Leereenheid 10

Imperatief programmeren

INTRODUCTIE

Voor een informaticus vormt een computer een essentieel stuk gereedschap. In deze leereenheid leert u de beginselen van het programmeren, oftewel, hoe u de computer voor u laat werken. De uitdaging hierbij is dat computers ontzettend precies zijn – alles wat niet aan de computer is uitgelegd, wordt dan ook niet gedaan.

In het vak Inleiding informatica programmeren we in de taal Python. Python is een relatief eenvoudige taal die de programmeur in staat stelt om snel programma's te schetsen. Het is een van de populairste programmeertalen wereldwijd en wordt ondermeer gebruikt bij YouTube en Google.

U zult in deze leereenheid ook enkele voorbeelden in andere programmeertalen zien. Deze dienen enkel ter illustratie en zullen dientengevolge niet in detail worden uitgelegd.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u:

- eenvoudige Python-programma's kunt lezen
- Python-programma's kunt uitvoeren
- zelf eenvoudige Python-programma's kunt schrijven
- commentaar kunt opnemen in een Python-programma
- de betekenis van de volgende kernbegrippen kunt geven: variabele, type, argument, parameter, herhaling, keuze, functie
- de scope van een variabele kunt bepalen
- functies in een Python-programma kunt definiëren en aanroepen
- het onderscheid tussen imperatief, objectgeörienteerd, functioneel, en logisch programmeren kunt herkennen.

Studeeraanwijzingen

Deze leereenheid is geschreven vanuit het idee dat u actief met de materie aan de slag gaat. Het is dan ook de bedoeling dat u de programma's die u tegenkomt en in opgaven schrijft, ook echt uitvoert. U kunt Python downloaden van `http://python.org` en installeren. Ook kan Python vanaf die website direct online gebruikt worden. Voor de oefeningen maakt het niet uit of u het online gebruikt of op uw computer installeert. Installeren levert wel gemak op: u kunt programma's bewaren en vaker uitvoeren door middel van `python bestandsnaam` op de commandline. Ook hoeft u niet telkens alles in te tikken. Let op: programma's lopen op de kleinste typefoutjes al vast. Als u een foutmelding van Python terugkrijgt, kijk dan of er een typefoutje is gemaakt in de regel genoemd in de foutmelding. Mocht u er niet uitkomen, zoek dan op het internet naar de foutmelding.

Deze leereenheid is gericht om u bekend te maken met programmeren, waardoor er nauwelijks aandacht aan achtergronden wordt besteed. Mocht u benieuwd zijn naar het hoe en waarom, zoek ook dan op internet. Bijvoorbeeld: hoe ver moet u inspringen voor een blok? Zoeken op 'python how much indent for a block' leidt u naar verschillende pagina's die hier achtergronden bij geven.

Programmeeropdrachten hebben bijna altijd meer dan één goede uitwerking. In de terugkoppeling tonen we steeds één mogelijkheid. Hebt u een andere oplossing die ook werkt, vergelijk die dan met de uitwerking in de terugkoppeling; in veel gevallen zal uw oplossing net zo goed zijn. U vindt de broncode van de voorbeelden in de online leeromgeving.

De studielast van deze leereenheid bedraagt ongeveer 9 uur.

LEERKERN

1 Inleiding

1.1 WAT IS PROGRAMMEREN?

Een computer doet niets uit zichzelf. Voor iedere taak die door een computer wordt uitgevoerd, is een programma nodig: een voorschrift dat door de computer kan worden verwerkt, waarin precies en ondubbelzinnig is vastgelegd wat de computer moet doen.

Machinetaal

De processor van een computer kan een aantal opdrachten uitvoeren. Deze opdrachten worden aangeleverd in *machinetaal* – de interne taal van de processor. Een programma in machinetaal is niets anders dan een (lange) reeks enen en nullen. Dit is erg onleesbaar voor mensen. De eerste taal waarin mensen hun opdrachten naar computers communiceerden, was een afgeleide hiervan: *assembly*. Assembly blijft erg dicht bij machinecode, maar gebruikt woorden in plaats van cijferreeksen.

Assembly

Inmiddels hoeven we als programmeur niet eens meer te weten hoe de machinetaal er uitziet van de computer die we gebruiken. We formuleren ons programma in een geschikte algemene programmeertaal en laten het aan de computer over daar machinetaal van te maken. Zo'n hogere programmeertaal is dus, in tegenstelling tot de machinetaal, onafhankelijk van welke processor dan ook.

