Hoofdstuk 5

Eenvoudige Functies

Ik heb in de voorgaande hoofdstukken al gesproken over een aantal basis "functies," zoals `print()` en `int()`. In dit hoofdstuk worden deze functies in meer detail besproken, en zal ik een aantal nieuwe functies introduceren die nuttig gaan zijn in de volgende hoofdstukken. In hoofdstuk 8 ga ik bespreken hoe je je eigen functies kunt maken.

5.1 Elementen van een functie

A functie is een blok herbruikbare code dat een bepaalde actie uitvoert. Om een functie aan het werk te zetten, roep je hem aan (Engels: "call"), met de parameters die de functie nodig heeft. Je hoeft niet te weten hoe de functie precies werkt. Je hoeft slechts drie dingen te weten:

- De naam van de functie
- De parameters die de functie nodig heeft (als die er zijn)
- De waarde die de functie teruggeeft (als er zo'n waarde is)

Ik ga deze elementen één voor één bespreken.

5.1.1 Functie naam

Iedere functie heeft een naam. Net als variabele namen, mogen functie namen alleen bestaan uit letters, cijfers, en underscores, en mogen ze niet starten met een cijfer. Vrijwel alle standaard Python functies bestaan alleen uit kleine letters. Gewoonlijk is de naam van een functie een korte beschrijving van wat de functie doet.

Als je in een tekst refereert aan een functie, is het de gewoonte dat je achter de naam van de functie een openings- en sluithaakje zet, omdat functies in code altijd die haakjes hebben (zelfs al staat er niks tussen de haakjes).

5.1.2 Parameters

Sommige functies worden aangeroepen met parameters ("argumenten"), die meestal verplicht zijn. De parameters worden geplaatst tussen de haakjes die achter de functienaam staan. Als er meerdere parameters zijn, moet je er komma's tussen zetten.

De parameters zijn de waardes die de programmeur aan de functie geeft om te verwerken. Bijvoorbeeld, de functie **int**() wordt aangeroepen met één parameter, namelijk de waarde waarvan de functie probeert een integer representatie te maken. De **print**() functie mag worden aangeroepen met een willekeurig aantal parameters (zelfs nul), die de functie op het scherm zal tonen, waarna de functie naar een nieuwe regel op het scherm zal gaan.

Over het algemeen is het zo dat een functie de waardes van de parameters niet kan wijzigen. Bekijk bijvoorbeeld de volgende code:

```
x = 1.56
print( int( x ) )
print( x )
```

Als je deze code uitvoert, zie je dat **int**() niet de waarde van x heeft aangepast; de functie heeft alleen aan **print**() doorgegeven wat de integer representatie van de waarde van x is. De reden dat dit zo is, is dat over het algemeen alleen de waarde van parameters wordt doorgegeven (Engels: "passed by value"). Dit betekent dat de functie geen toegang heeft tot de variabelen die als parameters gebruikt worden, maar dat de functie kopieën krijgt van de waardes die in de parameters staan. Ik zeg "over het algemeen" omdat dit niet geldt voor alle data types, maar het geldt in ieder geval voor de data types die tot op dit moment bediscussieerd zijn. Pas in hoofdstuk 12 ga ik spreken over data types die wel door functies gewijzigd kunnen worden, en op dat moment zal ik dat heel duidelijk maken.

Als een functie meerdere parameters krijgt, maakt de volgorde uit. Bijvoorbeeld, de standaard functie **pow**() krijgt twee parameters, en rekent de waarde uit van de eerste die wordt verheven tot de macht weergegeven door de tweede.

```
basis = 2
exponent = 3
print( pow( basis, exponent ) )
```

De namen die aan de variabelen zijn gegeven doen niet ter zake, de eerste wordt verheven tot de macht die de tweede is. Dus de volgende code geeft een ander antwoord dan de vorige, omdat de variabelen in een andere (nogal verwarrende) volgorde aan de functie worden doorgegeven.

```
basis = 2
exponent = 3
print( pow( exponent, basis ) ) # verwarrende variabele namen
```

Wat gebeurt er als je een functie aanroept met parameter waardes waarmee de functie niet kan werken? Bijvoorbeeld, wat gebeurt er als ik **int**() aanroep met een string die geen integer bevat, of **pow**() met strings in plaats van getallen? Dat leidt meestal tot "runtime errors" (fouten tijdens de uitvoering van code). Bijvoorbeeld, beide regels in de volgende code leiden tot runtime errors.

```
x = pow( 3, "2" )
y = int( "twee-en-een-half" )
```

