oneindig genoteerd met ∞ (lemniscaat, unicode 221E). Oneindigheid gaat het voorstellingsvermogen te boven. Dat geldt trouwens ook voor de tegenpool 'het absolute niets' (gapende leegte, zwart gat) . In de wiskunde worden hiervoor de begrippen 0 en ∅ (lege verzameling, unicode 2300) gebruikt. Bij exacte wetenschappen, zoals wiskunde en natuurkunde, spelen deze begrippen een belangrijke rol. Denk daarbij aan reeksen, limieten en integralen.

Zie ook de sites:
https://nl.wikipedia.org/wiki/Oneindigheid
https://nl.wikipedia.org/wiki/Niets

In Python zijn 0, 0.0, [] , (), {} en '' (lege string) gedefinieerd. De objecten zijn ook gekoppeld aan de boolean-waarde False.

Python heeft twee implementaties van oneindig

- float('inf')
- math.inf

Een demo:

```
import math
infinity = math.inf
print('2**10000 < infinity:     ', 2**10000 < infinity)
print('infinity:                ', infinity)
print('infinity+5:              ', infinity+5)
print('infinity-5:              ', infinity-5)
print('infinity+infinity:       ', infinity+infinity)
print('0*infinity:              ', 0*infinity)
print('infinity-infinity:       ', infinity-infinity)
print('-infinity:               ', -infinity)
print('(-1)*infinity:           ', (-1)*infinity)
print('(-infinity)*(-infinity):',(-infinity)*(-infinity))
```

Uitvoer:

```
2**10000 < infinity:     True
infinity:                inf
infinity+5:              inf
infinity-5:              inf
infinity+infinity:       inf
0*infinity:              nan
infinity-infinity:       nan
-infinity:               -inf
(-1)*infinity:           -inf
(-infinity)*(-infinity): inf
```

'nan' staat voor 'not a number' . In de wiskunde zijn de uitdrukkingen '0*∞' en '∞ – ∞' niet gedefinieerd.

Zie ook de sites:

https://nl.wikipedia.org/wiki/NaN
https://docs.python.org/dev/library/math.html

5. Oneindige iterators

De module 'itertools' bevat iterators zonder einde.
Zie https://docs.python.org/3/library/itertools.html

- itertools.count(start=0, step=1) : teller vanaf 'start' met stapgrootte 'step'
  de stapgrootte hoeft niet een geheel getal te zijn.
- itertools.cycle(iterable) : cyclische herhaling van een iterable
- itertools.repeat(object[, times]) : herhaling van een object

count

'count' is een teller . Standaard begint de teller bij 0 en heeft stapgrootte 1.

Een demo:

```
it = itertools.count(20,5)  # start: 20, step: 5
for _ in range(10):
    print(next(it),end = ' ')
```

Uitvoer:

```
20 25 30 35 40 45 50 55 60 65
```

De opdracht

```
it = itertools.count()
a = list(it)
```

is niet handig: er wordt een lijst gemaakt met oneindig veel elementen. Dit leidt tot een MemoryError-exception.

Een iterator bevat alleen het huidige element en de stapgrootte.
Bij een lijst staan alle elementen in het geheugen.
Met lijsten moet je dus voorzichtig zijn.

Je kunt een eindige deelrij krijgen door toepassing van 'slicing'.
Dit kan met

```
itertools.islice(iterable, stop)
itertools.islice(iterable, start, stop[, step])
```

Een voorbeeld:

```
it = itertools.count(step=5)
isl = itertools.islice(it,20,31)
a = list(isl)
print(a)
```

Uitvoer:

```
[100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150]
```

cycle

Een cycle-object loopt een iterable object steeds cyclisch door.