1.2 PYTHON GEBRUIKEN

Python kan op twee manieren gebruikt worden:

Script mode

- *Script mode*: u schrijft het programma in een editor en voert het vervolgens uit (`python filename.py`), of

Interactive mode

- *Interactive mode*: u start een interactieve Python omgeving (op uw computer of online), tikt daar een instructie in en ziet meteen het resultaat van deze instructie.

De interactieve modus is handig om kleine stukjes programma uit te testen. In deze modus krijgt u na iedere instructie een reactie van Python. Daarmee kan deze modus handig gebruikt worden bij het opsporen van fouten.

Voor korte programma's (een paar regels) kunt u prima de interactieve modus gebruiken. Al snel wordt het handiger om bestanden apart op te slaan en de script-modus te gebruiken. In het vervolg van deze leereenheid maken we hier verder geen onderscheid in. We gaan ervan uit dat u uw programma's opslaat en dus eenvoudig kunt hergebruiken en aanpassen. Alle programma's in deze leereenheid zijn getest in de interactieve modus.

2 Simpele programma's

2.1 HELLO WORLD

Het typische eerste programma van een beginnend programmeur is het op het scherm zetten van de tekst 'Hello world'. In Python:

```
1 print("Hello world.")
```

Expressie

U kunt ook berekeningen (*expressies*) laten uitvoeren in een instructie, bijvoorbeeld:

```
1 print(12345 + 678)
2 print(987 * 654 - 123)
3 print("hello" + " " + "world.")
4 print("hello" * 5)
```

String

Het stukje tussen de dubbele quotes is een voorbeeld van een *string*. Een string is een rijtje van cijfers, letters en andere tekens. In het voorbeeld zien we dat strings kunnen worden opgeteld (regel 3) en zelfs kunnen worden vermenigvuldigd (regel 4). Deze bewerkingen hebben de voor de hand liggende resultaten, wat u zult zien als u dit uitprobeert.

In deze leereenheid worden dubbele quotes (") gebruikt voor strings.
Mag een single quote (') ook?

Als u op internet zoekt naar 'python single quote' zult u snel zien dat het niet uitmaakt of u ' of " gebruikt, zolang u maar consistent bent.

Een volgende stap is een persoonlijke boodschap. In plaats van 'hello world' schrijft onderstaand programma de naam van de gebruiker (in dit geval 'De Jong') op het scherm.

```
1 naam = "De Jong"
2 print("Goedemorgen " + naam)
```

2.2 VARIABELEN EN TYPES

Variabele

In het vorige voorbeeld was `naam` een *variabele*. Een variabele is een stukje geheugen waarin de programmeur data kan opslaan. In het voorbeeld was dat een rijtje letters, maar een variabele kan ook een getal opslaan. Probeer bijvoorbeeld:

```
1 naam = 42
2 print(naam)
3 naam = 13 / 37
4 print(naam)
```

Probeer nu eens het volgende:

```
1 naam = 42
2 print("Goedemorgen " + naam)
```

Wat gebeurt er? En wat gebeurt er als u `naam = "42"` gebruikt?

Type
int
float
bool
str
String
lijst
list

We zien dat Python een onderscheid maakt tussen quotes en getallen. Dit komt doordat deze twee zaken een verschillend *type* hebben, ofwel de klasse van de data. Zo kent Python onder andere *int* voor (een deelverzameling van) $\mathbb{Z}$, *float* voor (een deelverzameling van) $\mathbb{R}$, *bool* voor de waarden `True` en `False`, *str* voor *strings* en *lijsten* voor rijtjes data. Een voorbeeld van een rij data is `[1,2, "hoi", -1.73, ['a']]`, waarin de eerste twee elementen type int hebben, het derde element het type string heeft, het vierde element type float heeft en het laatste element type *list*, omdat het zelf een lijst is (met één element).

int omzetten naar string: `str()`

U kunt ook het type van een stuk data omzetten. Zo kunt u met behulp van de ingebouwde functie `str()` een waarde omzetten van type int naar type string. De vorige opdracht hadden we dus als volgt kunnen uitvoeren:

```
1 naam = 42
2 print("Goedemorgen " + str(naam))
```

