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.

En nu eens helemaal ondersteboven

Algemene info

Escapen van backslash

In Python gebruik je een backslash om het volgende karakter in de string te "escapen", waardoor dit karakter zijn speciale betekenis verliest (bijvoorbeeld om een dubbel aanhalingsteken te plaatsen in een string die zelf zit ingesloten tussen dubbele aanhalingstekens) of net een speciale betekenis krijgt (zoals een regeleinde dat word voorgesteld door `'\n'` of een tab die wordt voorgesteld door `'\t'`).

Omdat hierdoor ook de backslash zelf een speciale betekenis krijgt binnen een string (namelijk het escapen van karakters), moet je een letterlijke backslash binnen een string noteren als een dubbele backslash (`'\\'`), waarbij de eerste backslash de rol opneemt van het escape-symbool en de tweede backslash het karakter is dat zijn letterlijke betekenis moet krijgen door het escapen.

Voorstelling van regeleindes

In Python kan je een regeleinde voorstellen als de string `'\n'`. Als je dus een string wilt opbouwen die uit meerdere regels bestaat, dan doe je dit zo

```
>>> tekst = 'Ziehier de eerste regel'
>>> tekst += '\n'
>>> tekst += 'En de tweede regel'
>>> tekst
'Ziehier de eerste regel\nEn de tweede regel'
>>> print(tekst)
Ziehier de eerste regel
En de tweede
```

Inlezen van meerdere regels

Als je op voorhand weet hoeveel regels er moeten ingelezen worden, dan kun je gebruik maken van de volgende constructie

```
>>> 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 kun je gebruik maken van de volgende constructie

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

Opmerkingen

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
...
```

De stringmethode `index`

De stringmethode `index` kan gebruikt worden om de meest linkse positie van een karakter in een string te bepalen.

```
>>> string = 'mississippi'
>>> teken = 'i'
>>> string.index(teken)
1
```

Dit codefragment vindt dus het eerste voorkomen van de letter `i` op positie 1 in de string `mississippi`. Let hierbij op het feit dat Python de posities van de karakters in een string indexeert vanaf 0, en niet vanaf 1.

Het valt wel op te merken dat deze methode niet altijd zeer efficiënt werkt. Om de positie te vinden van een karakter dat voor het eerst opduikt op plaats p, moet de functie de eerste $p - 1$ posities van de string doorlopen. Voor een karakter dat niet voorkomt in de string moet de stringmethode `index` zelfs de volledige string aflopen vooraleer er kan vastgesteld worden dat het karakter er niet gezien werd.

Het raadsel van Nob

Algemene info

Verwijderen van witruimte vooraan en/of achteraan

In Python kan je gebruik maken van de stringmethode `strip` om witruimte (spaties, tabs, regeleindes) vooraan en achteraan te verwijderen. Indien je enkel de witruimte vooraan wilt verwijderen, dan kan je gebruik maken van de stringmethode `lstrip`. Indien je enkel de witruimte achteraan wilt verwijderen, dan kan je gebruik maken van de stringmethode `rstrip`. Je kan aan deze stringmethode ook een argument meegeven, waarmee je aangeeft welke karakters er vooraan en/of achteraan moeten verwijderd worden. Voor de details hierover verwijzen we naar The Python Standard Library.

```
>>> tekst = '  Dit is een tekst   '
>>> tekst.strip()
'Dit is een tekst'
>>> tekst.lstrip()
'Dit is een tekst   '
>>> tekst.rstrip()
'  Dit is een tekst'
```

Uitlijnen van strings over een vast aantal posities

Je kunt in de *format specifier* van de stringmethod `format` aangeven dat voor een stuk tekst een vast aantal posities moeten gereserveerd worden. Dit doe je de tekst uit te schrijven als een string (aangegeven door de letter `s`) en daarvoor met een natuurlijk getal aan te geven hoeveel posities er voor die string moeten gereserveerd worden.

```
>>> '{:5s}'.format('abc')   # 5 posities voorbehouden
'abc  '
```

