Opdrachten week 2 les 2 – Functies met een returnwaarde en herhalen

In de vorige les heb je kennis gemaakt het **if**-, **elif**- en **else**-statement. Met behulp van deze statements kun je bepaalde code voorwaardelijk uit laten voeren. De code wordt dan alleen uitgevoerd als aan een bepaalde voorwaarde wordt voldaan. Op deze manier is het programma in staat om beslissingen te nemen. Daarnaast heb je kennis gemaakt met functies die zichzelf aanroepen, zogenoemde recursieve functies. Tot slot heb je de vorige les geleerd hoe je input van de gebruiker kunt verwerken in je programma.

Tot nu toe heb je functies geschreven die hun resultaat meteen afdrukken op het scherm. In dit hoofdstuk leer je dat dit helemaal niet handig is en dat het beter is om een functie zijn resultaat terug te laten geven aan de aanroepende code.

In week 1 heb je het **for**-statement leren gebruiken waarmee je bepaalde code herhaald uit kunt laten voeren. Het aantal maal dat de code herhaald zal worden is bij het starten van het **for**-statement bekend. Vorige week heb je nog een andere mogelijkheid geleerd om bepaalde code herhaald uit te laten voeren: het definiëren van een recursieve functie. In deze les ga je nog een Python-statement leren kennen en gebruiken waarmee je bepaalde code herhaald uit kunt laten voeren: het **while**-statement. Het aantal keren dat deze code herhaald zal worden hoeft bij de start van het **while**-statement nog niet bekend te zijn. Dit is een, vaak eenvoudiger, alternatief voor recursie en wordt iteratie genoemd. De code wordt dan uitgevoerd zolang aan een bepaalde voorwaarde wordt voldaan. Op deze manier is het Python-programma in staat om een bepaald deel van de code te herhalen.

Je leert deze les:

- hoe je een waarde kunt teruggeven vanuit een functie door middel van het **return**-statement;
- hoe je dit in een recursieve functie kunt gebruiken om een waarde te berekenen waarbij bepaalde code herhaald moet worden uitgevoerd;
- hoe je kunt controleren of een variabele van een bepaald type is met de ingebouwde functie `isinstance`;
- hoe je bepaalde code kan herhalen met behulp van het **while**-statement;
- wanneer je een herhaling moet implementeren met het **for**-statement en wanneer met het **while**-statement;

- het verschil tussen een recursieve en een iteratieve implementatie van een herhaling en de voor- en nadelen van deze implementaties.

In de volgende les zal de docent laten zien hoe je met de tot nu toe behandelde stof een eenvoudige programmeerprobleem *stap voor stap* kunt oplossen.

Lees nu hoofdstuk 6 tot en met paragraaf 6.4 van het boek[1].

Tot nu toe heb je functies geschreven die iets afdrukken met behulp van `print` of iets tekenen met behulp van een `turtle`. Als je kijkt naar de ingebouwde functies zoals bijvoorbeeld `math.sin` dan zie je dat deze functie niets afdrukt, maar de sinus van het argument berekent en teruggeeft. Het boek noemt functies die iets teruggeven fruitful[2] functies. Functies die niets teruggeven worden void[3] functies genoemd[4].

Vorige week heb je geleerd dat een goed ontworpen functie in meerdere programma's gebruikt kan worden. Functies die het berekende resultaat meteen afdrukken zijn eigenlijk helemaal niet handig om te gebruiken in verschillende programma's.

Stel je voor dat de functie `math.sin` de sinus van het argument meteen zou afdrukken in plaats van het resultaat terug te geven. Deze functie zou dan slechts in een beperkt aantal gevallen te gebruiken zijn. De versie die het resultaat teruggeeft is in veel meer gevallen te gebruiken. Soms zal de waarde van de sinus afgedrukt moeten worden, door de returnwaarde van de `sin`-functie als argument door te geven aan de `print`-functie:

```
print(math.sin(arg))
```

Maar meestal zal de waarde van de sinus gebruikt worden in een berekening:

```
overstaande_rechthoekszijde = math.sin(hoek) * schuine_zijde
```

2.2.1 In de vorige les heb je bij opdracht 2.1.2 een functie `print_is_schrikkeljaar` gemaakt die afdrukt of een als argument meegegeven jaar een schrikkeljaar is. Schrijf nu een functie `is_schrikkeljaar` die niets meer afdrukt maar die de waarde `True` teruggeeft als het als argument meegegeven jaar een schrikkeljaar is die de waarde `False` teruggeeft als dit niet zo is. Schrijf vervolgens een functie `aantal_dagen_in_jaar` die

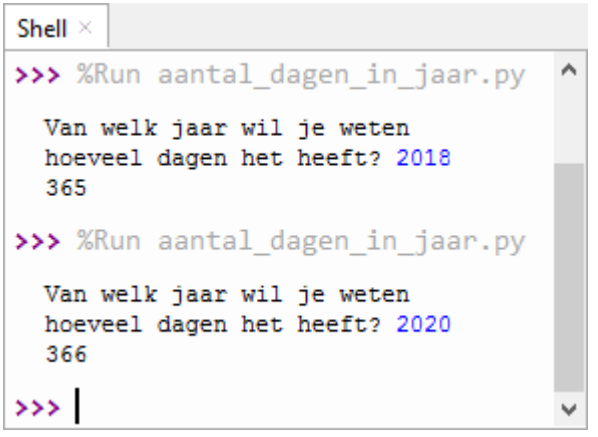
[1] Allen B. Downey. *Think Python: How to Think Like a Computer Scientist*. 2de ed. Green Tea Press, 2016. ISBN: 978-1-4919-3936-9. URL: http://greenteapress.com/wp/think-python-2e/.

[2] Het Engelse woord fruitful betekent vruchtbaar.

[3] Het Engelse woord void betekent leeg of leegte.