Variabelen zijn handig om programma's generiek te maken. Het onderstaande fragment rekent de oppervlakte uit van een vierkant met zijde n, namelijk: $n \cdot n$.

```
1 # berekent oppervlakte van vierkant met zijden n
2 def opp_vierkant(n):
3     return n * n
```

Toelichting

In regel 1 zien we commentaar. In Python geeft # aan dat de rest van de regel commentaar is. Commentaar wordt gebruikt om te documenteren wat het programma doet en wordt daarom genegeerd door de programmeertaal. In regel 2 hebben we een *functie* aangemaakt in regel 2, door middel van `def functienaam():`. Een functie is een stukje programma dat, eventueel op basis van invoer, een bewerking verricht en eventueel een stuk data retourneert aan het aanroepende programma.

Functie

Argument
Parameter

We kunnen de hierboven geschreven functie aanroepen door een waarde op te geven, bijvoorbeeld `print(opp_vierkant(10))` voor $n = 10$. We noemen n een *argument* of *parameter* van de functie. Het resultaat van de functie wordt doorgegeven door middel van `return`.

Hieronder ziet u voorbeelden van functieaanroepen. Wat verwacht u dat er gebeurt bij de volgende aanroepen? Probeer ze uit in interactive mode.

```
print(opp_vierkant(2))
print(opp_vierkant(2.0))
opp_vierkant(2)
opp_vierkant(2.0)
print(opp_vierkant("de jong"))
print(opp_vierkant(2.0)))
print(opp_vierkant(2.0)
```

Python behandelt de waarde `2` anders dan de waarde `2.0`, en behandelt beide anders dan een string. Daarnaast zien we hoe Python reageert als we te weinig of te veel haakjes zetten.

Blok

Merk op dat de inhoud van de functie ingesprongen is. Dit is hoe Python een code *blok* aangeeft: alle code die een gelijke hoeveelheid is ingesprongen, vormt tezamen één blok. Hieronder staat een Python-voorbeeld waarin de blokken expliciet zijn aangegeven.

inspringen

Merk op dat een blok in Python wordt gedefinieerd door *in te springen*. Alle opeenvolgende regels die even ver zijn ingesprongen, horen bij één blok. Als het inspringen veranderd, dan hoort het niet meer bij dit blok. Zorg dat de code altijd op dezelfde manier is ingesprongen; mix dus geen tabs en spaties.

```
# Dit is het eerste blok van programmacode
def sum(x, y):
    # een nieuw blok op niveau 2: blok 2
    def doit():
        # een nieuw blok op niveau 3
        return x+y
    # weer terug in blok 2
    somedata = doit()

    # nog steeds in blok 2
    return somedata

# blok 1
def minus(x,y):
    # een nieuw blok op niveau 2: blok-nieuw
    antwoord = x - y

    # nog steeds in blok-nieuw
    return antwoord
```

NB: in de interactieve modus ziet u na het begin van een blok op het hoofdniveau '...' links. Zolang u in één of meer blokken zit, blijft de prompt '...'. Om het eerste blok af te sluiten en weer terug te keren naar het hoofdniveau moet u een lege regel ingeven. De prompt wordt dan weer '>>>'.

Een functie bestaat uit een definitie met daaronder één blok (dus op één niveau lager dan de definitie zelf). Het einde van het blok is het einde van de functie – ook als `return` eerder staat of weggelaten wordt.

Als we meerdere argumenten in een functie willen opnemen, dan scheiden we deze door komma's, dus `def functienaam(argument1, argument2, ...):`.

OPGAVE 10.1
Maak een functie die de oppervlakte van een m bij n rechthoek berekent, door onderstaand voorbeeld uit te breiden.

```
1 def opp_rechthoek(m,n):
2     ...
```

Als u dit programma in interactieve modus intikt, vergeet dan niet de lege regel om het blok af te sluiten.

We hebben nu een functie om de oppervlakte van een vierkant te berekenen, en een andere functie om de oppervlakte van een rechthoek te berekenen. Maar: een vierkant is een rechthoek. Om precies te zijn: een vierkant is een rechthoek waarvan alle zijden precies even lang zijn. Als we dus een functie hebben om de oppervlakte van een rechthoek te berekenen, kunnen we die gebruiken om de oppervlakte van een vierkant te berekenen.

