Hoofdstuk 6

Condities

In een programma zijn er vaak regels code die je alleen wilt uitvoeren onder bepaalde omstandigheden. Om dat te regelen, bieden alle programmeertalen zogeheten "conditionele statements" of "condities." In dit hoofdstuk leg ik uit hoe condities werken in Python

6.1 Boolean expressies

Een conditioneel statement, vaak een "if"-statement genoemd, bestaat uit een test en één of meerdere acties. De test is een zogeheten "boolean expressie." De acties worden alleen uitgevoerd als de test evalueert als zijnde "waar." Bijvoorbeeld, een app op een smartphone kan een waarschuwing geven als de batterij minder dan 5% vol is. Dat betekent dat de app test of een zekere variabele `batterij_energie` kleiner is dan 5, dus of de vergelijking `batterij_energie < 5` als zijnde "waar" geëvalueerd wordt. Als de variabele momenteel de waarde 17 bevat, evalueert de test `batterij_energie < 5` als zijnde "onwaar."

6.1.1 Booleans

In Python wordt "waar" weergegeven door de waarde **`True`**, en "onwaar" door de waarde **`False`**.

`True` en **`False`** zijn zogeheten "boolean waardes," die door Python zijn gedefinieerd. **`True`** en **`False`** zijn zelfs de *enige* booleans, en alles wat niet **`False`**, is automatisch **`True`**.

Als je je afvraagt welke data type **`True`** en **`False`** hebben: ze zijn van het type **`bool`**. In Python kan echter elke waarde worden geïnterpreteerd als boolean, ongeacht het data type. Dus als je een test doet of iets **`True`** of **`False`** is, en je test dat van een waarde die niet van het data type **`bool`** is, dan wordt hetgeen je test toch als ofwel **`True`** ofwel **`False`** beschouwd.

De volgende waardes worden beschouwd als zijnde **`False`**:

- De speciale waarde **`False`**
- De speciale waarde **`None`** (die ik heb besproken in hoofdstuk 5)
- Iedere numerieke waarde die nul is, bijvoorbeeld 0 en 0.0

- Iedere lege serie, bijvoorbeeld een lege string ("")
- Iedere lege "afbeelding," bijvoorbeeld een lege "dictionary" (dictionaries zijn het onderwerp van hoofdstuk 13)
- Iedere functie of methode die één van de bovenstaande waardes retourneert (inclusief functies die niks retourneren)

Iedere andere waarde wordt beschouwd als zijnde **`True`**.

Een expressie die evalueert als **`True`** of **`False`** heet een "boolean expressie."

6.1.2 Vergelijkingen

De meestgebruikte boolean expressies zijn vergelijkingen. Een vergelijking bestaat uit twee waardes met een vergelijkingsoperator ertussen. Vergelijkingsoperatoren zijn:

```
<    kleiner dan
<=   kleiner dan of gelijk aan
==   gelijk aan
>=   groter dan of gelijk aan
>    groter dan
!=   niet gelijk aan
```

Een veelgemaakte fout is om twee waardes te vergelijken met een enkele =. De enkele = is de assignment operator. Meestal (maar niet altijd) produceert Python een syntax of runtime error als je de = probeert te gebruiken om twee waardes te vergelijken.

Je kunt vergelijksoperatoren gebruiken zowel tussen getallen als tussen strings. Vergelijkingen tussen strings zijn alfabetische vergelijkingen, waarbij je wel moet bedenken dat hoofdletters altijd beschouwd worden als kleiner dan kleine letters (en cijfers kleiner dan alle letters). Ik ga daar dieper op in in hoofdstuk 10.

Hier volgen een paar voorbeelden van vergelijkingen:

listing0601.py

```
print( "1.", 2 < 5 )
print( "2.", 2 <= 5 )
print( "3.", 3 > 3 )
print( "4.", 3 >= 3 )
print( "5.", 3 == 3.0 )
print( "6.", 3 == "3" )
print( "7.", "syntax" == "syntax" )
print( "8.", "syntax" == "semantiek" )
print( "9.", "syntax" == " syntax" )
print( "10.", "Python" != "rotzooi" )
print( "11.", "Python" > "Perl" )
print( "12.", "banaan" < "mango" )
print( "13.", "banaan" < "Mango" )
```

Zorg dat je deze vergelijkingen uitvoert, en dat je snapt waarom ze de uitkomst geven die ze geven!

Opgave Begrijp je waarom `3 < 13` **`True`** oplevert, maar `"3" < "13"` **`False`** oplevert? Indien niet, denk er dan goed over na!