Demo 1:

```
a = [1,2,3]
it = itertools.cycle(a)
for _ in range(10):
    print(next(it),end = ' ')
print()
```

Uitvoer:

```
1 2 3 1 2 3 1 2 3 1
```

Demo 2:

```
a = [1,2,3]
it = itertools.cycle(a)
isl = itertools.islice(it,0,20,2)
a = list(isl)
print(a)
```

Uitvoer:

```
[1, 3, 2, 1, 3, 2, 1, 3, 2, 1]
```

repeat

Een repeat-object herhaalt een object een aantal keer. Het aantal kan ook onbegrensd zijn.

Demo:

```
a = list(itertools.repeat([3],5))
print(a)
a[0][0] = 4
print(a)
print([id(e) for e in a])
```

Uitvoer:

```
[[3], [3], [3], [3], [3]]
[[4], [4], [4], [4], [4]]
[79151232, 79151232, 79151232, 79151232, 79151232]
```

In feite wordt een verwijzing naar een object herhaald. Wordt het eerste element gewijzigd, dan worden ook de andere elementen gewijzigd

Hetzelfde gebeurt bij het volgende voorbeeld:

```
a = [[3]]*5
print(a)
a[0][0] = 4
print(a)
print([id(e) for e in a])
```

Uitvoer:

```
[[3], [3], [3], [3], [3]]
[[4], [4], [4], [4], [4]]
[79151312, 79151312, 79151312, 79151312, 79151312]
```

Je kunt dit als volgt oplossen:

```
a = [[3] for _ in range(5)]
print(a)
a[0][0] = 4
print(a)
print([id(e) for e in a])
print()
```

Uitvoer:

```
[[3], [3], [3], [3], [3]]
[[4], [3], [3], [3], [3]]
[58114416, 58037392, 52962168, 58038472, 58037112]
```

6. Generators.

Een generator is een speciale functie:

- in plaats van een return-opdracht wordt een yield-opdracht gebruikt
- een generator geeft een generator-object terug.

Een voorbeeld:

```
def createGenerator():
    for i in range(3):
        print('before yield',i)
        yield i*i
        print('after yield',i)

it = createGenerator()
print(next(it))
print('=============')
print(next(it))
print('=============')
print(next(it))
print('=============')
print(next(it))
```

Uitvoer:

```
before yield 0
0
=============
after yield 0
before yield 1
1
=============
after yield 1
before yield 2
4
=============
after yield 2

StopIteration-exception
```

Bij de eerste aanroep wordt de code vanaf het begin tot aan de yield-opdracht uitgevoerd en de yield-waarde teruggegeven. Bij de volgende aanroepen wordt vanaf de yield-opdracht tot de volgende yield-opdracht de code uitgevoerd.

Het volgende voorbeeld doorloopt alle positieve gehele getallen.

```
def integers():
    n = 1
    while True:
        yield(n)
        n += 1;

a = integers()
print(next(a))
print(next(a))
print(next(a))
```

Uitvoer:

1
2
3

Generator-expressies.

Een voorbeeld van een generator expressie is:

```
(e*e for e in range(1,11))
```

Het is een verkorte schrijfwijze van:

```
def createGenerator():
    for i in range(1,11):
        yield i*i

g = createGenerator()
```

7. Onderwerpen uit functionele talen.

Anonieme functie.

De volgende functie berekent het kwadraat van een gegeven getal:

```
def square(x):
    return x*x
```

Je kunt de functie 'square' ook als volgt declareren:

```
square = lambda x : x*x
```

De uitdrukking '`lambda x : x*x`' is een voorbeeld van een anonieme functie.
Lambda-expressies worden gebruikt bij functionele talen.
Zie https://en.wikipedia.org/wiki/Anonymous_function
Een lambda-expressie is niet gekopeld aan een naam (identifier) .

De functie 'filter'.

Een voorbeeld:

Uit een gegeven rij willen we de positieve getallen selecteren.

Een oplossing:

```
a = [1,-2,3,-4,5,-6,7]
b = [e for e in a if e > 0]
```

Dit kan ook opgelost worden met de ingebouwde functie **filter(function, iterable)**

Een oplossing is:

```
a = [1,-2,3,-4,5,-6,7]
it = filter(lambda x : x > 0, a)
b = list(it)
```

Het object dat een filter-functie teruggeeft is een iterator.
Zie https://docs.python.org/3/library/functions.html#filter

Een filter-functie is een functie waarbij één van de parameters een functie is.