5.1.3 Retour waarde

Een functie heeft vaak een retour waarde. Als een functie een waarde retourneert, kun je die in je code gebruiken. Bijvoorbeeld, de **int()** functie retourneert een integer representatie van de parameter die is meegegeven. Je kunt deze retour waarde in een variabele stoppen middels een assignment, of je kunt de waarde op een andere manier gebruiken, bijvoorbeeld deze onmiddellijk printen. Je kunt er zelfs voor kiezen niks met de waarde te doen, maar in dat geval had het waarschijnlijk weinig zin om de functie aan te roepen.

```
x = 2.1
y = '3'
z = int( x )
print( z )
print( int( y ) )
```

Zoals je hierboven kunt zien, kun je zelfs functie aanroepen als parameter meegeven aan een functie, bijvoorbeeld, in de laatste regel van de code hierboven krijgt de **print()** functie als waarde een aanroep van de **int()** functie mee. De aanroep van **int()** vindt dan plaats voordat de **print()** wordt afgehandeld, dus de return waarde van **int()** is een parameter voor **print()**.

Niet alle functies retourneren een waarde. Bijvoorbeeld, **print()** geeft geen waarde terug. Als je niet uitkijkt, kan dit tot vreemd gedrag van je code leiden. Voer maar eens de volgende code uit:

```
print( print( "Hello, world!" ) )
```

Je ziet dat deze code twee regels print. De eerste bevat de tekst "Hello, world!" en de tweede het woord "None." Wat betekent dat woord "None"? Om dat te begrijpen, moet je uitpluizen hoe Python deze regel code verwerkt.

Wanneer Python deze regel code bekijkt, ziet het eerst **print(** <iets> **)**. Omdat <iets> een argument is, moet dat eerst geëvalueerd worden. <iets> is **print(** <nog_iets> **)**. Omdat <nog_iets> een argument is, moet Python dat eerst evalueren. <nog_iets> is de string "Hello, world!". Die hoeft niet verder geëvalueerd te worden, dus **print()** wordt uitgevoerd met als argument "Hello, world!", en Python "vangt" de retour waarde van deze uitvoering omdat die nodig is als <iets>.

En daar is het probleem: **print()** heeft geen retourwaarde, dus er is niks wat Python kan substitueren voor <iets>. Voor dit soort situaties heeft Python een speciale waarde die **None** heet. Dus het eerste **print()** commando krijgt als argument **None** mee, en dat leidt ertoe dat Python "None" op het scherm afdrukt.

None is een speciale waarde die aangeeft "geen waarde." Als je **None** probeert af te drukken, drukt Python het woord "None" af, maar dat is niet een string met de waarde "None". Het woord geeft slechts aan dat er niks af te drukken was. **None** is niet hetzelfde als een lege

string (""). Een lege string heeft nog steeds een waarde, namelijk een string met lengte nul. **`None`** is geen string, geen float, geen integer, niks. Dus wees voorzichting met het aanroepen van een functie als parameter; als de functie geen retour waarde heeft, kunnen er vreemde dingen gebeuren.

5.1.4 Een functie is een zwarte doos

In de wereld van programmeurs betekent een "zwarte doos" iets waar je wat in kunt stoppen en wat uit kunt halen, maar waarvan je niet kunt zien hoe het van binnen werkt. Je mag een functie beschouwen als een zwarte doos: het is niet van belang te weten hoe de functie werkt of hoe de code in de functie eruit ziet. Slechts de naam, de parameters, en de retour waarde moet je kennen om de functie te kunnen gebruiken. Het zou kunnen dat de functie intern variabelen aanmaakt en berekeningen doet, maar die hebben geen effect op de rest van de code.

...Tenminste, als de functie netjes geïmplementeerd is. Een functie die geen effect heeft op de rest van de code heet een "pure functie." Alle functies die ik hier bespreek zijn "pure functies." Maar het feit dat er een aparte naam is voor functies die de rest van de code niet aantasten, geeft al aan dat er ook functies bestaan die niet "puur" zijn. Dit is het duidelijkst bij functies waar de gebruiker parameters aan mee geeft, waarbij de inhoud van de variabelen die als parameter worden meegegeven gewijzigd wordt. Dat kan niet voor iedere soort variabele. En als het gebeurt kan dat best acceptabel zijn, als het de bedoeling is en goed gedocumenteerd is. Zulke functies heten "modifiers." Ik bespreek ze in een later hoofdstuk.

Vooralsnog mag je aannemen dat iedere functie die je gebruikt, geen effect heeft op de rest van de code. Het is veilig om functies aan te roepen.