Je kunt de uitkomst van een boolean expressie aan een variabele toekennen als je wilt:

```
groter = 5 > 2
print( groter )
groter = 5 < 2
print( groter )
print( type( groter ) )
```

Opgave Schrijf code die test of $1/2$ groter dan, gelijk aan, of kleiner is dan 0.5. Doe dat ook voor $1/3$ en 0.33. Doe het dan ook voor $(1/3) * 3$ en 1.

Vergelijkingen tussen data types die niet vergeleken kunnen worden, leiden meestal tot runtime errors.

```
# Deze code geeft een runtime error.
print( 3 < "3" )
```

6.1.3 `in` operator

Python heeft een speciale operator die de "lidmaatschap test operator" heet, en die vanwege die onverkwikkelijke mondvol meestal de "in operator" wordt genoemd aangezien hij gecodeerd wordt als **`in`**. De **`in`** operator test of een waarde voorkomt in een collectie, als de waarde links van de **`in`** staat, en de collectie rechts van de **`in`**.

Er zijn verschillende soorten collecties in Python, maar de enige die ik tot op dit moment bediscussiëerd heb is de string. Een string is een collectie van tekens. Je kunt testen of een specifiek teken, of een groepje tekens, onderdeel is van een string middels de **`in`** operator. De tegenhanger van de **`in`** operator is de **`not in`** operator, die **`True`** oplevert als **`in`** **`False`** oplevert, en vice versa. Bijvoorbeeld:

```
print( "y" in "Python" )
print( "x" in "Python" )
print( "p" in "Python" )
print( "th" in "Python" )
print( "to" in "Python" )
print( "y" not in "Python" )
```

Zorg er weer voor dat je deze evaluaties begrijpt!

Opgave Schrijf code die test van ieder van de klinkers (`"a"`, `"e"`, `"i"`, `"o"`, `"u"`) of ze voorkomen in je naam. Je mag hoofdletters negeren.

6.1.4 Logische operatoren

Boolean expressies kunnen gecombineerd worden middels logische operatoren. Er zijn drie logische operatoren: **`and`**, **`or`**, en **`not`**.

and en **or** plaats je tussen twee boolean expressies.

Als **and** tussen twee boolean expressies staat, is het resultaat **True** als beide expressies **True** zijn; anders is het resultaat **False**.

Als **or** tussen twee boolean expressies staat, is het resultaat **True** als één of beide expressies **True** zijn; het resultaat is alleen **False** als beide **False** zijn.

not kun je voor een boolean expressie plaatsen om hem om te keren van **True** naar **False** en vice versa.

Bijvoorbeeld:

```
t = True
f = False
print( t and t )
print( t and f )
print( f and t )
print( f and f )
print( t or t )
print( t or f )
print( f or t )
print( f or f )
print( not t )
print( not f )
```

Kijk uit met het gebruik van logische operatoren, want een combinatie van **and**s en **or**s kan leiden tot onverwachte resultaten. Gebruik haakjes om te zorgen dat ze in de gewenste volgorde geëvalueerd worden. Bijvoorbeeld, in plaats van `a and b or c` te schrijven, moet je `(a and b) or c` of `a and (b or c)` schrijven (afhankelijk van de gewenste volgorde), zodat het duidelijk is welke evaluatie je wilt uitvoeren. Zelfs als je weet in welke volgorde Python de evaluatie doet zonder haakjes, hoeft dat niet te gelden voor iemand anders die je code leest.

Opgave Geef voor de code hieronder waardes voor `a`, `b`, and `c`, die ertoe leiden dat de twee expressies verschillende uitkomsten hebben.

listing0602.py

```
a = # True of False?
b = # True of False?
c = # True of False?

print( (a and b) or c )
print( a and (b or c) )
```

Als je logische expressie maakt met alleen **and**s, of alleen **or**s, hoef je geen haakjes te gebruiken, want dan is er slechts één mogelijke evaluatie van de expressie.

Boolean expressies worden van links naar rechts geëvalueerd, en Python stopt de evaluatie op het moment dat de uitkomst van de evaluatie bekend is. Neem bijvoorbeeld de volgende code:

```
x = 1
y = 0
print( (x == 0) or (y == 0) or (x / y == 1) )
```

Als je deelt door nul, geeft Python een runtime error, dus de evaluatie van `x / y == 1` geeft een error als `y` nul is. En als je de code bestudeert, zie je dat `y` inderdaad nul is. Maar de code geeft geen foutmelding. Python evalueert de boolean expressie van links naar rechts, en ziet op een gegeven moment `... or (y == 0) or ...` `y == 0` evalueert als **`True`**. Omdat een expressie die via **`or`**s gecombineerd is **`True`** is als één van de componenten **`True`** is, kan Python na evaluatie van `(y == 0)` concluderen dat deze hele expressie **`True`** is. Het is dus niet nodig dat Python `x / y == 1` evalueert, en Python doet dat dan ook niet. Het is wel van belang dat `y == 0` links van `x / y == 1` staat, zodat Python `y == 0` eerst test.