[4] Een void-functie geeft, om precies te zijn, wel iets terug namelijk de waarde `None`.

het aantal dagen bepaalt en teruggeeft die het, als argument meegegeven, jaar bevat. Maak daarbij gebruik van de functie `is_schrikkeljaar`[5]. Schrijf een testprogramma dat aan de gebruiker vraagt om een jaar in te voeren en dat afdrukt hoeveel dagen dit jaar bevat, zie figuur 1.



Figuur 1: Twee voorbeelden van het uitvoeren van het testprogramma voor de functie `aantal_dagen_in_jaar`.

2.2.2 Misschien heb je in bij het implementeren van de functie `is_schrikkeljaar` bij opdracht 2.2.1 gebruik gemaakt van een **`if`**-statement. Dat is echter niet nodig want je kan achter een **`return`**-statement ook een booleaanse expressie gebruiken. In dat geval berekent Python de waarde van deze expressie en geeft die terug. Pas, indien nodig, de code van de functie `is_schrikkeljaar` aan zodat geen **`if`**-statement meer wordt gebruikt.

2.2.3 Waarschijnlijk heb je in bij het implementeren van de functie `isSchrikkeljaar` bij opdracht 2.2.1 gebruik gemaakt van lokale variabelen. Dat is echter niet nodig want je kan achter een **`return`**-statement ook een complexe booleaanse expressie gebruiken die in een keer uitrekent of het betreffende jaar een schrikkeljaar is. Pas, indien nodig, de code van de functie `is_schrikkeljaar` aan zodat geen lokale variabelen meer worden gebruikt[6]. Welke versie (met of zonder lokale variabelen) vind je beter? Waarom?

[5] Een schrikkeljaar heeft 366 dagen, elk ander jaar heeft 365 dagen.

[6] Een mogelijk antwoord is gegeven in listing 1. Een ander mogelijk, minder voor de hand liggend, antwoord is gegeven in listing 2.

2.2.4 Waarschijnlijk heb je in bij het implementeren van de functie `aantal_dagen_in_jaar` bij opdracht 2.2.1 gebruik gemaakt van een **`if`**-statement. Dat is echter niet nodig want je kan het aantal dagen ook uitrekenen met een optelling als je weet dat de waarde **`False`** bij een integerberekening automatisch wordt omgezet in de waarde 0 en dat de waarde **`True`** bij een integerberekening automatisch wordt omgezet in de waarde 1. Pas de code van de functie `aantalDagenInJaar` aan zodat geen **`if`**-statement meer wordt gebruikt[7]. Welke versie (met of zonder **`if`**-statement) vind je beter? Waarom?

2.2.5 Waarschijnlijk heb je in bij het implementeren van het testprogramma bij opdracht 2.2.1 gebruik gemaakt van meerdere statements. Dat is echter niet nodig want je kan volstaan met één print statement. Pas de code van het testprogramma aan zodat slechts één statement wordt gebruikt[8]. Welke versie (met één statement of met meerdere statements) vind je beter? Waarom?

```
def is_schrikkeljaar(jaar):
    return jaar % 4 == 0 and jaar % 100 != 0 or jaar % 400 == 0
```

Listing 1: Een compacte versie van de functie `is_schrikkeljaar`, zie `aantal_dagen_in_jaar_compact.py`.

```
def is_schrikkeljaar(jaar):
    return (jaar % 4 == 0) ^ (jaar % 100 == 0) ^ (jaar % 400 == 0)
```

Listing 2: Een andere compacte versie van de functie `is_schrikkeljaar`, zie `aantal_dagen_in_jaar_compact_met_exor.py`. Deze functie maakt gebruik van de exor-operatie die in Python gecodeerd wordt als `^`.

```
def aantal_dagen_in_jaar(jaar):
    return 365 + is_schrikkeljaar(jaar)
```

Listing 3: Een compacte versie van de functie `aantal_dagen_in_jaar`, zie `aantal_dagen_in_jaar_compact.py`.

In de voorgaande drie opdrachten heb je gezien dat je berekeningen en bewerkingen in Python heel compact kunt coderen, zie listings 1 tot en met 4. Dit heeft echter *niet* de voorkeur omdat:

- compacte code lastiger te lezen en te begrijpen is;

[7] Een mogelijk antwoord is gegeven in listing 3.

[8] Een mogelijk antwoord is gegeven in listing 4.

```
print(aantal_dagen_in_jaar(int(input('Van welk jaar wil je ↩
 ↪ weten\nhoeveel dagen het heeft? '))))
```

Listing 4: Een compacte versie van het testprogramma voor de functie `aantal_dagen_in_jaar`, zie `aantal_dagen_in_jaar_compact.py`.

- compacte code lastiger te debuggen is;
- compacte code in de meeste gevallen niet sneller uitgevoerd wordt.

Het is dus voor een beginnende programmeur *niet* aan te raden om zelf compacte code te schrijven. Wel moet je deze code kunnen lezen en begrijpen omdat je, in de praktijk, vaak code die door andere programmeurs is geschreven moet gebruiken of aanpassen[9].

Lees nu paragraaf 6.5, 6.6 en 6.7 van het boek.

2.2.6 Neem de functie `factorial` gegeven in paragraaf 6.5 van het boek over. Schrijf een testprogramma dat een geheel getal n inleest en de faculteit van dit getal $n!$ afdrukt. Test de functie door 100! uit te rekenen. Zorg dat je goed begrijpt hoe de recursieve functie `factorial` werkt door het berekenen van 3! stap voor stap uit te voeren door eerst op het de debug-knop te klikken of door op Ctrl + F5 te drukken en daarna steeds op de step-into-knop te klikken of op F7 te drukken.

2.2.7 In paragraaf 6.6 wordt gesproken over de 'leap of faith'[10]. Hoe neem je bij het debuggen met behulp van Thonny zo'n 'leap of faith'?

