Algemeen

Oneindige lussen

Let op voor oneindige lussen. Een oneindige lus is een lus die eindeloos blijft uitgevoerd worden: in de meeste gevallen gaat het om een `while`-lus waarbij de statements binnen de lus er nooit voor zorgen dat de voorwaarde van de lus uiteindelijk `False` wordt. Bekijk als voorbeeld eens het volgend stuk code

```
>>> i = 0
>>> a = 0
>>> while i < 4:
...     a += 1
```

Omdat het statement `a += 1` er nooit zal voor zorgen dat de waarde van de variabele `i` groter dan of gelijk aan 4 wordt, zal de voorwaarde `i < 4` altijd de waarde `True` opleveren.

Tip: Als je werkt met Pycharm, dan kan je nagaan dat je programma aan het uitvoeren is, door het rode vierkantje links van de Console of rechtsboven in de menubalk. Als je op dit vierkantje klikt, dan forceer je om de uitvoering van het programma te stoppen.

Tellen vanaf nul

Informatici beginnen standaard te tellen vanaf 0, niet vanaf 1. Python volgt deze traditie op heel wat verschillende plaatsen. Zo genereert de ingebouwde functie `range` een reeks opeenvolgende getallen die begint bij 0, als je slechts één enkel argument meegeeft aan de functie. Zo tel je bijvoorbeeld standaard tot 5

```
>>> for i in range(6):
...     print(i)
...
0
1
2
3
4
5
```

Als je niet wil beginnen tellen vanaf 0, dan kan je de startwaarde meegeven als een tweede argument aan de `range` functie.

```
>>> for i in range(1, 6):
...     print(i)
...
1
2
3
4
5
```

Het wordt echter als meer *pythonic* beschouwd om het voorgaande op de volgende manier uit te schrijven

```
>>> for i in range(5):
...     print(i + 1)
...
1
2
3
4
5
```

Lussen vroegtijdig afbreken

Je kunt de statements `break` en `continue` gebruiken om een lus vroegtijdig af te breken. Deze statements worden echter algemeen aanzien als slechte programmeerstijl. Je kunt ze dus beter niet gebruiken, want ze zullen je punten kosten als je ze gebruikt op een evaluatie of een exam.

Een situatie waarin je een lus vroegtijdig zou willen afbreken, doet zich bijvoorbeeld voor als je op zoek moet gaan naar een oplossing door een aantal mogelijke gevallen uit te proberen, en te stoppen van zodra je één oplossing gevonden hebt. In plaats van te werken met `break` en `continue` kan je in dit geval beter werken met een variabele die aangeeft of de oplossing al gevonden werd

```
>>> gevonden = False
>>> while not gevonden:
...     if <oplossing gevonden>:  # <oplossing gevonden> stelt voorwaarde voor
...         gevonden = True
...
```

Van zodra de oplossing gevonden werd (hier aangegeven door het feit dat de voorwaarde *oplossing gevonden* de waarde `True` aanneemt), wordt de variabele `gevonden` op `True` gezet, waardoor uit de `while`-lus zal gesprongen wanneer de `while`-voorwaarde opnieuw geëvalueerd wordt na de huidige iteratiestap.

Inlezen van meerdere regels

Als je vooraf weet hoeveel regels er moeten ingelezen worden, dan kan je de volgende constructie gebruiken

```
>>> aantal_regels = 4
>>> for _ in range(aantal_regels):
...     regel = input()
...     # doe iets met de regel
...
```

Als je niet op voorhand weet hoeveel regels er zijn, maar je weet dat de laatste regel bijvoorbeeld leeg is, dan kan je de volgende constructie gebruiken

```
>>> regel = input()
>>> while regel:
...
...     # doe iets met de regel
...
...     regel = input()
...
```

Ronde getallen

Absolute waarde

De ingebouwde functie `abs` kan gebruikt worden om de absolute waarde van een getal te bepalen.

```
>>> abs(42)
42
>>> abs(-42)
42
>>> abs(3.14159)
3.14159
>>> abs(-3.14159)
3.14159
```

Liftparadox

Voorloopnullen toevoegen met f-strings

Als je bij het omzetten van een getal naar een string wil zorgen dat de string een vast aantal karakters heeft (waarbij er eventueel nullen worden toegevoegd aan het begin van de string), dan kan je een *format specifier* gebruiken. *Format specifiers* worden altijd tussen het paar accolades geplaatst dat gebruikt wordt als plaatshouder in de template string. Een *format specifier* begint altijd met een dubbelpunt (`:`).