Merk op dat hoewel je heel ingewikkelde boolean expressies kunt bouwen via logische operatoren, ik je aanraad dat je je expressies zo eenvoudig mogelijk houdt. Eenvoudige expressies houden code leesbaar.

6.2 Conditionele statements

Zoals ik aan het begin van dit hoofdstuk aangaf, bestaan conditionele statements, die ook wel "condities" of "**`if`** statements" worden genoemd (omdat ze gedefinieerd worden met behulp van het gereserveerde woord **`if`**), uit een test en één of meer acties, waarbij de acties alleen worden uitgevoerd als de test **`True`** oplevert.

Hier is een voorbeeld:

```
x = 5
if x == 5:
    print( "x is 5" )
```

De syntax van een **`if`** statement is als volgt:

```
if <boolean expressie>:
    <acties>
```

Let op de dubbele punt (:) die achter de boolean expressie staat, en het feit dat `<acties>` inspringt.

6.2.1 Blokken code

In de syntactische beschrijving van de **`if`** statement, zie je dat `<acties>` "inspringt," dus één tabulatie naar rechts is geplaatst (in het Engels heet dit "indent"). Dit is opzettelijk zo gedaan en noodzakelijk. Python beschouwt statements die elkaar opvolgen en die hetzelfde niveau van inspringing hebben als één blok code. Het blok code dat onder het **`if`** statement staat is de lijst van acties die worden uitgevoerd als de boolean expressie **`True`** is. Bijvoorbeeld:

listing0603.py

```
x = 7
if x < 10:
    print( "Deze regel wordt alleen uitgevoerd als x < 10." )
    print( "En dat geldt ook voor deze regel." )
print( "Deze regel wordt echter altijd uitgevoerd." )
```

Opgave Wijzig de waarde van `x` en test hoe dat de resultaten beïnvloedt.

Dus alle regels code die onder de **`if`** staan en inspringen, horen tot het blok code dat wordt uitgevoerd als de boolean expressie behorende bij de **`if`** evalueert als **`True`**. Het blok code wordt daarentegen overgeslagen als de boolean expressie evalueert als **`False`**. Statements die volgen na de **`if`** en die niet inspringen (althans, niet zo diep als het blok code onder de **`if`**) worden uitgevoerd ongeacht het resultaat van de evaluatie van de boolean expressie.

Je hoeft je niet te beperken tot slechts één **`if`** statement. Je kunt er zoveel hebben als je wilt.

listing0604.py

```
x = 5
if x == 5:
    print( "x is 5" )
if x > 4:
    print( "x is groter dan 4" )
if  x >= 5:
    print( "x is groter dan of gelijk aan 5" )
if x < 6:
    print( "x is kleiner dan 6" )
if x <= 5:
    print( "x is kleiner dan of gelijk aan 5" )
if x != 6 :
    print( "x is niet 6" )
```

Opgave Wijzig weer de waarde van `x` en test hoe dat de resultaten beïnvloedt.

6.2.2 Inspringen

In Python is correct inspringen van het grootste belang! Zonder correcte inspringing kan Python niet zien welke regels code een blok vormen, en kan daarom niet je code uitvoeren zoals je bedoelt.[5]

[5] In veel programmeertalen (of eigenlijk in vrijwel alle programmeertalen) worden blokken code door de compiler/interpreter herkend doordat ze beginnen en eindigen met een speciaal symbool of gereserveerd woord. Bijvoorbeeld, in talen als Java en C++ worden blokken code omsloten door accolades, terwijl in talen als Pascal en Modula ze beginnen met het woord `begin` en eindigen met het woord `end`. Dat betekent dat in vrijwel alle talen correcte inspringing niet nodig is. Je ziet toch dat goede programmeurs correct inspringen, ongeacht de taal die ze gebruiken. Dat maakt het namelijk gemakkelijk te zien welke delen van de code bij elkaar horen, bijvoorbeeld, wat er hoort bij een `if` statement. Bij Python is het inspringen een verplichting. Voor ervaren programmeurs die voor het eerst Python leren komt dat wat vreemd over, maar ze beseffen al snel dat het ze niks uitmaakt – ze lieten hun code toch al netjes inspringen. Python maakt het echter ook voor beginnende programmeurs een noodzakelijkheid om netjes in te springen, wat betekent dat ze gedwongen zijn om nette code te schrijven. En dat is alleen maar nuttig voor iedereen.

