# 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 uiteindelijk `False` wordt. Bekijk bij wijze van bijvoorbeeld 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 initiële 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 Eclipse, dan kan je nagaan dat je programma aan het uitvoeren is, door het rode vierkantje bovenaan de Console. Als je op dit vierkantje klikt, dan forceer je om de uitvoering van het programma te stoppen.

### Ongebruikte variabelen

Als je in Eclipse gebruik maakt van pylint (Window > Preferences > PyDev > Pylint > Use pylint?) – een module die markeringen in de linkerkantlijn plaatst als je slechte programmeerstijl gebruikt – dan kan je wel eens een aanwijzing krijgen dat je een variabele gedefinieerd hebt die niet in je code gebruikt wordt. Op zich is dat niet erg, behalve als die variabele later toch ergens zou opduiken in je code en daar mogelijks een verkeerde waarde zou hebben.

Dit kan zich bijvoorbeeld voordoen als je een `for`-lus in combinatie met de `range` functie gebruikt om een reeks statements een vast aantal keer uit te voeren. Omdat de syntaxis van de `for`-lus altijd een lusvariable verwacht die onmiddellijk op het `for` keyword volgt, moet je dus wel een variabele gebruiken die je niet noodzakelijk nog nodig hebt in je code. In dit geval is het gebruikelijk om een underscore (`_`) als variabelenaam te gebruiken, wat in Python-middens expliciet aangeeft: *dit is een variabele die niet zal gebruikt worden in de code*. Pylint houdt hiermee rekening, en zal voor een underscore nooit aangeven dat die variabele niet gebruikt wordt. De code

```
>>> teller = 0
>>> for i in range(4):
...     teller = 2 * teller + 1
...
```

kan je dus beter herschrijven als

```
>>> teller = 0
>>> for _ in range(4):
...     teller = 2 * teller + 1
...
```

# Hittegolf

## Algemene info

### Lussen vroegtijdig afbreken

In Python kan je gebruik maken van de statements `break` en `continue` om een lus vroegtijdig af te breken. Deze statements worden echter algemeen aanzien als slechte programmeerstijl.

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 reeds 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.

## Specifieke info

Bij deze opgave is het een goed idee om bij te houden hoeveel opeenvolgende zomerse dagen je hebt zien passeren, en hoeveel tropische dagen er binnen die afgelopen periode van zomerse dagen werden waargenomen.

# De drie wijzen

## Algemene info

### Vermijden om te rekenen met floating point getallen

Omdat computers altijd afrondingsfouten kunnen maken als ze moeten rekenen met *floating point* getallen (zie onder andere de video **werken met floating point getallen** die ingelinkt staat bij de oefeningen van reeks 02), is het aangewezen om waar mogelijk niet met *floating point* getallen maar met gehele getallen te rekenen.

Als je bijvoorbeeld weet dat alle decimale getallen waarmee moet gewerkt worden maximaal twee cijfers na de komma hebben (bijvoorbeeld geldbedragen in euro of dollar), dan volstaat het om de getallen te vermenigvuldigen met honderd om ze om te zetten naar een gehele getal. Nadat je alle berekeningen hebt uitgevoerd met deze gehele getallen, kan je de resultaten terug omzetten naar decimale getallen.

```
>>> a_float = 5.33
>>> b_float = 2.5
>>> a_float + b_float
7.83
>>> a_int = round(100 * a_float)
>>> b_int = round(100 * b_float)
```

```
>>> som_int = a_int + b_int
>>> som_int
783
>>> som_int / 100
7.83

>>> a_float * b_float
13.325
>>> product_int = a_int * b_int
>>> product_int
133250
>>> product_int / 100          # verkeerde omzetten van resultaat naar float !!!
1332.5
>>> product_int / (100 ** 2)
13.325
```

Let in bovenstaand voorbeeld ook op het feit dat we gebruik maken van de ingebouwde functie `round` (en bijvoorbeeld niet van de ingebouwde functie `int`) om reële getallen om te zetten naar gehele getallen, net om de invloed van mogelijke afrondingsfouten te vermijden. Dit komt doordat computers decimale getallen niet perfect kunnen voorstellen. Deze afrondingsfouten kunnen bijvoorbeeld zwaar gaan doorwegen als je *floating point* getallen gaat vermenigvuldigen.

```
>>> getal = 69.96
>>> 100 * getal
6995.999999999999
>>> int(100 * getal)
6995
>>> round(100 * getal)
6996
```

## 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 echter 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 !