5.2 Basis functies

Ik introduceer nu een aantal basis functies die je in je programma's kunt gebruiken.

5.2.1 Type casting

Ik heb al gesproken over type casting functies in hoofdstuk 4, maar nu ik meer details van functies heb gegeven, kan ik de beschrijving completeren.

- **`float`**`()` heeft één parameter en retourneert een floating-point representatie van de waarde van de parameter. Als de waarde een integer is, krijg je dezelfde waarde terug als float. Als de parameter een float is, krijg je dezelfde waarde terug. Als de parameter een string bevat die je als integer of float zou kunnen interpreteren, dan krijg je die float terug als waarde. Anders krijg je een runtime error.
- **`int`**`()` heeft één parameter en retourneert een integer representatie van de waarde van de parameter. Als de waarde een integer is, krijg je dezelfde waarde terug. Als de waarde een float is, krijg je een integer terug die de waarde van de float naar beneden heeft afgerond. Als de parameter een string bevat die je als integer zou kunnen interpreteren, dan krijg je die integer terug als waarde. Anders krijg je een runtime error.

- **str**() heeft één parameter en retourneert een string representatie van de waarde van de parameter.

Opgave Wat denk je dat de volgende code doet? Als je het niet weet, test dan de code.

```
print( 10 * int( "100,000,000" ) )
```

Opgave De code hierboven geeft een runtime error. Los het probleem op door precies twee tekens te verwijderen.

5.2.2 Berekeningen

Een paar basis Python functies helpen met berekeningen.

- **abs**() krijgt een numerieke parameter waarde. Als de waarde positief is, wordt hij weer geretourneerd. Als de waarde negatief is, wordt de waarde vermenigvuldigd met -1 geretourneerd.
- **max**() krijgt twee of meer numerieke parameters en retourneert de grootste.
- **min**() krijgt twee of meer numerieke parameters en retourneert de kleinste.
- **pow**() krijgt twee numerieke parameters en retourneert de eerste verheven tot de macht weergeven door de tweede. Optioneel mag je een derde parameter meegeven, die een integer moet zijn. Als je dat doet, krijg je de waarde modulo de derde parameter terug.
- **round**() krijgt een numerieke parameter die wiskundig wordt afgerond. Optioneel mag je als tweede parameter een integer meegeven die aangeeft hoeveel cijfers achter de komma behouden moeten worden. Als de tweede parameter niet wordt meegegeven, wordt afgerond op gehele getallen.

Opgave Bekijk de code hieronder en bedenk wat er op het scherm getoond wordt. Voer daarna de code uit en controleer of je gelijk hebt.

listing0501.py

```
x = -2
y = 3
z = 1.27

print( abs( x ) )
print( max( x, y, z ) )
print( min( x, y, z ) )
print( pow( x, y ) )
print( round( z, 1 ) )
```

5.2.3 len()

len() is a basis functie die één parameter krijgt, en die de lengte van die parameter teruggeeft. Op dit moment is het enig zinvolle data type dat je mee kunt geven aan **len**()

een string, waarvan je dan de lengte krijgt. In latere hoofdstukken volgen meer data types waarvan je de lengte kunt bepalen.

Opgave Wat print de code hieronder? Controleer of je vermoeden klopt.

```
print( len( 'man' ) )
print( len( 'mango' ) )
print( len( "" ) )  # "" is een lege string
```

Opgave En wat denk je van de code hieronder? Denk goed na voor je een antwoord geeft.

```
print( len( 'mango\'s' ) )
```

5.2.4 `input()`

In veel programma's wil je dat de gebruiker van het programma data verstrekt. Je kunt de gebruiker vragen een string in te typen met behulp van de functie **`input()`**. De functie krijgt één parameter mee, namelijk een string. Deze string is de zogeheten "prompt." Als **`input()`** wordt aangeroepen, wordt de prompt op het scherm gezet en mag de gebruiker een tekst ingeven. De gebruiker mag ingeven wat hij of zij wil, inclusief niks. De gebruiker sluit het ingeven af met een druk op de `Enter` toets. De retour waarde van de functie is hetgeen de gebruiker heeft ingegeven, exclusief de `Enter`.

Het hangt van de omgeving waar je je programma in draait hoe de gebruiker de ingave precies doet. Soms wordt er een rechthoek op het scherm getoond met de prompt ervoor waarin de gebruiker mag typen. Als je een Python programma draait op de command-line, moet de gebruiker ook de ingave op de command-line doen. Sommige editors tonen een popup-window waarin de gebruiker moet typen.