Je kunt inspringen door middel van de Tab toets, of je kunt het doen middels spaties. De meeste editors doen aan "auto-indenting," dat wil zeggen, ze springen automatisch in als de code daartoe aanleiding geeft. Bijvoorbeeld, als je een editor hebt die Python ondersteund, en je schrijft een **if** statement, dan zal de editor de daarop volgende regel meteen laten inspringen (als dat niet gebeurt, heb je waarschijnlijk een syntax fout gemaakt, bijvoorbeeld de dubbele punt vergeten). Ook wordt het niveau van inspringing aangehouden voor iedere volgende regel, tenzij je middels de Backspace toets inspringing verwijdert.

In Python programma's is inspringing normaal vier spaties, dus een druk op de Tab toets moet vier posities inspringen. Zolang je in een specifieke editor werkt, kun je ofwel de Tab toets gebruiken, ofwel zelf de spatiebalk vier keer indrukken, om één niveau van inspringing op te schuiven. Je kunt echter in de problemen komen als je tussentijds wisselt van editor, die wellicht andere instellingen voor tabulaties gebruikt. Zelfs als je code in die andere editor er goed uitziet, kan Python toch tijdens uitvoering melden dat er "indentation conflicts" zijn (dat wil zeggen, dat tabulaties en spaties door elkaar gehutseld zijn). Dat kan gezien worden als een syntax fout. De meeste editors bieden de mogelijkheid via een optie om tabulaties automatisch te vervangen door spaties, wat vermijdt dat zulke problemen optreden. Dus als je een tekst editor gebruikt om Python code te schrijven, controleer dan of die optie bestaat, en laat hem dan automatisch tabulaties vervangen door vier spaties.

Opgave De volgende code bevat inspring-fouten. Verbeter de code.

listing0605.py

```
# Deze code bevat tabulatie-fouten!
x = 3
y = 4
if x == 3 and y == 4:
    print( "x is 3" )
   print( "y is 4" )
if x > 2 and y < 5:
print( "x > 2" )
print( "y < 5" )
if x < 4 and y > 3:
    print( "x < 4" )
        print( "y > 3" )
```

6.2.3 Twee-weg beslissingen

Het komt regelmatig voor dat een beslissing twee kanten uit kan gaan, dat wil zeggen, onder bepaalde omstandigheden wil je een bepaald iets doen, en als die omstandigheden niet optreden wil je iets anders doen. Python staat dit toe door aan een **if** statement een **else** tak toe te voegen.

```
x = 4
if x > 2:
    print( "x is groter dan 2" )
else:
    print( "x is kleiner dan of gelijk aan 2" )
```

De syntax is:

```
if <boolean expressie>:
    <acties>
else:
    <acties>
```

Merk op dat er een dubbele punt achter de **else** staat, net zoals achter de `<boolean expressie>` bij de **if**.

Het is van belang dat het woord **else** uitgelijnd is met het woord **if** waar het bij hoort. Als je ze niet correct uitlijnt, krijg je ofwel een tabulatie fout, ofwel je code doet niet wat je zou willen dat hij doet.

Een consequentie van het toevoegen van een **else** tak aan een **if** statement is dat altijd precies één van de twee blokken `<acties>` wordt uitgevoerd. Als de boolean expressie **True** is, wordt het blok code onder de **if** uitgevoerd, en wordt het blok code onder de **else** overgeslagen. Als de boolean expressie **False** is, wordt het blok code onder de **if** overgeslagen, maar wordt het blok code onder de **else** uitgevoerd.

Opgave Je kunt testen of een integer even of oneven is met de modulo operator. Namelijk als `x%2` nul is, dan is `x` even, en anders is `x` oneven. Schrijf code die vraagt om een integer en dan rapporteert of de integer even of oneven is (je kunt de `getInteger()` functie van `pcinput` gebruiken om om een integer te vragen).

Opmerking met betrekking tot inspringing: het is niet absoluut noodzakelijk om het blok code onder de **else** aan te lijnen met het blok code onder de **if**, zolang de inspringing maar consistent is binnen het blok code. Ervaren programmeurs gebruiken echter consistente inspringing door de hele code, wat het gemakkelijk maakt om te zien hoe een **if**-**else** statement werkt. Bijvoorbeeld, in de code hieronder is de inspringing voor de **else** tak kleiner dan voor de **if** tak. Syntactisch is dat correct, maar het maakt de code slechter leesbaar.