Als de string langer is dan het aantal voorbehouden plaatsen, dan wordt de volledige string uitgeschreven en wordt er dus meer plaats ingenomen dan voorbehouden. Als de string echter korter is dan het aantal voorbehouden plaatsen, dan wordt de string standaard links uitgelijnd. Je kunt in de *format specifier* echter ook expliciet de uitlijning aangeven: links, rechts of gecentreerd. Python zal dan automatisch het juiste aantal spaties vooraan en/of achteraan toevoegen.

```
>>> '{:<5s}'.format('abc')   # 5 posities voorbehouden, links uitlijnen
'abc  '
>>> '{:>5s}'.format('abc')   # 5 posities voorbehouden, rechts uitlijnen
'  abc'
>>> '{:^5s}'.format('abc')   # 5 posities voorbehouden, centreren
' abc '
```

Indien er moet gecentreerd worden, en het aantal toe te voegen spaties is oneven, dan kiest Python ervoor om rechts één spatie meer toe te voegen dan links.

Opmerkingen

Herhaling van een string

Om een bepaalde string een vast aantal keer te herhalen, kan je de string vermenigvuldigen met een natuurlijk getal. Net zoals bij de vermenigvuldiging van getallen gebruik je hiervoor de operator `*`. De volgorde van de string en het natuurlijk getal in de vermenigvuldiging speelt geen rol

```
>>> 'a' * 3
'aaa'
>>> 5 * 'ab'
'ababababab'
```

Dit is vooral handig als het aantal herhalingen van een string niet op voorhand vastligt, maar bijvoorbeeld zit opgeslagen in een variabele.

```
>>> herhalingen = 4
>>> herhalingen * 'abc'
'abcabcabcabc'
```

Bijbelcodes

Specifieke info

Om het verborgen woord uit een gegeven bijbelfragment te lezen, raden we je aan om de volgende strategie te gebruiken. Als eerst stap lees je alle regels van
het volledige tekstfragment in, en zet je dit om naar één enkele string waarin enkel de letters overgehouden worden. Alle andere karakters moet je immers negeren. Als bijvoorbeeld het volgende bijbelfragment krijg, verspreid over twee regels

```
And hast not suffered me to kiss my sons and my daughters?
thou hast now done foolishly in so doing.
```

Dan hou je op basis van de eerste stap enkel de string van opeenvolgende letters uit dit bijbelfragment over.

```
Andhastnotsufferedmetokissmysonsandmydaughtersthouhastnowdonefoolishlyinsodoing
```

Daarna kan je uit deze string het verborgen woord lezen, door te starten vanaf de opgegeven letter, en daarna de volgende letters uit te lezen door telkens het aantal gegeven letters vooruit of achteruit te springen. Als je in dit geval start vanaf de 45ste letter, en telkens vier letters vooruit leest tot je een woord van 7 letters hebt uitgelezen, dan bekom je het woord *roswell*.

Algemeen

Functies/methoden: return vs print

Bij opgaven waarbij er gevraagd wordt om functies om methoden te implementeren, moet je heel goed opletten of er gevraagd wordt dat de functie/methode een resultaat moet **teruggeven**, dan wel dat de functie/methode een resultaat moet **uitschrijven**. Om een functie/methode een resultaat te laten teruggeven maak je gebruik van van een `return` statement. Om een functie een functie/methode een resultaat te laten uitschrijven maak je gebruik van de ingebouwde functie `print`.

In de opgave C-som (reeks 05) wordt er bijvoorbeeld gevraagd om een functie `csom` te schrijven die een resultaat moet **teruggeven**. Een mogelijke correcte implementatie van deze functie is

```
def csom(getal):
    return getal % 9
```

waarbij een `return` statement gebruikt wordt om een berekende waarde te laten teruggeven door de functie. Stel dat we in plaats van een waarde terug te geven, verkeerdelijk gebruik zouden maken van een `print` statement om de berekende waarde uit te schrijven, en dat we de volgende oplossing zouden indien voor deze opgave.

```
def csom(getal):
    print(getal % 9)
```

Deze oplossing zal een **verkeerd antwoord** opleveren, waarbij we de volgende informatie te zien krijgen op de feedbackpagina.