OPGAVE 10.2
Maak onderstaande functie af.

```
1 # bereken de oppervlakte van een vierkant met zijden
      van lengte n
2 def opp_4kant(n):
3     return opp_rechthoek(...)
```

OPGAVE 10.3
Maak een functie `opp_driehoek` die de oppervlakte van een gelijkbenige driehoek berekent, gegeven de basis b en de hoogte h. (De oppervlakte van zo'n driehoek is basis maal halve hoogte.)

OPGAVE 10.4
Maak een functie `opp_cirkel` die de oppervlakte van een cirkel met straal r berekent. De oppervlakte van een cirkel is πr^2, waarbij π (spreek uit 'pie') ongeveer de waarde 3.14159265358979 . . . heeft.

In de laatste opgave is het antwoord wel correct, maar niet altijd even nuttig. Het kan handig zijn om het antwoord in aantal keer π te weten. Bijvoorbeeld, voor de omtrek van een cirkel ($2\pi r$):

```
1 def omtrek_cirkel(r):
2     return str(2 * r) + " pi"
```

OPGAVE 10.5
Schrijf een functie `inhoud_bol` die de inhoud van een bol met straal r als float retourneert ($\frac{4}{3}\pi r^3$).

OPGAVE 10.6
Schrijf een functie `opp_bol` die de oppervlakte van een bol in aantallen π retourneert ('5 pi'). De oppervlakte van een bol is $4\pi r^2$.

OPGAVE 10.7
Maak een functie `print_tafel` die de vermenigvuldigingstafel van 1 tot 10 van het argument n print met behulp van `print()`, bijv.:

```
1 keer 13 is 13.
2 keer 13 is 26.
...
10 keer 13 is 130.
```

De functie hoeft dus geen waarde te retourneren, maar kan zonder `return` eindigen.

NB: Gebruik de functie `str(int)` om een waarde van type int om te zetten naar een waarde van type string.

3 Herhaling

De laatste opgave werd wat repetitief. De functie ziet er ongeveer als volgt uit:

```
def print_tafel(n):
    print(...)
    print(...)
    print(...)
    print(...)
    print(...)
    print(...)
    print(...)
    print(...)
    print(...)
    print(...)
```

Gelukkig kan dit eenvoudiger. Python heeft (net zoals andere programmeertalen) een aantal manieren om herhaling aan te geven. Een simpel voorbeeld staat hieronder, dat gebruikmaakt van `for i in LIJST`.

```
def print_tafel(n):
   for i in 1,2,3,4,5,6,7,8,9,10:
       print()   # "i keer n is iets"
```

Dit kan zelfs nog korter. Python heeft een instructie `range`, die zo'n lijst genereert. `range` kent drie vormen:
- `range(stop)`: genereert een lijst van 0 tot (maar zonder) `stop`,
- `range(start, stop)`: genereert een lijst van `start` tot (maar zonder) `stop`.
- `range(start, stop, stapgrootte)`: genereert een lijst van `start` tot (maar zonder) `stop`, met stapgrootte `stapgrootte`.

Als voorbeeld staat hieronder een programma dat de getallen van 17 tot en met 33 print. Merk op dat het `stop` argument van `range` dan 34 moet zijn.

```
for i in range(17, 34):
    print(i)
```

Voor een tweede voorbeeld bekijken we de functie faculteit, genoteerd door $n!$. Deze functie berekent het product van alle gehele getallen tot een bepaalde waarde. Zo is $3! = 3 \cdot 2 \cdot 1 = 6$, en $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Dit kunnen we als volgt in Python opschrijven:

```
def faculteit(n):
    j = 1
    for i in range(1, n+1):
        j = j * i
    return j
```

OPGAVE 10.8
Herschrijf de functie `print_tafel` om `range` te gebruiken.

OPGAVE 10.9
De functie `print_tafel(n)` laat nu de vermenigvuldiging van n met alle getallen van 1 tot en met 10 zien. Hoe moet u deze herschreven functie aanpassen om de vermenigvuldiging van n met alle cijfers van 1 tot en met 20 te laten zien?

Door de stapgrootte expliciet op te geven kunnen we van hoog naar laag itereren, zoals in het voorbeeld hieronder:

```
for i in range(33, 16, -1):
   print(i)
```

Let op dat het begin meetelt en het einde niet. Het einde moet dus nog steeds één 'verder' zijn, vandaar dat we hier als eindpunt 16 moeten opgeven (en niet 17).