Om een getal uit te schrijven dat een vast aantal posities inneemt, gebruik je de *format specifier* `:0nd`. De `0` staat er omdat we vooraan nullen willen toevoegen (je kan ook andere karakters toevoegen), de letter `d` staat voor een digit (getal) en `n` voor de lengte die de string moet innemen. Als het getal dat je wil omzetten naar een string langer is dan de gewenste lengte, dan wordt het volledige getal overgenomen. In dat geval zal de string dus langer zijn dan het gewenst aantal karakters.

```
>>> f'{2}'
'2'
>>> f'{2:02d}'
'02'
>>> f'{34:02d]'
'34'
>>> f'{567:02d}'
'567'
>>> f'{89:06d}'
'000089'
```

Er zijn nog andere manieren om voorloopnullen te genereren. Zo kan je ook de stringmethode `zfill` (*zero fill*) gebruiken. Je kan ook het verschil tussen de huidige lengte en de gewenste lengte berekenen en dan weet je hoeveel nullen je moet toevoegen. Of je kan ook werken met een `while` lus die nullen blijft toevoegen totdat de string de gewenste lengte heeft.

```
>>> gewenste_lengte = 3
>>> getal = str(2)
>>> getal.zfill(gewenste_lengte)
'002'
>>> aantal_nullen = gewenste_lengte - len(getal)
>>> aantal_nullen
2
>>> '0' * aantal_nullen + getal
'002'
>>> while len(getal) < gewenste_lengte:
...     getal = '0' + getal
...
>>> getal
'002'
```

Voorwaardelijke expressies

De meeste programmeertalen hebben een ternaire operator die de voorwaardelijke expressie genoemd wordt. Het resultaat van een voorwaardelijke expressie hangt af van een voorwaarde `test`. Python gebruikt de volgende syntax voor voorwaardelijke expressies:

```
<expr_true> if <test> else <expr_false>
```

Let er hierbij op dat de volgorde van de voorwaarde en de expressies verschilt van andere programmeertalen, waar eerst de voorwaarde `test` geschreven wordt.

Als de voorwaarde `test` naar `True` evalueert, dan levert de expressie `expr_true` het resultaat op van de

voorwaardelijke expressie. Als de voorwaarde `test` naar `False` evalueert, dan levert de expressie `expr_false` het resultaat op van de voorwaardelijke expressie.

Het resultaat van een voorwaardelijke expressie kan bijvoorbeeld toegekend worden aan een variabele `resultaat`:

```
resultaat = <expr_true> if <test> else <expr_false>
```

Specifieke info

Bij deze oefening kan het nuttig zijn om eerst het aantal uren en minuten op een 24-uursklok om te zetten naar één enkele variabele `minuten_sinds_middernacht` die het aantal minuten aangeeft dat verstreken is sinds het begin van de dag (middernacht). Als het tijdstip bijvoorbeeld *15:20* is, dan wordt aan de variabele `minuten_sinds_middernacht` de waarde $15 \times 60 + 20 = 920$ toegekend. Dit maakt het een stuk eenvoudiger om de tijd te verhogen (of verlagen) met een vast aantal minuten m: tel eenvoudigweg m op bij de variabele `minuten_sinds_middernacht`.

Door gebruik te maken van de gehele deling en de modulo operator kan de variabele `minuten_sinds_middernacht` terug ontbonden worden in het aantal uren en minuten op een 24-uursklok.

```
>>> uren = 15
>>> minuten = 20
>>> minuten_sinds_middernacht = 60 * uren + minuten
>>> minuten_sinds_middernacht
920
>>> minuten_sinds_middernacht += 50          # verhoog aantal minuten met 50
>>> minuten_sinds_middernacht
970
>>> (minuten_sinds_middernacht // 60) % 24   # aantal uren op 24-uursklok
16
>>> minuten_sinds_middernacht % 60           # aantal minuten op 24-uursklok
10
```

Let op het feit dat we bij het afleiden van het aantal uren op een 24-uursklok een extra modulobwerking (`% 24`) toegevoegd hebben. Deze bewerking zorgt ervoor dat de afleiding nog steeds werkt als het aantal minuten sinds middernacht het totaal aantal minuten in één dag (24 uur) overschrijdt.

Pythagorese drietallen

Specifieke info

Een eerste aanzet voor deze opgave is het gebruik van de *brute force* techniek. Hierbij probeer je alle mogelijke waarden voor a, b en c uit en controleer je telkens of de waarden leiden tot een geldige oplossing.

Als je echter op een naïeve manier alle mogelijke oplossingen uitprobeert, dan zal je tegen de tijdslimiet aanlopen die we vooraf hebben ingesteld op Dodona. In dit geval betekent het niet noodzakelijk dat je ingediende oplossing in een oneindige lus is vastgeraakt, maar dat het te lang duurt om voor alle testgevallen de gevraagde oplossing te vinden.

Je zal dus nog enkele optimalisaties moeten doorvoeren aan de *brute force* om je oplossing efficiënter te maken, door na te denken welke gevallen je niet (meer) moet controleren omdat ze zeker niet tot een geldige oplossing zullen leiden. Vermijden om overbodige combinaties te evalueren wordt in technische termen *snoeien* genoemd. Hiervoor kan je alle informatie die je meekrijgt in de opgave goed gebruiken!