Hier is een voorbeeld:

```
tekst = input( "Geef een tekst in: " )
print( "Je hebt het volgende ingetypt:", tekst )
```

Realiseer je dat **`input()`** altijd een string teruggeeft. Bekijk de volgende code:

```
nummer = input( "Geef een getal: " )
print( "Je getal in het kwadraat is", nummer * nummer )
```

Het maakt niet uit wat je ingeeft, deze code geeft een runtime error. Omdat **`input()`** een string teruggeeft, wordt op de tweede regel een poging gedaan twee strings met elkaar te vermenigvuldigen, en dat kan niet. Het maakt niet uit of je string een getal bevat: een string is een string. Je kunt het probleem oplossen middels type casting, bijvoorbeeld:

```
nummer = input( "Geef een getal: " )
nummer = float( nummer )
print( "Je getal in het kwadraat is", nummer * nummer )
```

Voor deze code geldt dat als de gebruiker een getal ingeeft, de code doet wat hij moet doen. Maar als de gebruiker iets anders ingeeft, dat niet in een getal omgezet kan worden, krijg je toch weer een runtime error. Ook dat probleem kan opgelost worden, maar ik heb nog niet de zaken uitgelegd die je nodig hebt om dit probleem aan te pakken, en het zal nog tot hoofdstuk 17 duren voordat ik eraan toekom. Tot dat moment geef ik iets later in dit hoofdstuk een methode die je kunt gebruiken om je programma om een getal te laten vragen zonder dat het "crasht" als de gebruiker een wijsneus probeert te zijn en iets anders ingeeft.

Opgave Schrijf code die de gebruiker twee getallen laat ingeven, en die dan toont wat de uitkomst is als je ze optelt en als je ze vermenigvuldigt. Je code mag een runtime error geven als de gebruiker iets ingeeft wat geen getal is.

5.2.5 `print()`

De functie **`print()`** krijgt nul of meer parameters mee, toont ze op het scherm (als het er meerdere zijn, met spaties ertussen), en gaat daarna naar de volgende regel. Dus als je twee **`print`** statements gebruikt, komt de output van de tweede onder die van de eerste te staan.

Als **`print()`** wordt aangeroepen zonder parameters, gaat de functie alleen naar de volgende regel. Zo kun je lege regels op het scherm zetten.

Je mag als parameters meegeven wat je wilt, en **`print()`** zal zijn best doen het op het scherm weer te geven. Vooralsnog zul je echter alleen basis data types printen.

`print()` kan twee speciale parameters meekrijgen, die `sep` en `end` heten.

`sep` geeft aan wat er getoond moet worden tussen iedere twee parameters, en is als default een spatie. Je kunt die spatie wijzigen in iets anders via `sep`, inclusief in een lege string.

`end` geeft aan wat **`print()`** moet tonen nadat alle parameters zijn getoond, en is als default een "nieuwe regel." Door `end` te wijzigen kun je ervoor zorgen dat **`print()`** iets anders doet dan naar een nieuwe regel gaan als hij klaar is met het tonen van parameters.

On `sep` en `end` te gebruiken, moet je speciale parameters opnemen, namelijk parameters `sep=<string>` en/of `end=<string>` (merk op: als in een beschrijving van code je iets ziet dat tussen < en > staat, betekent dat meestal dat je dat niet letterlijk moet typen, maar moet vervangen door van wat de term tussen de < en > aangeeft, dus `<string>` betekent dat je daar een string moet plaatsen). Bijvoorbeeld:

```
print( "X", "X", "X", sep="x" )
print( "X", end="" )
print( "Y", end="" )
print( "Z" )
```

Als je deze code uitvoert, zie je twee regels. De eerste bevat "XxXxX," aangezien er is aangegeven dat er drie keer een hoofdletter "X" afgedrukt moet worden met tussen iedere twee als separator een kleine letter "x." De tweede regel bevat "XYZ", omdat weliswaar dit drie verschillende aanroepen van **`print()`** betreft, maar na ieder van de eerste twee er niet naar een volgende regel wordt gegaan.