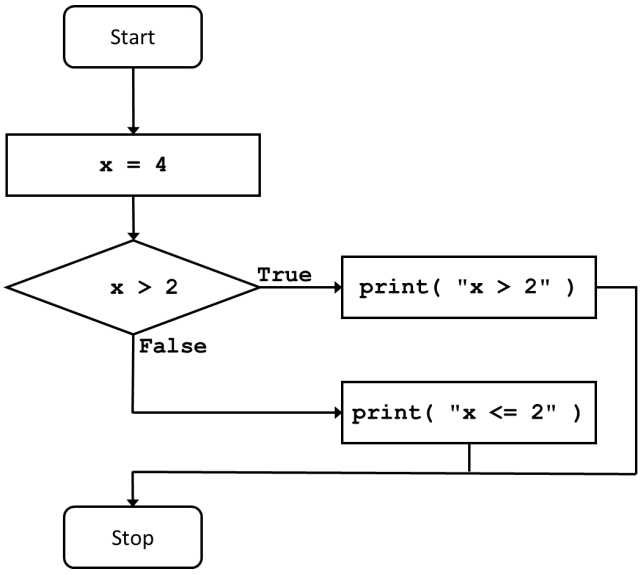
```
# Syntactisch correct maar lelijke inspringing.
x = 1
if x > 2:
    print( "x is groter dan 2" )
else:
  print( "x is kleiner dan of gelijk aan 2" )
```

6.2.4 Stroomdiagrammen

In de tijd dat het beroep "programmeur" nog vrij nieuw was, gebruikten programmeurs vaak een techniek die "stroomdiagram" genoemd wordt om algoritmes te beschrijven. Vandaag de dag worden stroomdiagrammen nog slechts zelden gebruikt. Studenten hebben mij echter aangegeven dat stroomdiagrammen helpen om begrip te krijgen van de exacte werking van conditionele expressies (en van iteraties, die in het volgende hoofdstuk besproken worden).

Een stroomdiagram is een schematische weergave van een algoritme middels blokken met pijlen ertussen. Er zijn drie soorten blokken (althans, ik heb voldoende aan drie soorten blokken voor dit boek). Rechthoekige blokken bevatten statements die uitgevoerd worden. Ruitvormige blokken bevatten een conditie die geëvalueerd wordt als **`True`** of **`False`**. Rechthoekige blokken met ronde hoeken geven ofwel de start (met de tekst "Start") ofwel het einde (met de tekst "Stop") van het algoritme aan.

Om een stroomdiagram te interpreteren, begin je bij het "Start" blok, en volgt de pijlen, waarbij je ieder statement dat je tegenkomt uitvoert. Als je een ruitvormig blok tegenkomt, evalueer je de conditie die erin staat, en dan volg je ofwel de pijl die met **`True`** gemarkeerd is als de conditie **`True`** is, ofwel de pijl die met **`False`** gemarkeerd is als de conditie **`False`** is. Wanneer je het "Stop" blok tegenkomt, ben je klaar.



Afb. 6.1: Stroomdiagram dat een twee-weg beslissing weergeeft.

Bijvoorbeeld, de code die hierboven (in 6.2.3) staat, waarbij een getal vergeleken wordt met 2 en waarbij wordt afgedrukt of het getal groter is dan 2, of kleiner dan of gelijk aan 2 is, is equivalent met het stroomdiagram dat getoond wordt in afbeelding 6.1.

6.2.5 Meer-weg beslissingen

Het komt voor dat je een situatie hebt waarbij je één van meerdere blokken code wilt uitvoeren, maar niet meer dan één. Dit soort meer-weg beslissingen kun je implementeren met een extra toevoeging aan een **if** statement, namelijk in de vorm van één of meer **elif** takken (**elif** staat voor "else if").

listing0606.py

```
leeftijd = 21
if leeftijd < 12:
    print( "Je bent een kind!" )
elif leeftijd < 18:
    print( "Je bent een teenager!" )
elif leeftijd < 30:
    print( "Je bent nog jong!" )
elif leeftijd < 50:
    print( "Beginnen grijze haren te komen?" )
else:
    print( "Wegen de jaren zwaar?" )
```

Deze code is equivalent aan het algoritme dat is weergeven in afbeelding 6.2.