Bij het vereenvoudigen van een breuk verkrijg je de grootst mogelijke vereenvoudiging door de teller en de noemer te delen door de grootste gemene deler van de teller en de noemer. De grootste gemene[11] deler van twee gehele getallen is het grootste positieve gehele getal, waar allebei deze getallen door gedeeld kunnen worden zonder dat er een rest overblijft. Bijvoorbeeld: de grootste gemene deler van 75 en 105 is 15. De grootste gemene deler van a en b wordt vaak aangeduid met ggd(a, b)

[9] Elke programmeertaal, dus ook Python, heeft bepaalde taalconstructies waarmee je bepaalde bewerkingen compact en *elegant* kan coderen. Deze taalconstructies moet je *wel* aanleren en gaan gebruiken, omdat deze constructies voor een ervaren programmeur beter te lezen en te begrijpen zijn dan de minder elegante varianten. Zie eventueel de volgende discussie.

[10] Letterlijk vertaald: sprong van vertrouwen.

[11] Het woord gemeen is in dit geval gebruikt in de betekenis van gemeenschappelijk.

Al ongeveer 300 jaar voor onze jaartelling heeft de Griekse wiskundige Euclid[12] een algoritme beschreven waarmee de grootste gemene deler bepaald kan worden. Dit algoritme kan recursief gedefinieerd worden zoals weergegeven in vergelijking (1).

$$\text{ggd}(a,b) = \begin{cases} a & \text{als } a = b \\ \text{ggd}(a-b,b) & \text{als } a > b \\ \text{ggd}(a,b-a) & \text{als } a < b \end{cases} \tag{1}$$

2.2.8 Schrijf een recursieve functie ggd die de grootste gemene deler berekent van twee parameters a en b. Schrijf ook een testprogramma om de functie te testen.

Lees nu paragraaf 6.8 van het boek.

2.2.9 Wat gebeurt er als je de functie ggd aanroept met twee argumenten met de waarde `0`? Is dit antwoord correct? Wat gebeurt er als je de functie ggd aanroept met een argument met een negatieve waarde? Wat gebeurt er als je de functie ggd aanroept met een argument met een floating point waarde, bijvoorbeeld `ggd(4, 3.1)`? Wat gebeurt er als je de functie ggd aanroept met een argument van het type string, bijvoorbeeld `ggd('hallo', 'daar')`?

2.2.10 Voeg code toe aan de functie ggd die ervoor zorgt dat er een duidelijke foutmelding wordt gegeven als deze functie met onjuiste argumenten wordt aangeroepen.

2.2.11 Is het volgens jou nodig om in een functie te controleren of de argumenten geldig zijn? Of is dit volgens jou de verantwoordelijkheid van degene die de functie aanroept? Wat gebeurt er als je de functie `math.sqrt` met een ongeldig argument aanroept?

In paragraaf 6.9 van het boek wordt uitgelegd dat je een functie kunt debuggen door `print`-statements toe te voegen. Het is echter veel gemakkelijker om een goede debugger, zoals Thonny, te gebruiken waarmee je de functie stap voor stap kunt doorlopen en alle variabelen kunt bekijken.

[12] Zie eventueel: https://en.wikipedia.org/wiki/Euclidean_algorithm

Paragraaf 6.10 van het boek definieert de verschillende (vak)termen die in hoofdstuk 6 gebruikt worden. Het kan handig zijn om deze paragraaf te raadplegen als je niet meer weet wat met een bepaalde (vak)term bedoeld wordt.

Vorige week heb je geleerd om recursie te gebruiken om bepaalde code herhaald uit te laten voeren. In deze les ga je het **while**-statement leren kennen en gebruiken waarmee je bepaalde code herhaald uit kunt laten voeren. Dit is een, vaak eenvoudiger[13], alternatief voor recursie en wordt iteratie genoemd. De code wordt dan uitgevoerd zolang aan een bepaalde voorwaarde wordt voldaan. Op deze manier is het Python-programma in staat om een bepaald deel van de code te herhalen.

Lees nu hoofdstuk 7 tot en met paragraaf 7.4 van het boek.

Bij opdracht 2.2.8 heb je de functie ggd waarmee de grootste gemene deler van twee getallen kan worden berekend als recursieve functie geïmplementeerd. Het is ook mogelijk om deze functie als iteratieve functie te implementeren met behulp van een **while**-statement.

De iteratieve functie om de grootste gemene deler van twee positieve gehele getallen a en b te bepalen bestaat uit de volgende stappen:

- herhaal de volgende code zolang $a \neq b$:
  - als $a > b$, dan $a = a - b$
  - anders $b = b - a$
- return a

2.2.12 Schrijf een iteratieve functie ggd die de grootste gemene deler berekent van twee parameters a en b. Schrijf ook een testprogramma om de functie te testen.

2.2.13 Gebruik het programma dat je bij opdracht 2.2.12 hebt geschreven om de grootste gemene deler van 1 en 1000 uit te rekenen. Gebruik daarna het programma dat je bij opdracht 2.2.8 hebt geschreven om de grootste gemene deler van 1 en 1000 uit te rekenen. Wat gebeurt er? Welke conclusie trek je daaruit?

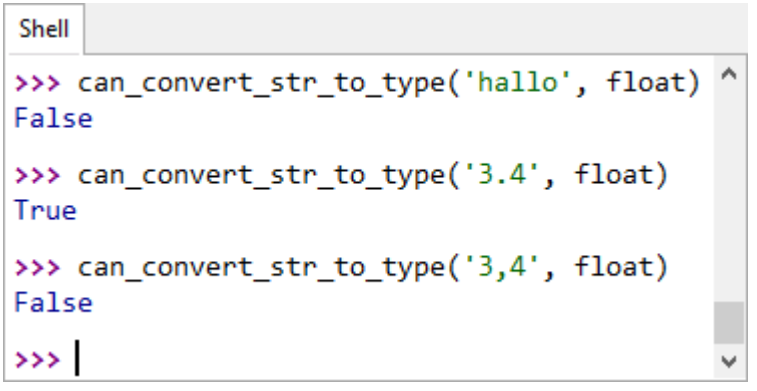
2.2.14 In een Python programma moet je regelmatig een geheel getal n inlezen dat voldoet aan de voorwaarde $min \leq n \leq max$ voor bepaalde waarden van min en max. Als je bijvoorbeeld een toetsresultaat r als geheel getal wilt inlezen dan moet gelden

[13] Vaak, maar niet altijd. Het oplossen van Sudoku puzzels is eenvoudiger met behulp van recursie.