Als de filter-functie i.p.v. een functie de waarde 'None' als parameter heeft worden de waarden, met boolean-waarde TRUE (d.w.z. ongelijk aan 0, 0.0, "", None, [] , etc.) geselecteerd.

Een voorbeeld:

```
a = [1,-2,0,-4,5,0.0,7,None,-8,"",6,[],-14]
it = filter(None, a)
b = list(it)
print(b)
```

Uitvoer:

```
[1, -2, -4, 5, 7, -8, 6, -14]
```

De functie 'map'.

Ga uit van de opdracht: vervang ieder element uit een lijst door het het kwadraat van het element.

Dit kan als volgt gerealiseerd worden:

```
a = [1, -2, 4, -6, 8, -10]
b = [e*e for e in a]
print(b)
```

Uitvoer:

```
[1, 4, 16, 36, 64, 100]
```

Met de map-functie gaat dit als volgt:

```
it = map(lambda x : x*x,a)
b = list(it)
```

Nog een voorbeeld:

```
a = [1, -2, 4, -6, 8, -10]
it = map(lambda x : x>0,a)
b = list(it)
print(b)
```

Uitvoer:

```
[True, False, True, False, True, False]
```

Opdracht: een string bevat een aantal getallen. Plaats deze getallen in een lijst.

Het volgende voorbeeld laat nog meer mogelijkheden van de functie 'map' zien.

```
a1 = [1, 2, 3]
a2 = [7, 8, 9]
it = map(lambda x,y : x*y,a1,a2)
b = list(it)
print(b)
```

Uitvoer:

```
[7, 16, 27]
```

Vraag: wat gebeurt er als de lengte van a1 niet gelijk is aan de lengte van a2?

Kunnen we bovenstaand probleem ook op andere manieren oplossen?

Methode 1.

```
m = min(len(a1),len(a2))
b = []
for i in range(m):
    b.append(a1[i]*a2[i])
```

De volgende oplossing werkt niet:

Methode 2.

```
b = [x*y for x in a1 for y in a2]
```

Vraag: waarom niet?

Je kunt het wel oplossen met de zip-functie.

De functie 'zip'.

Met de functie 'zip' kan op basis van twee rijen een lijst van 2-tupels gemaakt worden.

Een voorbeeld:

```
a1 = [1, 2, 3]
a2 = [7, 8, 9]
it = zip(a1,a2)
b = list(it)
print(b)
```

Uitvoer:

```
[(1, 7), (2, 8), (3, 9)]
```

Je kunt de zip-functie ook toepassen op meerdere rijen:

```
a1 = [1, 2, 3]
a2 = [7, 8, 9]
a3 = [-1,-2,-3]
it = zip(a1,a2,a3)
b = list(it)
print(b)
```

Uitvoer:

```
[(1, 7, -1), (2, 8, -2), (3, 9, -3)]
```

'Unzippen' is mogelijk met de '*'-operator:

```
a1 = [1, 2, 3]
a2 = [7, 8, 9]
it = zip(a1,a2)
b = list(it)
print('zip:', b)
it2 = zip(*b)
b2 = list(it2)
print('unzip:',b2)
c_a1,c_a2 = b2
print(c_a1,c_a2)
```

Uitvoer:

```
zip: [(1, 7), (2, 8), (3, 9)]
unzip: [(1, 2, 3), (7, 8, 9)]
(1, 2, 3) (7, 8, 9)
```

De opdrachten

```
it = map(lambda x,y : x*y,a1,a2)
b = list(it)
```

kunnen vervangen worden door:

```
b = [x*y for x,y in zip(a1,a2)]
```

Opgave:

De volgende opdrachten maken op basis van een lijst 'a' een tupel-lijst 'b' .

```
a = [1, -2, 4, -6, 8, -10]
enum = enumerate(a)
b = list(enum)
print(b)
```

Uitvoer:

```
[(0, 1), (1, -2), (2, 4), (3, -6), (4, 8), (5, -10)]
```

Hoe krijgen we op basis van 'b' de lijst 'a' terug? Gebruik de map-functie.