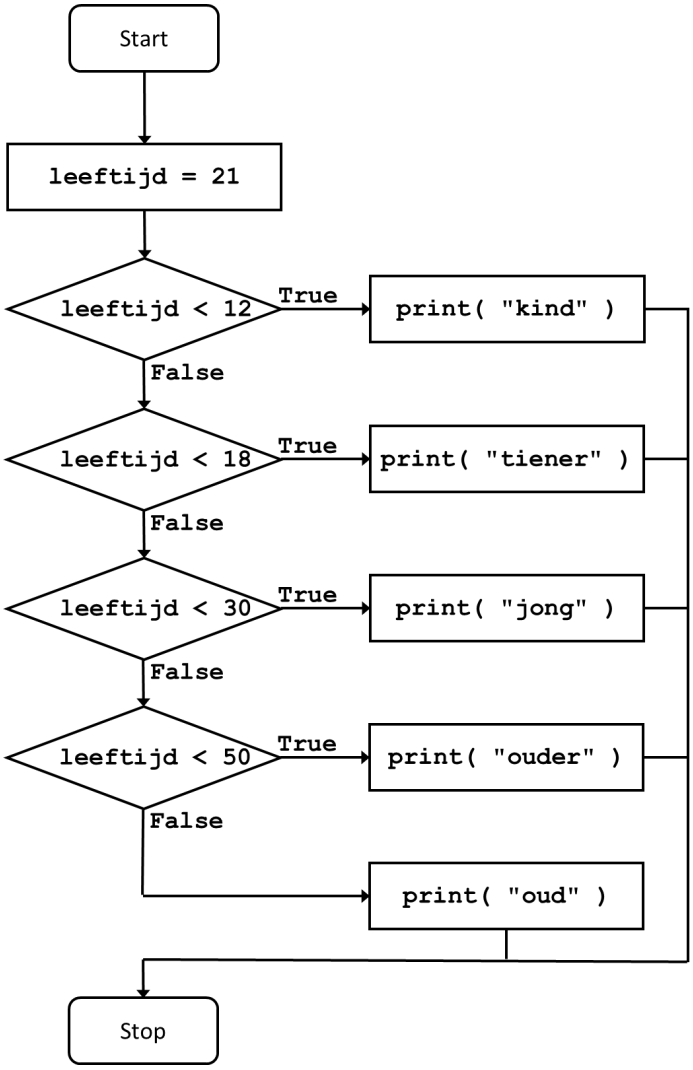
Opgave Verander in de code hierboven de waarde van de variabele `leeftijd` en bestudeer de resultaten.

De syntax is:

```
if <boolean expressie>:
    <acties>
elif <boolean expressie>:
    <acties>
else:
    <acties>
```

De syntax hierboven toont slechts één **elif**, maar je mag er meerdere hebben. De verschillende tests in een **if-elif-else** constructie worden in volgorde uitgevoerd. De eerste boolean expressie die geëvalueerd wordt als **True** laat het bijbehorende blok code uitvoeren. Geen andere blokken code in de hele constructie worden daarna nog uitgevoerd, en er worden daarna ook geen boolean expressies in de constructie meer getest.

Kortom, eerst wordt de boolean expressie bij de **if** geëvalueerd. Als die **True** is, wordt het blok code onder de **if** uitgevoerd. Anders wordt de boolean expressie bij de eerste **elif** geëvalueerd. Als die **True** blijkt te zijn, wordt de code behorende bij deze **elif** uitgevoerd. Zo niet, dan wordt de boolean expressie bij de tweede **elif** geëvalueerd. Etcetera. Slechts als alle boolean expressies in de constructie **False** blijken te zijn, wordt het blok code bij de **else** uitgevoerd.



Afb. 6.2: Stroomdiagram dat een meer-weg beslissing weergeeft.

Voor het voorbeeld hierboven betekent dit dat bij de eerste **`elif`** de test `leeftijd < 18` volstaat, en niet hoeft te worden aangevuld met een test **`and`** `leeftijd >= 12`, want als `leeftijd` kleiner geweest zou zijn dan 12, dan zou de boolean expressie voor de **`if`** al **`True`** zijn geweest, en zou de boolean expressie bij de eerste **`elif`** niet eens door Python zijn gezien.

Het toevoegen van **`else`** is altijd optioneel. In de meeste gevallen waarin ik zelf meerdere **`elif`**s gebruik, zet ik wel een **`else`**, al was het maar voor het afvangen van fouten.

Opgave Schrijf een programma dat een variabele `gewicht` heeft. Als `gewicht` groter is dan 20 (kilo), print je: "Er moet een toeslag van $25 betaald worden voor baggage die te zwaar is." Als `gewicht` kleiner is dan 20, print je: "Goede reis!" Als `gewicht` precies 20 is, print je: "Poeh! Dat gewicht is precies goed!" Wijzig de waarde van `gewicht` een paar keer om de code te testen.

6.2.6 Geneste condities

Gegeven de syntactische regels voor **if-elif-else** constructies en inspringing, is het mogelijk om **if** statements op te nemen in de code blokken behorende bij andere **if** statements. Zo'n geneste **if** wordt alleen uitgevoerd als de boolean expressie behorende bij het code blok waarin de geneste **if** staat **True** is.

listing0607.py

```
x = 41
if x%7 == 0:
    # --- Hier begint een genest blok code  ---
    if x%11 == 0:
        print( x, "is deelbaar door 7 en 11." )
    else:
        print( x, "is deelbaar door 7, maar niet door 11." )
    # --- Hier eindigt een genest blok code ---
elif x%11 == 0:
    print( x, "is deelbaar door 11, maar niet door 7." )
else:
    print( x, "is niet deelbaar door 7 of 11." )
```

Deze code is equivalent aan het algoritme dat wordt weergegeven in afbeelding 6.3.