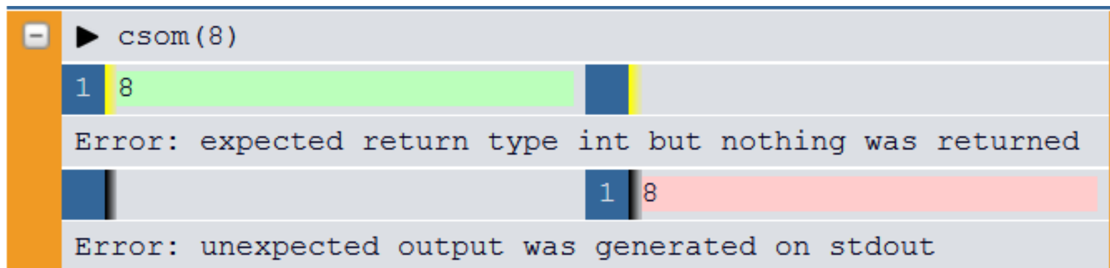

Figure 1: return vs print

Hierbij worden twee opmerkingen geformuleerd. De eerste opmerking geeft aan dat er verwacht werd dat de functie een integer waarde zou teruggeven (de waarde 8), maar dat de functie niet heeft teruggegeven (of meer specifiek, dat de functie de waarde `None` heeft teruggegeven). Dit is de betekenis van de foutmelding

```
Error: expected return type int but nothing was returned
```

Bovendien komt er nog een tweede opmerking die aangeeft dat de functie iets heeft uitgeschreven (naar *standard output*, of kortweg *stdout*) terwijl dat helemaal niet verwacht werd. Dit is de betekenis van de foutmelding

```
Error: unexpected output was generated on stdout
```

Als je een `return` statement had gebruikt in plaats van de `print` functie, dan waren beide foutmeldingen verdwenen en kreeg je een **correct antwoord**. Merk op dat resultaten die door een functie/methode worden teruggegeven in de feedbacktabel worden gemarkeerd met een gele band, en dat resultaten die door een functie/methode worden uitgeschreven worden gemarkeerd met een zwarte band.

Vigenèrecodering

Algemene info

Itereren over posities en waarden van sequentietypes

De ingebouwde functie `enumerate` kan gebruikt worden om een een iterator op te vragen van een sequentietype (*sequence type*: strings, lijsten, tuples, bestanden, ...), die zowel de positie als de waarde van het volgende element van het type teruggeeft. Onderstaand voorbeeld geeft bijvoorbeeld aan hoe je tegelijkertijd de posities en de karakters op die posities kunt overlopen voor een string.

```
>>> for index, karakter in enumerate('abc'):
...     print("index: {}".format(index))
...     print("karakter: {}".format(karakter))
...
index: 0
karakter: a
index: 1
karakter: b
index: 2
karakter: c
```

Je kan dit bijvoorbeeld gebruiken als je synchroon de karakters van twee strings wil overlopen.

```
>>> eerste = 'abc'
>>> tweede = 'def'
>>> for index, karakter in enumerate(eerste):
...    print("{}-{}".format(karakter, tweede[index]))
...
a-d
b-e
c-f
```

In dat geval is het echter aangewezen om gebruik te maken van de ingebouwde functie `zip`, die net haar bestaansrecht haal uit het feit dat ze een iterator teruggeeft voor twee of meer iterateerbare objecten.

```
>>> eerste = 'abc'
>>> tweede = 'def'
>>> for karakter1, karakter2 in zip(eerste, tweede):
...    print("{}-{}".format(karakter1, karakter2))
...
a-d
b-e
c-f
```

Humble-Nishiyama gokspel

Algemene info

Opmerkingen

Controle of bepaalde voorwaarden gelden

Soms moet je in je programmacode expliciet nagaan of er aan bepaalde voorwaarden voldaan is, en moet je programma reageren als één van de voorwaarden geschonden is. Eén van de manieren waarop je dit kunt doen in Python is door gebruik te maken van het `assert` statement.

```
>>> x = 2
>>> y = 2
>>> assert x == y, 'beide getallen zijn niet gelijk'
>>> x = 1
>>> assert x == y, 'beide getallen zijn niet gelijk'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: beide getallen zijn niet gelijk
```