5.2.6 format()

format() is een nogal complexe functie die op een speciale manier gebruikt moet worden. De functie staat je toe een geformatteerde string te bouwen, dus een string waarin bepaalde waardes op een specifiek geformatteerde manier zijn opgenomen. Om een voorbeeld te geven, stel je voor dat ik de uitkomst van de berekening van een float wil tonen:

```
print( 7/11 )
```

Nu stel ik dat je de uitkomst moet tonen met 3 decimalen. Dat zou je kunnen doen met de **round()** functie, dus iets als:

```
print( round( 7/11, 3 ) )
```

Dit werkt, maar misschien heb ik extra eisen. Misschien stel ik dat je 10 posities ruimte moet reserveren voor deze uitkomst, en dat je binnen die 10 posities de uitkomst links moet aanlijnen. Dat lijkt lastig te realiseren, maar middels de **format()** functie is het vrij eenvoudig om waardes te formatteren op allerlei manieren. De volgende toepassing van **format()** realiseert het afronden op drie decimalen:

```
print( "{:.3f}".format( 7/11 ) )
```

format() "werkt" op een string. Tot op dit moment heb ik alleen functies gebruikt die aangestuurd worden via parameters. Echter, er zijn functies die alleen werken met een specifiek data type, en die op zo'n manier gedefinieerd zijn dat een variabele (of waarde) van dat data type vóór de functie naam moet staan, met een punt tussen de variabele (of waarde) en de naam van de functie. De reden hiervoor is een techniek die "object oriëntatie" heet, en die ik zal bediscussiëren in hoofdstukken 20 tot en met 23. Je hoeft nu alleen te weten dat zulke functie "methodes" genoemd worden, en dat je, om ze aan te roepen, een variabele (of waarde) van het juiste type voor de aanroep van de methode moet zetten, met een punt ertussen. Die variabele (of waarde) zelf is ook beschikbaar voor de methode, net als de parameters.

De **format()** methode (laten we de correcte benaming gebruiken, het is geen functie maar een methode) wordt als volgt aangeroepen: <string>**.format()**. Hij retourneert een nieuwe string, die een geformatteerde versie is van de string waarvoor de methode is aangeroepen. **format()** kan een willekeurig aantal parameters meekrijgen, die in de geformatteerde string ingebracht kunnen worden op specifieke plaatsen.

De plaatsen waar **format()** de parameter waardes in de string plaatst worden in de string aangegeven middels accolades ({ en }). Als je alleen {} gebruikt om de parameters aan te duiden, worden ze van links naar rechts afgehandeld. Bijvoorbeeld:

```
print( "De eerste drie getallen zijn {}, {} en {}.".format(
    "een", "twee", "drie" ) )
```

Als je ze in een andere volgorde wilt afhandelen, kun je de volgorde bepalen door een getal tussen de acolades te zetten. De eerste parameter is nummer 0, de tweede nummer 1, de derde nummer 2, etcetera (als je het vreemd vindt om nummering te beginnen bij nul, weet dan dat dat gebruikelijk is in programmeertalen, en dat je het nog vaker zult tegenkomen in dit boek). Bijvoorbeeld:

```
print( "Achterwaarts zijn ze {2}, {1} en {0}.".format(
    "een", "twee", "drie" ) )
```

format() kan variabelen van ieder type verwerken, zolang ze maar een fatsoenlijke string representatie hebben. Bijvoorbeeld, **format()** kan getallen verwerken, en zelfs verschillende soorten data types mixen:

```
print( "De eerste drie getallen zijn {}, {} en {}.".format(
    1, "twee", 3.0 ) )
```

Als je de parameters op een specifieke manier wil formatteren, zijn daar mogelijkheden voor, als je een dubbele-punt (:) tussen de accolades zet, na het volgorde-nummer als je dat gebruikt, met rechts van de dubbele-punt formatteringsinstructies. Ik zal een aantal formatteringsinstructies opnoemen.

Ik begin met instructies voor string parameters. Als je een bepaalde hoeveelheid tekens wilt reserveren voor een string, dan kun je dat aangeven met een integer rechts van de dubbele-punt. Dit wordt de "precisie" genoemd. De volgende code gebruikt een precisie van 7.

```
print( "De eerste drie getallen zijn {:7}, {:7} en {:7}.".format(
    "een", "twee", "drie" ) )
```

Als de precisie te kort is voor de lengte van de string, neemt **format()** gewoon meer ruimte voor de string. Je kunt de precisie dus niet gebruiken om een string voortijdig af te breken.

```
print( "De eerste drie getallen zijn {:3}, {:3} en {:3}.".format(
    "een", "twee", "drie" ) )
```

Als je precisie gebruikt, kun je de parameter links aanlijnen, centreren, of rechts aanlijnen in de ruimte die je hebt gereserveerd. Dat doe je door een "alignment" teken te plaatsen tussen de dubbele punt en de precisie. Deze "alignment" tekens zijn < voor links aanlijnen, ^ voor centreren, en > voor rechts aanlijnen.

```
print( "De eerste drie getallen zijn {:<7}, {:^7} en {:>7}.".
    format( "een", "twee", "drie" ) )
```