$1 \leq r \leq 10$. Voordat iets wordt ingelezen, wordt meestal een tekst afgedrukt die de gebruiker vraagt om de gewenste data in te voeren. Zo'n tekst wordt een prompt[14] genoemd. Schrijf een functie `input_int` die een integer inleest en teruggeeft. Deze functie moet drie parameters hebben: `prompt`, `min` en `max`. De functie moet eerst de meegegeven `prompt` afdrukken en vervolgens de door de gebruiker ingetypte invoer inlezen. Vervolgens moet de functie controleren of de door de gebruiker ingetypte invoer een geheel getal is[15] en of dat dit getal $\geq$ `min` en $\leq$ `max` is. Zolang dit niet zo is, moet de gebruiker om nieuwe invoer worden gevraagd en moet deze invoer opnieuw worden gecontroleerd. Test deze functie door twee toetsresultaten r_1 en r_2 als gehele getallen in te lezen en het gemiddelde van deze resultaten $(r_1 + r_2)/2$ af te drukken. Er moet daarbij gecontroleerd worden dat $1 \leq r_1 \leq 10$ en dat $1 \leq r_2 \leq 10$.

```
def can_convert_str_to_type(str, type):
    try:
        type(str)
    except:
        return False
    return True
```

Listing 5: De functie `can_convert_str_to_type.py`.



Figuur 2: Drie voorbeelden van het uitvoeren van de functie `can_convert_str_to_type`.

[14] Zie eventueel: https://www.vandale.nl/gratis-woordenboek/nederlands/betekenis/prompt.

[15] Vreemd genoeg heeft Python geen ingebouwde functie om te controleren of een ingelezen variabele van het type `str` geconverteerd kan worden naar een bepaald ander type, zoals `int` of `float`. Daarom hebben wij zo'n functie voor je gemaakt, zie listing 5. Experimenteer zelf met deze functie in de Python Shell zodat je begrijpt hoe je deze functie kunt aanroepen, zie figuur 2.

2.2.15 Misschien heb je bij opdracht 2.2.14 de functie `input_int` wel gedefinieerd zoals weergegeven in listing 6. Deze implementatie is echter niet ideaal omdat de regel `i = input(prompt)` twee keer in de functie wordt gebruikt. Dit maakt het programma slecht onderhoudbaar. Stel dat je na de `prompt` nog een extra spatie wilt afdrukken dan moet deze regel *twee* keer gewijzigd worden in `i = input(prompt + ' ')`. Het zou dus beter zijn als de betreffende regel slechts één keer in de functie voorkomt. Dit kun je bereiken door een **`while True`**:-statement te combineren met een **`if`** `...`: **`break`**-statement. Implementeer de functie `input_int` opnieuw waarbij je ervoor zorgt dat elke regel van de functie uniek is.

```
def input_int(prompt, min, max):
    i = input(prompt)
    while not can_convert_str_to_type(i, int) or int(i) < min or ↩
 ↪ int(i) > max:
        i = input(prompt)
    return int(i)
```

Listing 6: Een mogelijke implementatie van de functie `input_int_while.py`.

Vorige week heb je ook het **`for`**-statement leren kennen waarmee je ook bepaalde code herhaald uit kunt laten voeren. Wanneer moet je nu het **`for`**-statement gebruiken en wanneer het **`while`**-statement? Als het aantal herhalingen bekend is op het moment dat de herhaling start, dan is een **`for`**-statement de beste keuze, maar als het aantal herhalingen onbekend is op het moment dat de herhaling start, dan is een **`while`**-statement de beste keuze.

Bij opdracht 2.2.6 heb je de functie `factorial` als recursieve functie geïmplementeerd. Het is ook mogelijk (en naar mijn mening eenvoudiger) om deze functie als iteratieve functie te implementeren met behulp van een **`for`**-statement.

2.2.16 Schrijf een functie `factorial` met een parameter n die $n!$ (n- faculteit) berekent en teruggeeft. Gebruik een **`for`**-statement om $n!$ te berekenen. Bedenk dat $n!$ gedefinieerd is als $1 \times 2 \times 3 \times \cdots \times n$. Schrijf een testprogramma dat een geheel getal n inleest en de faculteit van dit getal afdrukt.

2.2.17 Gebruik het programma dat je bij opdracht 2.2.16 hebt geschreven om 1000! uit te rekenen. Gebruik daarna het programma dat je bij opdracht 2.2.6 hebt geschreven om 1000! te berekenen. Wat gebeurt er? Welke conclusie trek je daaruit?

2.2.18 Schrijf een functie `factorial` met een parameter n die $n!$ (n- faculteit) berekent en teruggeeft. Gebruik een **`while`**-statement om $n!$ te berekenen. Schrijf een testprogramma dat een geheel getal n inleest en de faculteit van dit getal afdrukt.

2.2.19 Vergelijk het programma dat je bij opdracht 2.2.16 hebt geschreven met het programma dat je bij opdracht 2.2.18 hebt geschreven. Welk van deze twee programma's vind je de beste implementatie van de functie `factorial`? Waarom? Ben je het eens met de bovenstaande bewering: 'als het aantal herhalingen bekend is op het moment dat de herhaling start, dan is een **`for`**-statement de beste keuze, maar als het aantal herhalingen onbekend is op het moment dat de herhaling start, dan is een **`while`**-statement de beste keuze'?

