Universiteit van Amsterdam

Programmeertalen

Python

Jouke Witteveen

Sudoku

1	2	3	7	8	9	4	5	6
4	5	6	1	2	3	7	8	9
7	8	9	4	5	6	1	2	3
2	3	1	8	9	7	5	6	4
5	6	4	2	3	1	8	9	7
8	9	7	5	6	4	2	3	1
3	1	2	9	7	8	6	4	5
6	4	5	3	1	2	9	7	8
9	7	8	6	4	5	3	1	2

Deadline: Zondag 28 februari 2021, 23:59

Puzzels

Implementeer de opgaven 1 t/m 4 in Python versie 3 op een zo *pythonic* mogelijke manier in bestand `puzzels.py`:

```
import collections

def opgave1(mylist):
    pass

def opgave2(mylist):
    pass

def opgave3a(filename):
    pass

def opgave3b(mylist):
    pass

def opgave3(filename):
    opgave3b( opgave3a( filename ) )

def sum_nested_it(mylist):
    pass
```

> Wanneer je op CodeGrade test, lever dan alleen de implementatie van bovenstaande functies in en niet ook je testcode.

Opgave 1

Schrijf functie `opgave1` die, met als input een reeks van `n` integers, test of deze reeks alle integers van 1 t/m `n` bevat.

Opgave 2

Schrijf een *generator* genaamd `opgave2` die een reeks van `n` integers als input neemt en alle integers van 1 t/m `n` produceert die geen deel zijn van die reeks.

Opgave 3a

Schrijf functie `opgave3a` die een lijst van lijsten van integers uit een bestand leest, één lijst per regel. Elke regel van het bestand bevat een rij getallen, gescheiden door een enkele spatie. Als input geef je de naam van het bestand mee. Gebruik de volgende constructie:

```
with open(filename) as f:
    # 'f' is nu een iterator die over the regels van 'filename' itereert
```

Stel dus dat je een bestand meegeeft met de volgende inhoud:

```
1 2
5 10
```

dan zou je functie de lijst `[[1, 2], [5, 10]]` moeten teruggeven.

Opgave 3b

Schrijf functie `opgave3b` die een lijst van lijsten van integers als input krijgt, en per lijst een regel met een rij van getallen print die gescheiden zijn door een enkele spatie. Doe dit op zo een manier dat functie `opgave3` in feite gewoon het ingevoerde bestand print, maar tussendoor de ingelezen getallen naar en van een nuttige interne representatie omzet.

Opgave 4

In Python bestaat de functie `sum`: deze functie telt alle waardes van een lijst bij elkaar op. Als we echter een lijst hebben als `[[1,2], 3]` werkt deze functie niet, want `[1,2]` en `3` kunnen niet bij elkaar opgeteld worden. Stel dat we wel ondersteuning willen bieden voor dit soort gevallen, dan zouden we een functie kunnen schrijven zoals hieronder weergegeven:

```
import collections

def sum_nested_rec(lst):
    if isinstance(lst, collections.abc.Iterable):
        return sum(sum_nested_rec(item) for item in lst)
    else:
        return lst
```

Dit soort recursieve functies wordt vaak gezien als niet *idiomatisch* binnen Python, gedeeltelijk omdat dit in Python zeer inefficient is. Daarom is het voor deze opgave de bedoeling om de functie `sum_nested_rec` om te schrijven naar de functie `sum_nested_it`, die geen gebruikt maakt van recursie, globale variabelen of andere libraries dan de `collections` library.

Deze functie moet dus van een geneste lijst van getallen alle elementen nemen en die bij elkaar optellen. Het resultaat van `sum_nested_it([[1], 2, 3, [4, 5, [6]]])` is dan 21.

Inleveren

Lever alleen het bestand `puzzels.py` in op CodeGrade.

Intermezzo – Wat is Pythonic?

We willen dat je Python-code zo *pythonic* mogelijk is, maar het begrip pythonic is redelijk vaag en soms lastig te bevatten. Daarom gaan we in dit voorbeeld toewerken naar een pythonic implementatie van een functie die de getransponeerde van een gegeven matrix teruggeeft. Deze functie hoeft alleen voor 2D matrices te werken.