Ook kunnen we bijvoorbeeld alleen de even nummers printen, zoals in onderstaand voorbeeld.

```
def evennumbers_until(n):
   for i in range(2, n+1, 2):
      print(i)
```

OPGAVE 10.10
Maak een functie `oddnumbers_until` die alle oneven getallen van 1 tot en met het opgegeven argument n print.

3.1 BEWERKINGEN OP STRINGS

Lijst begint bij [0]
String is een lijst

Gegeven een string `naam` willen we de vierde letter bepalen. In Python gaat dat eenvoudig, want Python behandelt strings als lijsten, die beginnen te tellen bij 0. De eerste letter van `naam` is `naam[0]`; de vierde letter is dus `naam[3]`. Omdat Python strings als lijsten van

tekens ziet, kunnen we strings gemakkelijk in een `for`-loop bewerken. In het volgende voorbeeld wordt op iedere regel de volgende letter van de string geprint.

```
def print_lettervoorletter(string):
    for i in string:
        print(i)
```

Substring

Een andere manier is om iedere keer de juiste *substring* (ter lengte één) te printen. Dus we printen `string[0]`, vervolgens `string[1]`, `string[2]`, ..., tot we aan het eind van de string gekomen zijn. Dan houdt `i` de plaats in de string bij, wat betekent dat `i` loopt van 0 (begin van de string) tot het einde van de string, wat wordt gegeven door `len(string)`. Dit levert het volgende programma op:

```
def print_letter4letter(string):
    for i in range(len(string)):
        print(string[i])
```

Ga na dat deze functies hetzelfde doen.

We kunnen ook langere substrings maken door middel van de notatie `string[begin:eind]`. Dit kan op variabelen, maar zelfs direct op strings:

```
"Probeer dit in de interactive mode!"[3:12]
```

Let op! Net als bij `range(begin, eind)` begint de substring bij index `begin`, maar houdt hij op vóór `eind`. Stel, we willen het i^e karakter en zijn twee buren, dan werkt het eerste voorbeeld hieronder dus niet, maar het tweede wel.

```
cijfers = "1234567890"
i = 3 # plaats in string begint bij 0, dus dit is "4"
print(cijfers[i-1:i+1]) # "34"
print(cijfers[i-1:i+2]) # "345"
```

OPGAVE 10.11

Maak een programma `print_per3(string)` dat de invoerstring per drie letters uitprint.

Hint: gebruik hiervoor `range(begin, eind, stapgrootte)`.

`string.index(substring)`

Een andere truc die we met strings kunnen uithalen is zoeken naar een specifieke substring. Dit gebeurt door middel van *`string.index(substring)`*. Deze functie retourneert de index vanaf de plek waar de gezochte substring staat.

Probeer onderstaande instructies eens in de interactieve modus van Python.

```
"Probeer dit in de interactive mode!".index('P')
"Probeer dit in de interactive mode!".index('int')
"Probeer dit in de interactive mode!".index('Helaas')
```

string.lower()
string.upper()
string.isupper()
string.islower()

Er zijn meer van dit soort trucs, zoals *string.lower()* om een string om te zetten naar kleine letters, *string.upper()* om de string in hoofdletters om te zetten, en *string.isupper()* en *string.islower()* die aangeven of de string geen kleine letters respectievelijk geen hoofdletters bevat. (Leestekens worden dus genegeerd.)

Letters verschuiven

We hebben nu genoeg Python-kennis in huis om een leuk voorbeeld te tonen. De bewerking ROT13 is een bewerking van tekst. Deze bewerking vervangt iedere letter door de letter dertien plaatsen verder in het alfabet, waarbij na 'z' "geroteerd" wordt naar 'a' (vandaar de naam). In plaats van `abcdefg` komt er dus `nopqrst` te staan. Dit komt neer op het vervangen van de letter in rij 1 door de letter eronder uit rij 2.

rij 1	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
rij 2	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m

ROT13 ongedaan maken gaat dan ook omgekeerd: pak de letter in rij 2 en schrijf in plaats hiervan de letter in rij 1.