De functie 'reduce' .

Citaat van Guido van Rossum:

"So now reduce(). This is actually the one I've always hated most, because, apart from a few examples involving + or *, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what the reduce() is supposed to do. So in my mind, the applicability of reduce() is pretty much limited to associative operators, and in all other cases it's better to write out the accumulation loop explicitly."

Bij Python 3 is de functie 'reduce' geplaatst in de module 'functools' .

De functie heeft de volgende vorm:

```
reduce(func, seq)
```

Als seg = [s1,s2, s3, …. , sn] , wordt achtereenvolgens het volgende berekend:

- f1 = func(s1,s2)
- f2 = func(f1,s3)
- f3 = func(f2,s4)
- etc

Het eindresultaat wordt teruggegeven.

Een voorbeeld:

```
import functools

def mysum(a):
    return functools.reduce(lambda x,y: x+y,a)

a = [1,2,3,4,5]
print(mysum(a))
```

Uitvoer:

```
15
```

Nog een voorbeeld:

```
def mymax(a):
    return functools.reduce(lambda x,y: x if x>y else y,a)

a = [1,6,13,5,3]
print(mymax(a))
```

Uitvoer:

```
13
```

8. Klassen en functies.

In Python is alles een object.
Dit geldt voor getallen, maar ook voor functies.

Attributen zijn in Python in principe 'public'. Het is mogelijk een attribuut te verstoppen, maar het blijft (indirect) toegankelijk.

Een voorbeeld:

```
class MyClass:
    def public(self):
        pass
    def __private(self): # plaats '__' voor de naam
        print('private!')
    def __init__(self):
        self.attr1 = 100
        self.__attr2 = 101

mc = MyClass()
a = list(filter(lambda x: not (x.startswith('__') and x.endswith('__')),
                dir(mc)))
print(a)
print(mc.__dict__)
mc._MyClass__private()
print(mc._MyClass__attr2)
mc.attr3 = 101
print(mc.__dict__)
```

Uitvoer:

```
['_MyClass__attr2', '_MyClass__private', 'attr1', 'public']
{'attr1': 100, '_MyClass__attr2': 101}
private!
101
{'_MyClass__attr2': 101, 'attr3': 102, 'attr1': 100}
```

Omdat '**__private**' begint met '__' wordt '**__private**' intern vervangen door
'**_MyClass__private**'
Hetzelfde geldt voor '**__attr2**' .

Je ziet aan dit voorbeeld ook dat je aan object dynamisch attributen kunt toevoegen.
Je kunt dit blokkeren door '**__slots__**' te gebruiken.

Een voorbeeld:

```
class MyClass:
    def public(self):
        pass
    __slots__ = ['attr1' ]
    def __init__(self):
        self.attr1 = 100
#        self.__attr2 = 101   # niet toegestaan

mc = MyClass()
a = list(filter(lambda x: not (x.startswith('__') and x.endswith('__')),
                dir(mc)))
print(a)
# print(mc.__dict__)         # niet toegestaan
# mc.attr3 = 102             # niet toegestaan
```

Uitvoer:

```
['attr1', 'public']
```

'**__slots__**' bevat de attributen, die toegestaan zijn.
Een object van deze klasse heeft niet meer het attribuut '**__dict__**' .

Vraag: welke foutmelding krijg je als je aan een int-object een attribuut wil toevoegen?

Het is ook mogelijk aan functie attributen toe te kennen.