Lees nu paragraaf 7.5 en 7.6 van het boek.

2.2.20 Schrijf een functie `my_sqrt` die de wortel van een als argument meegegeven positief getal benadert met behulp van de methode van Newton die besproken wordt in het boek. Zorg ervoor dat de iteratieve benadering stopt als het verschil tussen twee opeenvolgende benaderingen kleiner is dan 1×10^{-10}.

Paragraaf 7.8 van het boek definieert de verschillende (vak)termen die in hoofdstuk 7 gebruikt worden. Het kan handig zijn om deze paragraaf te raadplegen als je niet meer weet wat met een bepaalde (vak)term bedoeld wordt.

Hier eindigen de verplichte opdrachten van week 2 les 2. Er volgen nu nog twee gedeelten:

- oefening, in dit gedeelte vind je extra oefeningen die je helpen om de leerstof van deze les beter te onthouden;
- verdieping, in dit gedeelte vind je uitdagende, verdiepende opdrachten.

Oefening

Veel van de onderstaande oefeningen (met de bijbehorende *uitwerking*) zijn ook beschikbaar op Anki flashcards zodat je ze regelmatig kunt herhalen. Deze stok kun je hier downloaden: EMS10 Week 2 Les 2.apkg.

2.2.21 Schrijf een functie genaamd `is_even` die **`True`** teruggeeft als de parameter `getal` deelbaar is door twee en **`False`** teruggeeft als dit niet zo is. Deze functie moet als volgt gebruikt kunnen worden:

```
invoer = int(input('Geef een getal: '))
if is_even(invoer) and is_even(invoer // 2):
    print(invoer, 'is deelbaar door vier')
```

2.2.22 Schrijf een functie genaamd `inhoud_kubus` die de inhoud van een kubus berekent en teruggeeft. De lengte van een ribbe[16] wordt als argument meegegeven aan de parameter `lengte_ribbe`. Deze functie moet als volgt gebruikt kunnen worden:

```
print('De inhoud van een kubus met ribben met een lengte 2 ↩
 ↪ is', inhoud_kubus(2))
```

Het correcte antwoord is dan:

```
De inhoud van een kubus met ribben met een lengte 2 is 8
```

2.2.23 Schrijf een functie genaamd `inhoud_bol` die de inhoud V van een bol berekent en teruggeeft. De straal r van de bol wordt als argument meegegeven aan de parameter `straal`. Je kunt gebruik maken van de volgende formule:

$$V = \frac{4}{3}\pi r^3$$

Deze functie moet als volgt gebruikt kunnen worden:

```
print('De inhoud van een bol met een straal van 2 is', ↩
 ↪ inhoud_bol(2))
```

Het correcte antwoord is dan:

```
De inhoud van een bol met een straal van 2 is 33.510321638291124
```

2.2.24 Schrijf een functie genaamd `inhoud_balk` die de inhoud van een balk berekent en teruggeeft. Deze functie heeft de volgende parameters: `lengte`, `breedte` en `hoogte`. Deze functie moet als volgt gebruikt kunnen worden:

[16] Zie eventueel: https://nl.wikipedia.org/wiki/Ribbe.

```
print('De inhoud van een balk van 200 bij 5 bij 4 is', ↩
 ↪ inhoud_balk(200, 5, 4))
```

Het correcte antwoord is dan:

```
De inhoud van een balk van 200 bij 5 bij 4 is 4000
```

2.2.25 Schrijf een functie genaamd `inhoud_kegel` die de inhoud V van een kegel[17] berekent en teruggeeft. De straal r van de kegel wordt als argument meegegeven aan de parameter `straal` en de hoogte h wordt meegegeven aan de parameter `hoogte`. Je kunt gebruik maken van de volgende formule:

$$V = \frac{1}{3}\pi r^2 \cdot h$$

Deze functie moet als volgt gebruikt kunnen worden:

```
print('De inhoud van een kegel met een straal van 2 en een ↩
 ↪ hoogte 10 is', inhoud_kegel(2, 10))
```

Het correcte antwoord is dan:

```
De inhoud van een kegel met een straal van 2 en een hoogte 10 ↩
 ↪ is 41.8879020478639
```

2.2.26 Schrijf een kort programma om de getallen 1 tot en met 10 te printen met behulp van een **`while`**-statement. Schrijf ook een kort programma om de getallen 1 tot en met 10 te printen met behulp van een **`for`**-statement. Welk van de twee heeft je voorkeur en waarom?

2.2.27 Schrijf een kort programma om alle even getallen tussen 10 en 21 te printen met behulp van een **`while`**-statement. Kan dit ook met een **`for`**-statement? Zo ja, hoe?

2.2.28 Schrijf een functie genaamd `is_pos_int` die bepaald of de parameter genaamd `par` een geheel getal (een `int`) bevat en als dat zo is, of deze positief[18] is. Deze functie moet **`True`** of **`False`** teruggeven en moet als volgt gebruikt kunnen worden:

[17] Zie eventueel: https://nl.wikipedia.org/wiki/Kegel_(ruimtelijke_figuur).

[18] In Nederland noemen we een getal positief als het groter dan nul is. In België denken ze daar anders over, zie eventueel: https://nl.wikipedia.org/wiki/Positief_getal.

```
print(is_pos_int(7))
print(is_pos_int(-7))
print(is_pos_int(7.0))
print(is_pos_int("zeven"))
```

Het correcte antwoord is dan:

```
True
False
False
False
```

2.2.29 Schrijf een functie genaamd `som_rekenkundige_rij` die de som van een rekenkundige rij[19] berekent en teruggeeft. De som S_n van een rekenkundige rij van n getallen met een beginwaarde t_1 en een verschil van v is gelijk aan:

$$S_n = t_1 + (t_1 + v) + (t_1 + 2v) + \cdots + (t_1 + (n-1) \cdot v)$$

Deze functie heeft de volgende parameters: `beginwaarde`, `verschil` en `aantal`. Deze functie moet als volgt gebruikt kunnen worden:

```
print(som_rekenkundige_rij(1, 1, 9))
```

Het correcte antwoord is dan:

```
45
```

De uitwerking van deze opdracht kun je vinden op https://bitbucket.org/HR_ELEKTRO/ems10/wiki/sommen_van_rijen.md.