Merk op dat de letter die boven een letter in rij 2 staat (bijv. de 'a' staat boven de 'n' in rij 2) hetzelfde is als de letter die ònder die letter in rij 1 staat: onder de 'n' in rij 1 staat de 'a'. Dit is voor alle letters zo. Dit betekent dat we dus ROT13 ongedaan kunnen maken door de letters in rij 1 op te zoeken en te vervangen door de letter eronder in rij 2. Maar dit is precies de originele ROT13 bewerking – twee keer ROT13 toepassen levert dus weer de originele tekst op. Dit is ook logisch: het alfabet heeft precies 26 letters. Twee keer 13 letters opschuiven is hetzelfde als 26 letters opschuiven.

We kunnen nu een functie `rot13` in Python schrijven die deze bewerking uitvoert. Hiervoor gaan we stap voor stap de invoerstring af, en vervangen we ieder karakter door het karakter 13 plaatsen verder in het alfabet. Uiteraard moeten we dan wel het alfabet erbij hebben. Dit kan bijvoorbeeld als volgt.

```
1  def rot13(text):
2     alphabet='abcdefghijklmnopqrstuvwxyz'
3     retstr = ""
4     for i in text:
5        index = alphabet.index(i)
6        retstr = retstr + alphabet[(index + 13) % 26]
7     return retstr
```

Het werkt als volgt: eerst zetten we het hele alfabet in de variabele `alphabet` (regel 2). We initialiseren de uitvoer van de functie op geen tekst (regel 3). Daarna nemen we iedere letter van de invoertekst apart (regel 4). Voor iedere letter bepalen we zijn positie in het alfabet (regel 5). In regel 6 pakken we vervolgens een andere letter uit `alphabet`. Om precies te zijn: de letter die 13 plaatsen verder staat, waarbij we na de 'z' weer beginnen bij 'a'. Hiervoor rekenen we modulo 26, in Python: '`% 26`'.

Modulo: zie leereenheid 6

OPGAVE 10.12

a Bepaal zonder Python de ROT13 van de tekst 'aaa', 'aap' en 'clown'. Ga na dat het programma `rot13` op dezelfde wijze werkt als wanneer ROT13 handmatig wordt berekend.

b Bepaal met Python de ROT13 van 'kwaliteitseisen', 'presidentskandidaat' en 'viaduct'. Hoe roept u het programma `rot13` programma aan?

OPGAVE 10.13

Maak een functie `rot5` die een string als invoer neemt, en alle tekens 5 plaatsen verschuift. Maak ook een functie `rot5_inverse` die dit weer omdraait. Hoe controleert u of inderdaad geldt dat `rot5_inverse(rot5(string))` hetzelfde oplevert als `string`?

Julius Caesar gebruikte al dit soort letter-verschuif-functies. Dit soort functies worden dan ook vaak 'caesar-functies' genoemd. Zo levert de verschuiving van de tekst 'aaa' met drie plaatsen 'ddd' op. Een verschuiving met vijf plaatsen levert 'fff' op.

OPGAVE 10.14

Maak een functie `caesar(n, string)` die alle karakters van de string *n* plaatsen in het alfabet opschuift. Probeer `caesar(13, alphabet)`. Wat valt u op?

Waarschijnlijk is u opgevallen dat de string `alphabet` in ieder van deze functies weer gebruikt wordt. Nu wordt deze variabele in iedere `rot` functie opnieuw gedefiniëerd. Dat is nodig, omdat variabelen alleen bestaan in het blok waarin ze gedefinieerd worden en de daaronder liggende blokken. Dit heet de *scope van een variabele*. In plaats van deze variabele iedere functie opnieuw te definiëren, kunnen we hem ook één keer definiëren buiten de functies – dus in het blok op het hoogste niveau. Dan is zijn scope dus alle blokken, en kan deze variabele overal gebruikt worden. Dit heet een *globale variabele*. Een *lokale variabele* is een variabele die in een blok dat niet het hoofdblok is, is gedefiniëerd. Parameters zijn per definitie lokale variabelen. Een lokale variabele mag dezelfde naam hebben als een globale variabele. In zo'n geval zijn er twee stukjes geheugen die toevallig in het programma dezelfde naam hebben. Binnen de scope van de lokale variabele verwijst die naam dan naar de lokale variabele. Daarbuiten verwijst die naam naar de globale variabele.