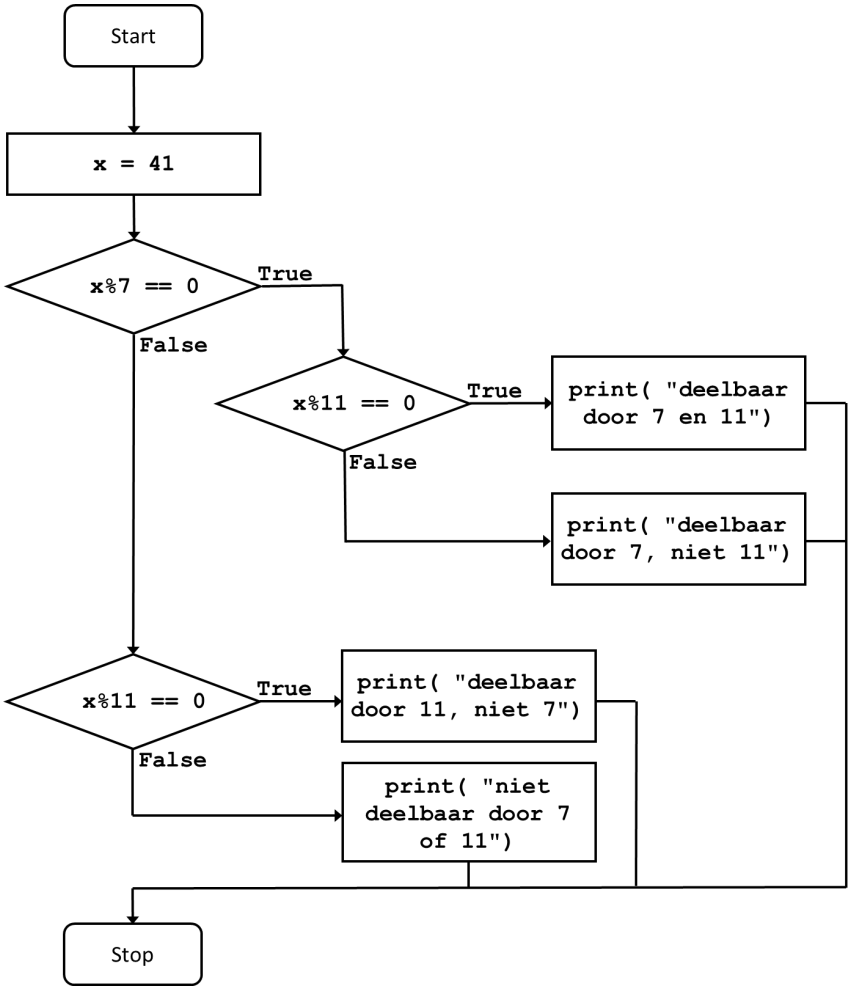
Opgave Wijzig de waarde van `x` en controleer de resultaten.

In het voorbeeld hierboven wil ik alleen maar het nesten van **if** statements demonstreren. Er zijn leesbaardere manieren om te doen wat deze code doet. Het nesten van **if** statements kan vaak vermeden worden door **elif**s te gebruiken. Bijvoorbeeld, de code hieronder doet hetzelfde als de "leeftijd" code hierboven, maar nu met geneste **if**s in plaats van **elif**s.

listing0608.py

```
leeftijd = 21
if leeftijd < 12:
    print( "Je bent een kind!" )
else:
    if leeftijd < 18:
        print( "Je bent een teenager!" )
    else:
        if leeftijd < 30:
            print( "Je bent nog jong!" )
        else:
            if leeftijd < 50:
                print( "Beginnen grijze haren te komen?" )
            else:
                print( "Wegen de jaren zwaar?" )
```

Ik neem aan dat je het ermee eens bent dat de versie met **elif**s prettiger leest.



Afb. 6.3: Stroomdiagram dat een geneste conditie weergeeft.

6.3 Vroegtijdig afbreken

Soms wil je een programma vroegtijdig beëindigen onder bepaalde condities. Bijvoorbeeld, je programma vraagt de gebruiker om een waarde, en voert met die waarde een aantal berekeningen uit. Als de gebruiker een waarde invoert die niet in de berekeningen gebruikt kan worden, wil je het programma meteen beëindigen. Dat kun je als volgt coderen:

```
from pcinput import getInteger

num = getInteger( "Geef een positief geheel getal: " )
if num < 0:
    print( "Je had een positief geheel getal moeten geven!" )
else:
    print( "Ik handel je getal", num, "af" )
    print( "Nog meer code" )
    print( "Honderden regels code" )
```