2.2.30 Schrijf een functie genaamd `som_meetkundige_rij` die de som van een meetkundige rij[20] berekent en teruggeeft. De som S_n van een meetkundige rij van n getallen met een beginwaarde a en een rede r is gelijk aan:

$$S_n = a + a \cdot r + a \cdot r^2 + a \cdot r^3 + \cdots + a \cdot r^{n-1}$$

[19] Zie eventueel: https://nl.wikipedia.org/wiki/Rekenkundige_rij

[20] Zie eventueel: https://nl.wikipedia.org/wiki/Meetkundige_rij

Deze functie heeft de volgende parameters: `beginwaarde`, `rede` en `aantal`. Deze functie moet als volgt gebruikt kunnen worden:

```
print(som_meetkundige_rij(1, 2, 10))
```

Het correcte antwoord is dan:

```
1023
```

De uitwerking van deze opdracht kun je vinden op https://bitbucket.org/HR_ELEKTRO/ems10/wiki/sommen_van_rijen.md.

2.2.31 Schrijf een functie genaamd `som_van_getallen` die de som S van alle natuurlijke getallen van b tot en met e berekent en teruggeeft. Wiskundig kunnen we dit als volgt noteren:

$$S = \sum_{i=b}^{e} i$$

Deze functie heeft de volgende parameters: `beginwaarde` en `eindwaarde`. Deze functie moet als volgt gebruikt kunnen worden:

```
print(som_van_getallen(1, 9))
print(som_van_getallen(9, 1))
print(som_van_getallen(9, 9))
```

Het correcte antwoord is dan:

```
45
0
9
```

Let op! Je kunt deze functie implementeren met een **`for`**-loop, met een **`while`**-loop, zonder herhaling[21] en als recusieve functie. Probeer het alle vier. De uitwerking van deze opdracht kun je vinden op https://bitbucket.org/HR_ELEKTRO/ems10/wiki/sommen_en_producten.md.

2.2.32 Schrijf een functie genaamd `product_van_getallen` die het product P van alle natuurlijke getallen van b tot en met e berekent en teruggeeft. Wiskundig kunnen we dit als

[21] Met behulp van de methode van Gauss, die je hier kunt vinden: https://nl.wikipedia.org/wiki/Rekenkundige_rij.

volgt noteren:

$$P = \prod_{i=b}^{e} i$$

Deze functie heeft de volgende parameters: `beginwaarde` en `eindwaarde`. Deze functie moet als volgt gebruikt kunnen worden:

```
print(product_van_getallen(3, 9))
print(product_van_getallen(9, 3))
print(product_van_getallen(3, 3))
```

Het correcte antwoord is dan:

```
181440
0
3
```

Let op! Je kunt deze functie implementeren met een **for**-loop, met een **while**-loop, zonder herhaling[22] en als recusieve functie. Probeer het alle vier. De uitwerking van deze opdracht kun je vinden op https://bitbucket.org/HR_ELEKTRO/ems10/wiki/sommen_en_producten.md.

Nu volgt nog een voorbeeld waarin het gebruik en het nut van het **break**-statement wordt uitgelegd:

In listing 7 is een programma gegeven waarmee je het bekende raadspelletje hoger/lager kunt spelen.

```
import random
geheim = random.randint(1, 99)
getal = int(input('Raad een getal tussen 0 en 100: '))
while getal != geheim:
    if getal > geheim:
        print('Te hoog')
    else:
        print('Te laag')
    getal = int(input('Raad een getal tussen 0 en 100: '))
print('Je hebt het geraden!')
```

Listing 7: Het programma raden1.py.

[22] Met behulp van de formule: $P = \frac{e!}{(b-1)!}$

Er is gebruik gemaakt van de library `random`[23] om een willekeurig getal tussen 0 en 100 te genereren dat geraden moet worden.

We moeten hier gebruik maken van een **`while`**-statement omdat we niet weten hoeveel pogingen de gebruiker nodig heeft om het geheime getal te raden. De regel onderstaande regel komt nu 2 × in het progamma voor.

```
getal = int(input('Raad een getal tussen 0 en 100: '))
```

Dat is slecht voor de aanpasbaarheid en dus de onderhoudbaarheid van de code. In listing 8 is geprobeerd om dit 'probleem' op te lossen.

```
import random
geheim = random.randint(1, 99)
getal = -1
while getal != geheim:
    getal = int(input('Raad een getal tussen 0 en 100: '))
    if getal == geheim:
        print('Je hebt het geraden!')
    elif getal > geheim:
        print('Te hoog')
    else:
        print('Te laag')
```

Listing 8: Het programma `raden2.py`.

Deze versie is echter een beetje gekunsteld. De waarde waarmee `getal` geïnitialiseerd wordt moet een waarde zijn die ongelijk is aan de waarde van `geheim`. Stel dat we het programma zo aanpassen dat de gebruiker een getal tussen −100 en +100 moet raden. Dan moeten we niet vergeten om `getal` bijvoorbeeld met −101 te initialiseren. Als we dat vergeten werkt het programma goed, behalve als `geheim` toevallig −1 is. We moeten een extra **`if`**-statement toevoegen. De voorwaarde voor deze **`if`** moet tegengesteld zijn aan de voorwaarde van de **`while`**. Bij nader inzien is deze versie dus ook niet erg onderhoudbaar.