De algemene syntaxis van een `assert` statement is

```
assert <voorwaarde>, <boodschap>
```

Het `assert` statement controleert of er aan de voorwaarde voldaan is. Indien dat niet het geval is, dan zal er een `AssertionError` opgeworpen worden met de boodschap die wordt meegegeven op het einde van het `assert` statement. Indien deze uitzondering niet wordt opgevangen (wat voor deze cursus altijd het geval zal zijn), dan wordt op het punt van het opwerpen van de `AssertionError` de uitvoer van de code ook beëindigd (*runtime error*).

Rövarspråket

Specifieke info

De oplossing van deze opgave bestaat erin dat je inziet dat je de verwerking van een groep medeklinkers moet uitstellen tot op het ogenblik dat je een niet-medeklinker tegenkomt in de string (waardoor je weet dat de groep medeklinkers compleet is). Dit wordt geïllustreerd aan de hand van het onderstaand voorbeeld, waarbij de string `abcdefg` wordt omgezet naar Rövarspråket.

Hierbij wordt de string die moet gecodeerd worden, karakter per karakter overlopen. Als hierbij een medeklinker wordt tegengekomen, dan wordt die niet onmiddellijk toegevoegd aan de gecodeerde string, maar wordt die toegevoegd aan de groep opeenvolgende medeklinkers. Op het moment dat er een niet-medeklinker wordt tegengekomen, dan sluit deze de groep van opeenvolgende klinkers af. Indien er zich een groep gevormd had, dan wordt die eerst aan de gecodeerde string toegevoegd, gevolgd door de letter `o` en nog een herhaling van de groep opeenvolgende medeklinkers. Pas daarna wordt de niet-medeklinker aan de gecodeerde groep toegevoegd, en wordt er een nieuwe groep medeklinkers gevormd.

Nadat alle karakters van de te coderen string zijn overlopen, kan het zijn dat er zich nog een groep van opeenvolgende medeklinkers gevormd heeft die niet aan de gecodeerde string werd toegevoegd. Dit is het geval wanneer de string eindigt op een medeklinker. In het voorbeeld is dit het geval voor de groep `fg`. Indien dit geval is, dan moet deze groep ook nog toegevoegd worden aan de gecodeerde string, gevolgd door de letter `o` en nog een herhaling van de groep opeenvolgende medeklinkers.

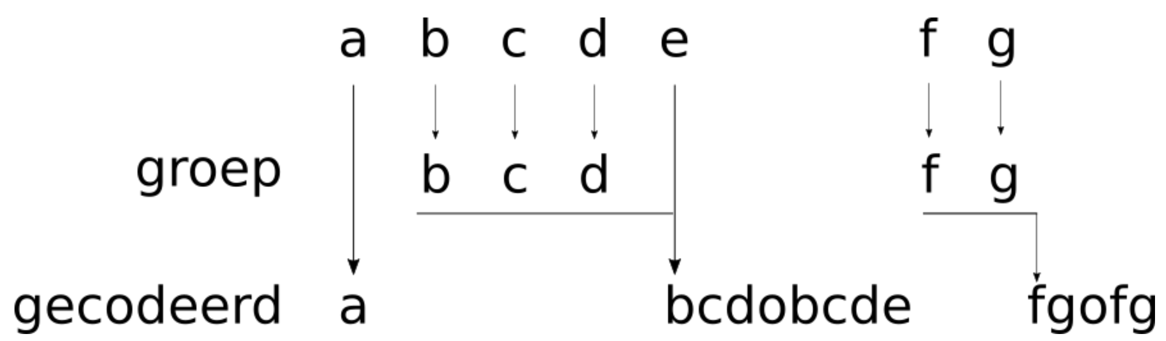

Figure 2: Rövarspråket