Ik ga nu over op formatteringsinstructies voor getallen. Als je een getal wilt laten interpreteren als een integer, moet je de kleine letter "d" plaatsen rechts van de dubbele punt ("d" staat hierbij voor "decimaal"). Wil je dat het getal wordt geïnterpreteerd als een float, dan moet je een "f" plaatsen rechts van de dubbele-punt. **format()** maakt de juiste conversie voor je als dat kan. Je kunt echter niet van een float een integer maken, want dat veroorzaakt een runtime error.

```
print( "{} gedeeld door {} is {}".format( 1, 2, 1/2 ) )
print( "{:d} gedeeld door {:d} is {:f}".format( 1, 2, 1/2 ) )
print( "{:f} gedeeld door {:f} is {:f}".format( 1, 2, 1/2 ) )
```

Net als bij strings, kun je voor getallen precisie en aanlijning gebruiken. Dit doe je op dezelfde manier. En net als bij strings geldt dat als de precisie niet voldoende groot is, de functie gewoon de ruimte neemt die nodig is. Let erop dat een eventueel min-teken en een eventuele punt in een float ook plaats nodig hebben.

```
print( "{:5d} gedeeld door {:5d} is {:5f}".format( 1, 2, 1/2 ) )
print( "{:<5f} gedeeld door {:^5f} is {:>5f}".format( 1,2,1/2 ))
```

Tenslotte, en misschien het meest nuttig, kun je aangeven met hoeveel decimalen een float getoond moet worden, door een punt en een getal te plaatsen links van de letter "f." De `format()` methode zal het getal afronden tot het juiste aantal decimalen. Merk op dat je ook mag aangeven dat je nul decimalen wilt zien door `.0` te gebruiken, wat ervoor zal zorgen dat een float wordt getoond als een integer.

```
print( "{:.2f} gedeeld door {:.2f} is {:.2f}".format( 1,2,1/2 ))
```

De combinatie van precisie, aanlijning, en decimalen staat je toe om redelijk uitziende tabellen te tonen.

listing0502.py

```
s = "{:>5d} keer {:>5.2f} is {:>5.2f}"
print( s.format( 1, 3.75, 1 * 3.75 ) )
print( s.format( 2, 3.75, 2 * 3.75 ) )
print( s.format( 3, 3.75, 3 * 3.75 ) )
print( s.format( 4, 3.75, 4 * 3.75 ) )
print( s.format( 5, 3.75, 5 * 3.75 ) )
```

5.3 Modules

Python biedt basis functies, waarvan ik er een aantal hierboven besproken heb. Naast die basis functies biedt Python ook een groot aantal zogeheten "modules," waarin zich vele nuttige functies bevinden. Om de functies van een module te gebruiken in een programma, moet je de juiste module importeren door boven in je programma **import** <modulenaam> op te nemen. Je kunt dan alle functies die in de betreffende module staan in je programma gebruiken, maar je moet de functie-aanroepen vooraf laten gaan door de naam van de module en een punt. Bijvoorbeeld, om de functie `sqrt()` uit de `math` module (die de wortel van een getal trekt) te gebruiken, roep je `math.sqrt()` aan nadat je `math` geïmporteerd hebt.

Als alternatief kun je ook specifieke functies vanuit een module importeren, via:
from <module> **import** <functie1>, <functie2>, <functie3>, ...
Het voordeel van een dergelijke manier van functies importeren is dat je in je code niet de naam van de module voor de functie-aanroep hoeft te zetten.

Bijvoorbeeld:

```
import math

print( math.sqrt( 4 ) )
```

is equivalent aan:

```
from math import sqrt

print( sqrt( 4 ) )
```

Als je een functie onder een andere naam in je programma wilt gebruiken, kun je dat doen middels het gereserveerde woord **as**. Dit kan zinvol zijn als je meerdere modules gebruikt waarin toevallig functies voorkomen die dezelfde naam hebben.

```
from math import sqrt as squareroot

print( squareroot( 4 ) )
```

Ik bespreek nu een aantal functies uit twee veelgebruikte standaard modules, en een aantal functies die in een module staan die ik voor dit boek gebouwd heb (in hoofdstuk 8 leg ik uit hoe je je eigen modules kunt maken). Er zijn veel meer standaard modules naast de modules die ik hieronder noem, en sommige ervan komen later nog aan de orde. Andere zul je zelf moeten opzoeken als je ze nodig hebt. Je mag er echter van uitgaan dat voor ieder min-of-meer-algemeen probleem dat je wilt oplossen, er iemand is geweest die er een module voor ontwikkeld heeft die het eenvoudig of zelfs triviaal maakt om het probleem op te lossen. Dus in de praktijk geldt: ga niet meteen zelf coderen, maar zoek eerst even uit of je niet gebruik kunt maken van de moeite die iemand anders gedaan heeft.