Het gebruik van een **`break`**-statement in een oneindige lus[24] lost het 'probleem' eleganter op, zie listing 9.

[23] Zie eventueel: https://docs.python.org/3/library/random.html.

[24] Zie eventueel: https://nl.wikipedia.org/wiki/Oneindige_lus. Het **`break`**-statement zorgt ervoor dat de lus niet echt oneindig uitgevoerd wordt.

```
import random
geheim = random.randint(1, 99)
while True:
    getal = int(input('Raad een getal tussen 0 en 100: '))
    if getal == geheim:
        print('Je hebt het geraden!')
        break
    if getal > geheim:
        print('Te hoog')
    else:
        print('Te laag')
```

Listing 9: Het programma raden3.py.

Paragraaf 6.11 van het boek bevat enkele oefeningen. Deze oefeningen zou je op dit moment moeten kunnen maken. Je kunt ze gebruiken als extra oefenmateriaal. Oefening 6.5 is een (snellere) variant van het algoritme dat je in opdracht 2.2.8 hebt geïmplementeerd. Het is overigens helemaal niet nodig om zelf een functie te definiëren die de grootste gemene deler berekent. De `math` module bevat een functie gcd[25] die de grootste gemene deler (Engels: greatest common divisor) berekent.

Paragraaf 7.9 van het boek bevat enkele oefeningen. Deze oefeningen zou je op dit moment moeten kunnen maken. Je kunt ze gebruiken als extra oefenmateriaal.

Verdieping

Hieronder volgen nog twee uitgebreidere opdrachten waarbij je programma's gaat ontwikkelen die je handig kunt gebruiken als elektrotechnicus. Het doel van deze opdracht is om nogmaals te oefenen met het schrijven en gebruiken van functies. Functies kun je handig gebruiken om een groter probleem op te delen in meerdere kleine problemen. De oplossingen van deze kleinere problemen vormen tezamen de oplossing van het grotere probleem. Als de oplossingen van de kleinere problemen als afzonderlijke functies worden geïmplementeerd dan wordt deze manier van het oplossen van een groter probleem *functionele decompositie* genoemd.

[25] Zie: https://docs.python.org/3/library/math.html#math.gcd.

2.2.33 **A** Schrijf een functie die de n^{de} basiswaarde uit de E12-reeks[26] berekent en teruggeeft. De E12-reeks bestaat uit 12 getallen die op een logaritmische schaal een decade[27] in zo gelijkmatig mogelijke stappen verdelen. De eerste twee gekleurde ringen op een weerstand uit de E12 reeks coderen deze basiswaarde. De n^{de} basiswaarde in deze reeks voor $1 \leq n \leq 12$ kun je berekenen met vergelijking (2). De functie round zal het als argument meegegeven floating-point getal afronden naar de dichtstbijzijnde integer. In Python kun je dit coderen met behulp van de ingebouwde functie `round`[28]. De functie moet -1 teruggeven als de waarde van n buiten de gedefinieerde grenzen ligt.

$$E12\_base(n) = \text{round}\left(10 \times 10^{\frac{n-1}{12}}\right) \tag{2}$$

De eerste regel van de functie is:

```
def calc_E12_base(n):
```

B Schrijf ook een testprogramma om de functie `calc_E12_base` te testen voor $0 \leq n \leq 13$. Als het goed is produceert dit programma de uitvoer die gegeven is in figuur 3.

C Als je de door de functie `calc_E12_base` berekende waarden vergelijkt met de bekende weerstandswaarden uit de E12-reeks, zie figuur 4, dan zie je een aantal verschillen. Deze verschillen zijn in het verleden veroorzaakt door afrondfouten[29]. Schrijf een functie die de n^{de} basiswaarde van de weerstandswaarden uit de E12-reeks bepaalt en teruggeeft voor $1 \leq n \leq 12$. De functie moet -1 teruggeven als de waarde van n buiten de gedefinieerde grenzen ligt. De eerste regel van deze functie is:

```
def E12_base(n):
```

Maak zoveel mogelijk gebruik van de functie `calc_E12_base`.

D Schrijf ook een testprogramma om de functie `E12_base` te testen voor $0 \leq n \leq 13$. Als het goed is produceert dit programma de uitvoer die gegeven is in figuur 5.

---

[26] Zie https://en.wikipedia.org/wiki/Preferred_number#E_series.

[27] Als je niet weet wat een decade is lees dan: https://nl.wikipedia.org/wiki/Decade_(wis-_en_natuurkunde).

[28] Zie: https://docs.python.org/3/library/functions.html#round.

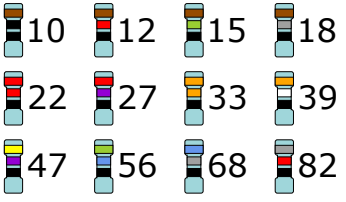
[29] Zie NEN-EN-IEC 60063.

```
Shell
>>> %Run calc_E12_base.py

  calc_E12_base(0) = -1
  calc_E12_base(1) = 10
  calc_E12_base(2) = 12
  calc_E12_base(3) = 15
  calc_E12_base(4) = 18
  calc_E12_base(5) = 22
  calc_E12_base(6) = 26
  calc_E12_base(7) = 32
  calc_E12_base(8) = 38
  calc_E12_base(9) = 46
  calc_E12_base(10) = 56
  calc_E12_base(11) = 68
  calc_E12_base(12) = 83
  calc_E12_base(13) = -1

>>>
```

Figuur 3: De gewenste uitvoer van het testprogramma voor de functie `calc_E12_base`.