Een voorbeeld:

```
def cf():
    cf.count += 1
    print(cf.count)
cf.count = 0
cf()
cf()
a = list(filter(lambda x: not (x.startswith('__') and x.endswith('__')),
                dir(cf)))
print(a)
print(cf.__dict__)
```

Uitvoer:

```
1
2
['count']
{'count': 2}
```

9. Closures.

Voor closures gelden de volgende regels:

- er is een omvattende functie en een geneste functie
- de geneste functie verwijst naar een variabele buiten de functie-declaratie
- de omvattende functie geeft de geneste functie terug

Een voorbeeld:

```
def make_multiplier_of(n):
    return lambda x : x*n

times3 = make_multiplier_of(3)
times5 = make_multiplier_of(5)

print(times3(9))
print(times5(3))
print(times5(times3(2)))

print('times3.__closure__:',times3.__closure__)
      # bevat cellen uit de 'closure' van times3
print([e.cell_contents for e in times3.__closure__])
```

Uitvoer:

```
27
15
30
times3.__closure__: (<cell at 0x04144810: int object at 0x1D713730>,)
[3]
```

In dit voorbeeld is de omvattende functie 'make_multiplier_of' en
de geneste functie '`lambda x : x*n`'.
De geneste functie gebruikt de variabele 'n', die buiten de functie is gedeclareerd.

Een geneste functie heeft een __closure__- attribuut. Deze bevat cellen die de objecten buiten de scope bevatten.

10. Decorators.

Een decorator is een functie, waarbij een functie als parameter wordt meegegeven en een functie wordt teruggegeven

Door decorators te gebruiken kan code netjes worden ingedeeld. (separation of concerns)

Een voorbeeld:

```
def smart_divide(f):
    def inner(a,b):
        print("I am going to divide",a,"and",b)
        if b == 0:
            print("Whoops! cannot divide")
            return
        return f(a,b)
    return inner

def divide(a,b):
    return a/b

divide = smart_divide(divide)

print(divide(3,4))
print(divide(3,0))
```

Uitvoer:

```
I am going to divide 3 and 4
0.75
I am going to divide 3 and 0
Whoops! cannot divide
None
```

De functie **'smart_divide'** heeft functie **'f'** als parameter en functie **'inner'** als return-waarde.

De code

```
def divide(a,b):
    return a/b

divide = smart_divide(divide)
```

kan vervangen worden door een kortere schrijfwijze:

```
@smart_divide
def divide(a,b):
    return a/b
```

Voorbeeld 2:

```
def count(f):
    def inner(a,b):
        inner.counter += 1
        return f(a,b)
    inner.counter = 0
    return inner

@count
def myadd(a,b):
    return a+b

print(myadd(1,2))
print(myadd(3,4))
print(myadd(5,6))

print ('myadd.counter =',myadd.counter)
```

Uitvoer:

```
3
7
11
myadd.counter = 3
```

De count-functie werkt alleen voor functies met twee parameters.

Een voorbeeld van een generieke functie:

```
def echo(*args, **kwargs):
    print(args, kwargs)

echo(10, 20, a=30, b=40)
```

Uitvoer:

```
(10, 20) {'a': 30, 'b': 40}
```

Een generalisatie van 'count' is:

```
def count(f):
    def inner(*args, **kwargs):
        inner.counter += 1
        return f(*args, **kwargs)
    inner.counter = 0
    return inner
```

Voorbeeld 3:

```
def mylog(f):
    def inner(*args, **kwargs):
        print('function:',f.__name__)
        print('args:',args)
        print('kwargs:',kwargs)
        return f(*args, **kwargs)
    return inner

@mylog
def myadd2(a,b=10):
    return a+b

print(myadd2(1,2))
print(myadd2(3,4))
print(myadd2(5))
print(myadd2(7,b=20))
```

Uitvoer:

```
function: myadd2
args: (1, 2)
kwargs: {}
3
function: myadd2
args: (3, 4)
kwargs: {}
7
function: myadd2
args: (5,)
kwargs: {}
15
function: myadd2
args: (7,)
kwargs: {'b': 20}
27
```

Opdracht: Maak een decorator die bij een gegeven functie de tijdsduur van een function-call bepaalt.