5.3.1 `math`

De `math` module bevat een aantal nuttige wiskundige functies. Deze functies zijn meestal zeer efficiënt, en retourneren meestal een float. Ik noem hier een klein aantal van de functies; als je er meer wilt kennen kun je ze opzoeken in de Python referentie):

- `exp()` krijgt één numerieke parameter en retourneert e tot de macht van die parameter. Als je niet weet wat e is: e is een speciaal getal met veel interessante eigenschappen, en wordt veel gebruikt in natuurkunde, wiskunde, en statistiek.
- `log()` krijgt één numerieke parameter en retourneert het natuurlijk logaritme van die parameter. Het natuurlijk logaritme is de waarde die de parameter als uitkomst heeft als je e verheft tot deze waarde. Net als e heeft het natuurlijk logaritme toepassingen in natuurkunde, wiskunde, en statistiek.
- `log10()` krijgt één numerieke parameter en retourneert het logaritme met 10 als basis van de parameter.
- `sqrt()` krijgt één numerieke parameter en retourneert de vierkantswortel van die parameter.

Bijvoorbeeld:

listing0503.py

```
from math import exp, log

print( "De waarde van e is bij benadering", exp( 1 ) )
```

```
e_sqr = exp( 2 )
print( "e kwadraat is", e_sqr, "wat betekent" )
print( "dat log(", e_sqr, ") gelijk is aan", log( e_sqr ) )
```

5.3.2 random

De random module bevat functies die pseudo-toevalsgetallen genereren. Ik zeg "pseudo-toevalsgetallen" en niet "toevalsgetallen," aangezien het onmogelijk is voor digitale computers om echt toevalsgetallen te genereren. Maar voor alle toepassingen mag je ervan uitgaan dat deze module toevalsgetallen genereert.

- random() krijgt geen parameters, en retourneert een toevalsgetal als een float binnen het bereik $[0, 1)$, dat wil zeggen een bereik tussen nul en 1, waarbij 0.0 wel meedoet maar 1.0 niet.
- randint() krijgt twee parameters, beide integers, waarbij de eerste kleiner dan of gelijk aan de tweede moet zijn. Het retourneert een toevalsgetal dat een integer is dat ligt binnen het bereik dat begrensd wordt door deze twee parameters, inclusief beide parameters. Bijvoorbeeld, randint(2,5) retourneert 2, 3, 4, of 5, elk met een gelijke kans.
- seed() initialiseert de toevalsgetal generator van Python. Als je een lijst van toevalsgetallen wilt hebben die iedere keer hetzelfde is voor je programma, kun je dat voor elkaar krijgen door aan het begin van je programma seed() aan te roepen met een vast getal, bijvoorbeeld 0. Dit kan nuttig zijn bij het testen van je programma. Als je de generator weer echt toevallige getallen wilt laten genereren op een later punt in je programma, kun je seed() nogmaals aanroepen zonder parameter.

For example:

listing0504.py

```
from random import random, randint, seed

seed()
print( "Een toevalsgetal tussen 1 en 10 is", randint( 1, 10 ) )
print( "Een ander is", randint( 1, 10 ) )
seed( 0 )
print( "3 toevalsgetallen zijn:", random(), random(), random() )
seed( 0 )
print( "Dezelfde 3 zijn:", random(), random(), random() )
```

5.3.3 pcinput

pcinput is een module die ik voor dit boek geschreven heb. Je vindt hem in appendix C, en je kunt hem gemakkelijk zelf maken (of eenvoudigweg downloaden via http://www.spronck.net/pythonbook). De module bevat vier handige functies, die de gebruiker op een veilige manier om specifieke input vragen. De functies zijn de volgende:

- `getInteger()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om een integer in te geven. Als de gebruiker iets ingeeft wat geen integer is, wordt gevraagd de input opnieuw in te geven. De functie eindigt pas als de gebruiker een correcte integer heeft ingegeven, en de functie retourneert dan de ingegeven waarde als een integer.
- `getFloat()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om een float in te geven. Als de gebruiker iets ingeeft wat geen float is, wordt gevraagd de input opnieuw in te geven. De functie eindigt pas als de gebruiker een correcte float heeft ingegeven, en de functie retourneert dan de ingegeven waarde als een float.
- `getString()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om een string in te geven. Alles wat de gebruiker ingeeft wordt als correct beschouwd. De functie retourneert de ingegeven waarde, waarbij spaties voor en na de ingegeven tekst verwijderd zijn.
- `getLetter()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om één letter in te geven. Alleen letters van het alfabet zijn acceptabel. Pas als de gebruiker precies één letter heeft ingegeven eindigt de functie, en de letter wordt dan als een hoofdletter geretourneerd.

