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: Individueel: Woensdag 27 februari 2019 23:59
Team: zondag 3 maart 2019, 23:59

Individueel

Implementeren de opgaven 1 t/m 4 in Python versie 3 op een zo *pythonic* mogelijke manier in bestand `individual.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
```

> Voor deze opgaven lever je alleen de implementatie van bovenstaande functies in. Je moet natuurlijk deze functies wel testen, maar het ingeleverde bestand bevat alleen deze functies en niet ook je test-code.

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.Iterable):
        return sum(sum_nested_rec(item) for item in lst)
    else:
        return lst
```

Dit soort recursieve functies wordt vaak niet gezien als *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.

Sudoku

Beschrijving

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.

Input/output formaat (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 0-en 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 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 op op te slaan. Het geheugengebruik van je programma moet redelijk zijn.

Het algoritme (Gevorderde)

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)

Om de maximale score te behalen, moet je programma afsluiten met een (behulpzame!) foutmelding als de input niet correct geformatteerd is. Ook moet er een kort rapport geschreven worden (van max. 3 pagina's) dat het effect analyseert van het snoeien van de zoekboom. Dit betekent dat je je programma vergelijkt met een versie die wel alle mogelijke getallen probeert in te vullen in de lege cellen of die op een slimmere manier de volgorde van invullen kiest. Je resultaten moeten ondersteund worden door experimenten die de prestatie van beide versies vergelijken.

De extra features zullen je alleen punten opleveren als de rest van je programma aan de eisen voldoet; het weergeven van een mooie foutmelding zal je dus niet helpen als je programma recursief geïmplementeerd is.

Voorbeeld

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 transpose functie (als je niet weet wat zo'n functie moet doen, moet je dat even opzoeken). Deze functie hoeft alleen voor 2D matrices te werken.

Laten we eerst een naïeve implementatie maken, iets wat 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 zoals 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, ook dit is niet netjes zoals eerder beschreven.

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. Het grootste probleem is dat we meerdere keren `list.append` doen als enige onderdeel van een `for` loop, dit betekent meestal dat dit de verkeerde manier is. Als je merkt dat je alleen een list aan het maken bent in een `for` loop, dan kan je deze code beter omschrijven naar een list comprehension. Met deze kennis kunnen we de volgende iteratie maken van onze functie:

```
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 die niet zo erg als bij functionele talen zoals Haskell (daar betekent het meestal dat je iets erg fout aan het doen bent), 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 returned waarbij element i het i-de element van alle lijsten bevat. Die functie neemt dus al bijna de transpose! Het probleem is nu alleen nog dat `zip` met 1 argument niet echt wat nuttigs doet, gelukkig kunnen we dat oplossen met *list unpacking*. Het lijkt erop dat onze functie dus zoals dit wordt:

```
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, echter kunnen we nu niet meer iets doen als `transpose_try_four(lst)[0]`! Ook geeft geen iterator over lijsten terug, maar over tuples, ook niet 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 met zo'n simpele functie dat pythonic een lastig begrip is. Het is de mix vinden tussen iteratief en functioneel programmeren. Als we van Haskell zouden kunnen zeggen dat kort beter is, zouden we van python kunnen zeggen dat duidelijk beter is.

Inleveren

Lever voor de individuele deadline `individual.py` in, en lever voor de team deadline `sudoku.py` in.