Scope van een variabele

Globale variabele

Lokale variabele

```
# Voorbeeld: lokale en globale variabelen
alphabet='abcdefghijklmnopqrstuvwxyz'

def rot7(string):
   alphabet='hijklmnopqrstuvwxyzabcdefg'
   # nu geldt voor alle onderliggende blokken dat
       alphabet
   # met een 'h' begint. Voor alle bovenliggende
       blokken
   # begint alphabet nog steeds met een 'a'
   for i in alphabet:
      # ook in dit blok begint alphabet met een 'h'

   return string

# dit roept rot7 aan met de string 'abcd...xyz'
print(rot7(alphabet))
```

OPGAVE 10.15

Herschrijf de functies `rot5` en `rot5_inverse` om gebruik te maken van een globale variabele `alphabet`. Heeft het zin om dit ook met `retstr` te doen?

OPGAVE 10.16

Beschouw het programma `rot13` op pagina 190.

a Hoeveel blokken zijn er in dit programma? Geef de regelnummers van ieder blok.

b Wat is de scope van variabele `alphabet`? En van variabele `i`? Geef de bloknamen waarin `i` bestaat als antwoord.

var += iets

Merk overigens op dat `retstr` vaak gebruik wordt in de vorm `retstr = retstr + iets`. Dit is een typische constructie om in een functie de retourwaarde op te bouwen. Python heeft voor deze constructie een afkorting: *`var += iets`*. Dit betekent hetzelfde als `var = var + iets`.

4 Keuzes

In het `rot13`-programma dat we tot nu toe hebben gemaakt, zijn alleen kleine letters zijn toegestaan. Als we een string met een hoofdletter, een spatie, een leesteken, of een cijfer invoeren, dan geeft het programma een foutmelding.

Tijd om dit te verbeteren! Onderstaande versie laat alles wat geen kleine letter is ongewijzigd.

```
# ROT13 met leestekens
def rot13(text):
   alphabet='abcdefghijklmnopqrstuvwxyz'
   retstr = ""
   for i in text:
      if i in alphabet:
        index = alphabet.index(i)
        retstr += alphabet[(index + 13) % 26]
      else:
        retstr += i
   return retstr
```

if conditie:

else:

In regel 6 maken we een keuze met behulp van *if conditie:*. We testen of het huidige karakter in onze lijst met letters voorkomt. Zo ja, dan passen we de oude `rot13`-verschuiving toe (regels 7 en 8). Zo nee, aangegeven door *else:* in regel 9, dan nemen we het karakter ongewijzigd over in de uitvoer (regel 10). Merk op dat += wordt gebruikt (regel 8, 10).

Vergelijkingsoperatoren

in

<

<=

>

>=

==

!=

elif

Er zijn verschillende operatoren om twee zaken te vergelijken. Hierboven zagen we al *in* om te bepalen of iets in een lijst voorkomt: "*var* `in` *lijst*". Maar ook kunnen we twee getallen vergelijken met *<* (kleiner dan), *<=* (ten hoogste), *>* (groter dan), *>=* (ten minste), *==* (gelijkheid) en *!=* (ongelijkheid). Deze laatste twee (== en !=) werken ook voor strings. Zie ook het volgende voorbeeld, waarin we *elif* gebruiken wat 'else if' betekent:

```
1  if 10 <= 5:
2    # False, 10 is niet kleiner dan of gelijk aan 5
3    print('niet waar!')
4  elif 10 == 5:
5    # False
6    print('10 is niet hetzelfde als 5')
7  elif 10 >= 5:
8    # True, 10 is groter dan 5
9    print('Klopt, 10 is groter dan 5')
10
11 if "aa" == "bb":
12   # False
13   print('fout! aa is niet hetzelfde als bb')
14 elif "aa" != "bb":
15   # True
16   print('klopt, aa en bb verschillen')
```

OPGAVE 10.17

Pas uw programma `caesar(n, string)` aan, zodat het alle tekens die niet in `alphabet` zitten, ongewijzigd in de uitvoer overneemt.

= voor toekenning
== voor vergelijking

Let op: als we een variabele een waarde willen geven, dan gebruiken we één gelijkteken: `variabele = expressie`, waarmee we aan `variabele` de waarde van de expressie `expressie` toekennen. Als we een keuze willen maken afhankelijk van een variabele, gebruiken we twee gelijktekens:
`if variabele == expressie`. Dus:
– waarde van variabele veranderen: gebruik =
– waarde van variabele vergelijken: gebruik ==

OPGAVE 10.18

Schrijf een programma `div3(n)` dat de bool `True` retourneert als n deelbaar is door drie en de bool `False` als n niet deelbaar is door drie.