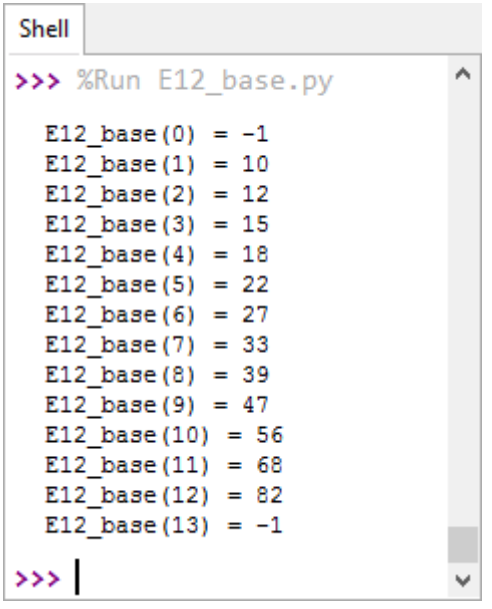
Figuur 4: De basiswaarden uit de E12-reeks. Bron Wikipedia

E De derde kleurenring op een weerstand uit de E12-reeks codeert de exponent p. De weestandwaarde kun je berekenen door de basiswaarde die gecodeerd wordt door de eerste twee ringen te vermenigvuldigen met 10^p. Schrijf een functie die de waarde van de n^{de} weerstand uit de E12-reeks berekent en teruggeeft voor een gegeven exponent $-3 \leq p \leq 9$.

Maak daarbij gebruik van de functie `E12_base` die je hebt geschreven bij deelopdracht C. De functie moet -1 teruggeven als de waarde van n of p buiten de gedefinieerde grenzen ligt.

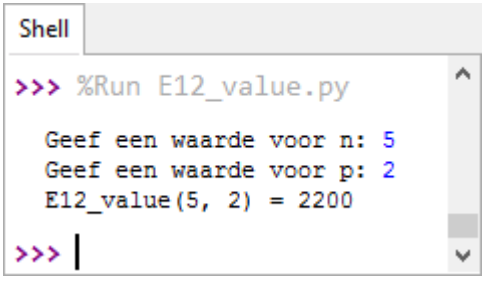
De eerste regel van deze functie is:

```
def E12_value(n, p):
```



Figuur 5: De gewenste uitvoer van het testprogramma voor de functie `E12_base`.

F Schrijf ook een testprogramma om de functie `E12_value` te testen. Een mogelijke uitvoer van dit programma is gegeven in figuur 6.

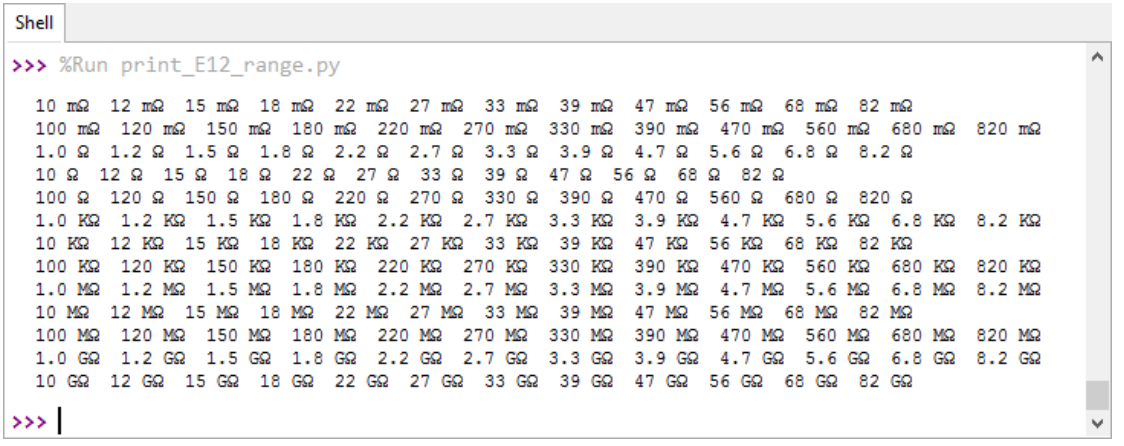


Figuur 6: Een mogelijke uitvoer van het testprogramma voor de functie `E12_value`.

G Schrijf een functie die alle mogelijke weerstandswaarden uit de E12-reeks afdrukt. De gewenste uitvoer is weergegeven in figuur 7. Maak gebruik van de functie die je hebt geschreven bij deelopdracht E.

Schrijf eerst een aparte functie `print_E12_value` die een weerstandswaarde kan afdrukken indien nodig gevolgd door een spatie en een SI-voorvoegsel (m, k, M,

G) en afgesloten met twee spaties. Tip: je kan hierbij handig gebruik maken van de functie `math.log10`.



Figuur 7: De gewenste uitvoer van de functie `print_E12_range`.

2.2.34 Ontwerp, implementeer en test nu een programma dat de kleurcode van een weerstand uit de E12-reeks inleest en de waarde van deze weerstand afdrukt. Maak daarbij zoveel mogelijk gebruik van de functies die je bij opdracht 2.2.33 hebt geschreven. Het programma hoeft alleen de kleuren van de eerste 3 ringen in te lezen. We gaan ervan uit dat de tolerantie 10% is, maar dat wordt niet gecontroleerd door het programma. Om de kleuren in te voeren gebruiken we de kleurcodes die gedefinieerd zijn in IEC 60062:2016[30]. Bijvoorbeeld de kleur rood wordt gecodeerd als `RD`. Het program moet alleen geldige kleurcodes accepteren en alleen kleurcombinaties die geldig zijn in de E12-reeks. Bij ongeldige invoer moet opnieuw om invoer gevraagd worden. De geldige waarden uit de E12 reeks heb je bij opdracht 2.2.33 afgedrukt. De gewenste uitvoer is weergegeven in figuur 8.

[30] Zie: https://en.wikipedia.org/wiki/Electronic_color_code#Resistor_color-coding.

Figuur 8: De gewenste uitvoer van het programma `colors_to_E12_value.py`.