Laten we eerst een naïeve implementatie maken, zoals we misschien in C zouden schrijven.

```
def transpose_try_one(matrix):
    res = []
    for i in range(len(matrix)):
        res.append([])
    while matrix:
        i = 0
        row = matrix[0]
        while row:
            res[i].append(row[0])
            i += 1
            row = row[1:]
        matrix = matrix[1:]
    return res
```

Dit zou een redelijk normale implementatie zijn in een taal als C (ook niet goed, maar dit soort dingen zie je wel vaak). Deze code is echter totaal niet pythonic. Het eerste probleem is dat er geloopt wordt over de `range` van een `len`, dit kan veel mooier gedaan worden door de lijst zelf heen te lopen. Als de index wel nodig is kan je beter de functie `enumerate` gebruikt worden. Ook is het gebruikelijk om een variabele die je niet gebruikt `_` te noemen, pas wel op deze naam heeft geen speciale betekenis in python!

Daarna wordt weer over de matrix geloopt, maar dit wordt gedaan met een `while` loop, dit is in python iets wat echt niet kan (al is het in C ook steeds minder normaal). Een `while` loop gebruik je in Python echt alleen als je een bepaalde conditie hebt – dus niet om elke element van een lijst te behandelen, daar gebruik je `for el in lijst` voor. Ook in deze loop wordt weer een counter bijgehouden, wat zoals eerder beschreven niet netjes is.

Laten we met deze kennis een nieuwe iteratie van onze functie maken:

```
def transpose_try_two(matrix):
    res = []
    for _ in matrix:
        res.append([])
    for row in matrix:
        for i, item in enumerate(row):
            res[i].append(item)
    return res
```

Deze functie ziet er al een stuk beter uit, maar is nog steeds niet helemaal pythonic. Meerdere keren doen we in een `for`-loop alleen `list.append`, waarmee we dus een lijst opbouwen. Hiervoor heeft Python een korter en leesbaarder alternatief: de list-comprehension. Wanneer we de `for`-loops vervangen door een list-comprehension ziet onze functie er zo uit:

```
def transpose_try_three(matrix):
    return [[matrix[j][i] for j, _ in enumerate(matrix)]
            for i, _ in enumerate(matrix[0])]
```

Dit ziet er al een stuk leesbaarder uit. Toch is ook deze versie nog niet perfect. Dit kan je voornamelijk zien aan het feit dat we een directe index doen in `matrix`. Nu is dat in Python niet zo erg als bij functionele talen zoals Haskell

(daar betekent het meestal dat je iets erg fout aan het doen bent), maar toch zien we het liever niet. Gelukkig heeft Python nog twee trucjes die ons kunnen helpen om het nog mooier te maken. Als eerste hebben we in Python, net als in Haskell, een `zip`-functie. Als we `help(zip)` in onze Python REPL intypen dan zien we dat `zip` 1 of meer lijsten neemt en een lijst teruggeeft waarbij element i het i-de element van alle lijsten bevat. Die functie neemt dus al bijna de getransponeerde! Het probleem is nu alleen nog dat `zip` met één argument niet echt iets nuttigs doet. Gelukkig kunnen we dat oplossen met *list unpacking*. Onze functie ziet er nu zo uit:

```
def transpose_final(matrix):
    return zip(*matrix)
```

Deze functie is **wel** pythonic, maar helaas niet meer helemaal correct. De functie `zip` geeft namelijk geen lijst terug maar een zip-object (dit is effectief een iterator). Dit zou prima werken voor veel doeleinden, maar we kunnen nu niet meer iets doen als `transpose_try_four(lst)[0]`! Daarnaast geeft dit geen iterator over lijsten terug, maar over tuples, wat dus ook niet helemaal is wat we willen. Met deze informatie komen we tot onze uiteindelijke versie:

```
def transpose_try_four(matrix):
    return [list(a) for a in zip(*matrix)]
```

We zien zelfs bij zo'n simpele functie dat pythonic een lastig begrip kan zijn. Deels is het natuurlijk het optimaal gebruik maken van de functionaliteit van Python zoals deze is bedoeld, wat eigenlijk altijd mooiere en duidelijkere code oplevert. Maar meestal kan het daarna nóg korter, waarbij je vaak een balans moet zoeken tussen iteratief en functioneel programmeren. Doe dit alleen als dit niet ten koste gaat van de de duidelijkheid en leesbaarheid van je code. Waar bij Haskell korter eigenlijk altijd beter was, staat bij Python duidelijkheid voorop.