Het is irritant dat een groot deel van het programma al één inspringing diep staat, terwijl je er de voorkeur aan zou hebben gegeven als het programma gestopt was na de foutmelding, en de rest van het programma zonder inspringing geschreven zou kunnen worden. Je kunt dat regelen met behulp van de functie `exit()` die in de module `sys` staat. De code is dan:

listing0609.py

```
from pcinput import getInteger
from sys import exit

num = getInteger( "Geef een positief geheel getal: " )
if num < 0:
    print( "Je had een positief geheel getal moeten geven!" )
    exit()

print( "Ik handel je getal", num, "af" )
print( "Nog meer code" )
print( "Honderden regels code" )
```

Als je deze code uitvoert en een negatief getal ingeeft, kan het gebeuren dat je ziet dat Python een `SystemExit` melding genereert, die eruit ziet als een grote, lelijke fout. Dit is afhankelijk van de editor die je gebruikt (IDLE geeft deze melding niet). Het is geen fout, ook al ziet het er zo uit. Deze melding zegt alleen dat je het programma geforceerd beëindigd hebt, maar dat is precies wat je wilde doen. Je mag dit beschouwen als een nette manier van afbreken.

In principe moet je meldingen van Python over je programma niet negeren, maar deze is een uitzondering. Je mag je programma op deze manier afbreken. In hoofdstuk 8 zal ik een andere manier van programma afbreken bespreken, die ervoor zorgt dat je deze melding niet krijgt. Dat kun je tegen die tijd gebruiken (als de melding je echt stoort), maar vooralsnog moet je hem maar accepteren.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat boolean expressies zijn
- Boolean waardes **`True`** en **`False`**
- Vergelijkingen met <, <=, ==, >, >=, en !=
- De **`in`** operator
- Logische operatoren **`and`**, **`or`**, en **`not`**
- Conditionele statements met **`if`**, **`elif`**, en **`else`**
- Blokken code
- Inspringing
- Geneste condities
- `exit()`

Opgaves

Opgave 6.1 Cijfers voor proefwerken en tentamens vallen tussen nul en 10 (inclusief nul en 10), en worden altijd afgerond op halve punten. De Amerikaanse stijl van beoordelen gebruikt letters. Ter vergelijking, de cijfers 8.5 tot en met 10 zijn in Amerika "A," 7.5 en 8 zijn "B," 6.5 en 7 zijn "C," 5.5 en 6 zijn "D," en 5 en lager is "F." Schrijf code die deze vertaling van cijfers naar letters maakt, waarbij de gebruiker gevraagd wordt om het cijfer in te geven. Als de gebruiker een cijfer buiten het gegeven bereik ingeeft, moet je een foutmelding geven. Je hoeft niet te eisen dat de gebruiker alleen getallen ingeeft die geheel zijn of eindigen in .5, maar je mag dat wel doen, aangezien je op zich voldoende kennis hebt om dit op te lossen. In dat geval, geef een zinvolle foutmelding als de gebruiker iets incorrects ingeeft.

Opgave 6.2 Snap je welke redeneerfout gemaakt is in de volgende code?

exercise0602.py

```
score = 98.0
if score >= 60.0:
    oordeel = 'D'
elif score >= 70.0:
    oordeel = 'C'
elif score >= 80.0:
    oordeel = 'B'
elif score >= 90.0:
    oordeel = 'A'
else:
    oordeel = 'F'
print( oordeel )
```

Opgave 6.3 Vraag de gebruiker om een string. Druk af hoeveel *verschillende* klinkers er in de string zitten. De hoofdletter versie van een klinker wordt beschouwd als gelijk aan de kleine-letter versie. Probeer de uitvoer een beetje netjes te maken (bijvoorbeeld, het is lelijk om te zeggen: "Er zitten 1 verschillende klinkers in de string"). Voorbeeld: als de gebruiker als string "De Heilige Handgranaat van Antioch" ingeeft, zegt het programma dat er 4 verschillende klinkers in de string zitten.

Opgave 6.4 Je kunt kwadratische vergelijkingen oplossen met de wortelformule. Kwadratische vergelijkingen hebben de vorm $Ax^2 + Bx + C = 0$. Dit soort vergelijkingen heeft nul, één of twee oplossingen. De eerste oplossing is $(-B + sqrt(B^2 - 4AC))/(2A)$. De tweede oplossing is $(-B - sqrt(B^2 - 4AC))/(2A)$. Er zijn geen oplossingen als de waarde onder de wortel negatief is. Er is één oplossing als de waarde onder de wortel nul is. Schrijf een programma dat de gebruiker vraagt om waardes voor A, B, en C, en dan rapporteert hoeveel oplossingen er zijn, en welke oplossingen dat zijn. Let erop dat je ook afhandelt wat er gebeurt als A nul is (er is dan één oplossing, namelijk $-C/B$), of als A en B allebei nul zijn.