Deze functies helpen je dus om code te schrijven die de gebruiker vraagt om input met een specifiek data type te verstrekken, omdat ze garanderen dat het programma inderdaad iets binnenkrijgt dat van het gevraagde data type is. De code geeft geen runtime error als de gebruiker iets anders ingeeft. De functies zijn niet erg netjes, omdat ze foutmeldingen geven in het Nederlands als iets fouts ingegeven wordt. Dat betekent dat je deze functies niet moet gebruiken als je een Engelstalig programma schrijft (daar heb ik een andere versie van de module voor), maar om Python te leren zijn deze functies afdoende.

Opgave Creëer of download de `pcinput` module, zorg dat hij staat in de folder waar je je programma's schrijft, en maak dan een Python programma met onderstaande code. Voer het programma uit en test het door iets in te geven wat geen integer is.

```
from pcinput import getInteger

num1 = getInteger( "Geef een geheel getal: " )
num2 = getInteger( "Geef een ander geheel getal: " )

print( num1, "+", num2, "=", num1 + num2 )
```

Opgave Vraag de gebruiker om een string in te geven. Gebruik dan die string als prompt om de gebruiker te vragen een float in te geven.

Merk op: Ik leg niet uit hoe `pcinput` werkt, omdat ik er concepten voor gebruik die pas in hoofdstuk 17 aan bod komen. Je zult later leren hoe je zelf dit soort functies kunt maken. Je hoeft je vooralsnog niet druk te maken over hoe ze werken, je hoeft ze alleen maar te gebruiken. Dat is de houding die je tegenover de meeste standaardfuncties moet hebben: zolang je maar weet wat ze doen, welke parameters ze nodig hebben, en wat ze retourneren, heeft het geen zin na te denken over hoe ze werken.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat functies zijn
- Functie namen
- Functie parameters
- Functie retourwaardes
- Details van type casting functies **`float`**`()`, **`int`**`()`, en **`str`**`()`
- Basis berekeningen met **`abs`**`()`, **`max`**`()`, **`min`**`()`, **`pow`**`()`, en **`round`**`()`
- **`len`**`()`
- **`input`**`()`
- Details van the **`print`**`()` functie
- String formattering met **`format`**`()`
- Wat modules zijn
- De `math` functies `exp()`, `log()`, `log10()`, en `sqrt()`
- De `random` functies `random()`, `randint()`, en `seed()`
- De `pcinput` functies `getInteger()`, `getFloat()`, `getString()`, en `getLetter()`

Opgaves

Opgave 5.1 Vraag de gebruiker om een string, en druk de lengte van de string af. Gebruik de **`input`**`()` functie en niet de `getString()` functie, aangezien de `getString()` functie spaties verwijdert die voor en na de ingegeven tekst staan.

Opgave 5.2 De stelling van Pythagoras zegt dat bij een rechthoekige driehoek het kwadraat van de schuine zijde gelijk is aan de som van de kwadraten van de twee andere zijden (ofwel $a^2 + b^2 = c^2$). Schrijf een programma dat de gebruiker om de lengte van de twee rechte zijden vraagt, en bereken dan de lengte van de schuine zijde (met andere woorden, trek de wortel uit de som van de kwadraten van de twee rechte zijden). Toon hem op een netjes geformatteerde manier. Je hoeft geen rekening te houden met het feit dat de gebruiker ook negatieve waardes of nul zou kunnen ingeven.

Opgave 5.3 Vraag de gebruiker om drie getallen, en toon dan de grootste, de kleinste, en hun gemiddelde afgerond op twee decimalen.

Opgave 5.4 Bereken de waarde van e tot de machten -1, 0, 1, 2, en 3, en toon de resultaten met vijf decimalen op een netjes geformatteerde manier.

Opgave 5.5 Stel dat je een geheel toevalgetal tussen 1 en 10 wilt hebben (inclusief 1 en 10), maar je hebt alleen de `random()` functie beschikbaar (je mag wel functies uit andere modules gebruiken). Hoe doe je dat?