Sudoku

De Sudoku is een algemeen bekende puzzel; achtergrondinformatie en regels erover zijn makkelijk online te vinden. De bedoeling is dat je een algemene Sudoku-solver implementeert in het bestand `sudoku.py`. Dit programma krijgt als enige argument de naam van een bestand dat een (onopgeloste) Sudoku bevat van willekeurige grootte n, waarbij n een kwadraat is. Als deze Sudoku oplosbaar is, moet je programma er een oplossing voor geven.

Invoer-/uitvoerformaat (Ontwikkelend)

Een Sudoku van grootte n wordt gerepresenteerd door n regels van n door spaties gescheiden getallen, waarbij elke regel een rij van de Sudoku representeert. In de input representeert het getal 0 een lege cel. Het is dus de taak van je programma om alle nullen te vervangen door getallen, op zo'n manier dat het resultaat een geldige oplossing van de Sudoku is. Begin eerst met het schrijven van code voor de representatie, het inlezen, en het printen van de gegeven Sudoku-borden.

Het algoritme (Competent)

Het algoritme dat je implementeert om een Sudoku op te lossen moet door een *pruned* (gesnoeide) zoekboom van potentiële oplossingen heen lopen. Zodra een oplossing is gevonden, wordt de zoektocht afgebroken en de oplossing geprint.

> Hierbij komt dat je programma niet afhankelijk mag zijn van de *call stack* in zijn tocht door de zoekboom. Dit betekent dat elke vorm van recursie verboden is. Anders gezegd: **geen enkele** functie in je programma mag zichzelf aanroepen, direct noch indirect. Gebruik in plaats daarvan bijvoorbeeld een *while loop* en een eigen *stack* (`https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-stacks`) om tussenresultaat-borden in op te slaan. Het geheugengebruik van je programma moet redelijk zijn.

Het algoritme (Gevorderd)

Je programma mag niet zomaar elke waarde proberen in te vullen in een cel. Voor elke lege cel mag het alleen de waarden proberen die nog niet de rij, de kolom of het blok staan die bij de cel horen. Dus voor de Sudoku

1	2	3	4
0	0	2	1
2	1	4	3
0	4	1	2

zou een mogelijke zoekboom er zo uit kunnen zien:

```
.
|-- (0,1) = 3
|
`-- (0,1) = 4
    |
    `-- (1,1) = 3
        |
        `-- (0,3) = 3
```

maar afhankelijk van de volgorde waarop de lege cellen worden afgehandeld zijn er ook andere zoekbomen mogelijk.

Extra features (Expert)

Een expert gaat er natuurlijk niet van uit dat de ingevoerde Sudoku altijd geldig is. Zorg ervoor dat je programma afsluit met een (behulpzame!) foutmelding als de input niet correct geformatteerd is.

Meerdere oplossingen (Master)

Er zijn sudoku's[1] die meer dan 1 oplossing hebben, de lege sudoku is hier een voorbeeld van. Pas je solver zo aan dat deze meerdere oplossingen kan vinden, waarbij het aantal oplossingen bepaald wordt door het tweede argument aan je solver. Dus, als we het programma als volgt aanroepen `python3 sudoku.py sudoku 2`, dan moet de solver twee verschillende oplossingen van de gegeven sudoku printen. Als er minder oplossingen zijn dan het tweede argument aangeeft, dan worden alle oplossingen geprint. Als er geen argument is gegeven, wordt er één oplossing geprint.

Inleveren

Lever alleen het bestand `sudoku.py` in op CodeGrade.

Tests

Je solver wordt automatisch nagekeken met gebruik van pytest[2] en een subset van de tests is zichtbaar op CodeGrade. Flake8[3] wordt gebruikt voor de stijl.

[1] Eigenlijk wordt het dan geen sudoku genoemd, maar dat terzijde.
[2] `https://docs.pytest.org/en/stable/getting-started.html`
[3] `https://flake8.pycqa.org/en/latest/`