Een int n is deelbaar door drie als n mod $3 = 0$ (in Python: `n % 3` is gelijk aan 0).

De volgende vraag is dan uiteraard: kunnen we ook hoofdletters meenemen? Als we in het `rot13`-programma regel 6 vervangen door `if i.lower() in alphabet:` dan nemen we zowel hoofd- als kleine letters mee bij het omzetten. Wel moeten we dan ook regel 7 aanpassen, zodat we niet per ongeluk op zoek gaan naar een hoofdletter.

Nu moeten we bij het omzetten nog zorgen dat we hoofdletters naar hoofdletters omzetten, en kleine letters naar kleine letters. Met `i.islower()` kunnen we testen of `i` geen hoofdletters bevat. Zo ja, dan werkt de bestaande omzetting (regel 8). Zo nee, dan moet het omgezette karakter worden omgezet naar een hoofdletter door middel van `string.upper()`.

OPGAVE 10.19
Pas uw nieuwe versie van `caesar` aan voor hoofdletters. Als de string een hoofdletter bevat, bevat de uitvoer een correct verschoven hoofdletter. Met andere woorden, `caesar(3, 'Aa')` geeft als uitvoer `'Dd'`.

Hint: `caesar` lijkt sterk op de hierboven beschreven versie van `rot13`.

Caesar-encryptie
zie leereenheid 13

Het `caesar`-programma dat u nu heeft gemaakt, voert een zogenaamde Caesar-encryptie uit. Dit is een eenvoudig voorbeeld van een (onveilig) cryptosysteem. In leereenheid 13 wordt er dieper ingegaan op cryptosystemen. Het Caesar-encryptiesysteem komt dan ook weer aan bod.

5 Een cellulaire automaat

We hebben nu wat ervaring met strings manipuleren in Python. Het is u al gelukt om een eenvoudig encryptieprogramma te maken: het programma `caesar(n, string)`. In deze paragraaf gaan we op de ingeslagen weg en bouwen we langzaam een iets groter programma op. We beginnen met de functie `watdoeik`.

```
1 def watdoeik(string):
2     ret = 0
3     for i in string:
4         if i == "*":
5             ret += 1
6     return ret
```

OPGAVE 10.20
Wat doet de functie `watdoeik`?

U kunt `watdoeik` bijvoorbeeld met de volgende strings testen.

```
1 print(watdoeik(""))
2 print(watdoeik("1*2*3*4*5*6"))
```

We kunnen functies als bouwstenen voor andere functies gebruiken. Stel we hebben een functie `numstars(string)`, die het aantal sterren in de invoerstring telt. Daarmee kunnen we een functie `nextstring(string)` maken, die een nieuwe string maakt door de invoerstring te kopiëren, maar na iedere oorspronkelijke ster `numstars(string)` sterren in te voegen. Zie hieronder voor een voorbeeld van invoer en resultaat.

```
print(">" + nextstring("*") + "<")
# uitvoer: >**<
print(">" + nextstring("a*bc*efg") + "<")
# uitvoer: >a***bc***efg<
print(">" + nextstring("a*bc**efg") + "<")
# uitvoer: >a****bc********efg<
```

OPGAVE 10.21
Schrijf een functie `numstars` en een programma `nextstring` die werken zoals hierboven beschreven.

In `nextstring` telt iedere '*' in de invoerstring mee voor het aantal sterren dat ingevoerd moet worden. We gaan nu enkel de linker- en rechterbuur van ieder karakter meetellen. Een probleem hierbij is het bepalen van buren van het eerste en laatste karakter van de string - van de 'randen' van de string. Eén optie is om te testen of we op de rand zitten. Een andere truc staat hieronder:

```
def nextstring(string):
    widestr = " " + string + " "
    ...
    for i in range(1, len(widestr) - 1):
        i_en_buren = ....
        ...

#voorbeelden voor test = "1234567890":
# in next_gen(test), voor i = 1
#   i_en_buren = "123"
# in next_gen(test), voor i = 8
#   i_en_buren = "890"
```

OPGAVE 10.22

a Wat vult u in bovenstaand programma in voor regel 5?
Hint: zie opgave 10.11.

b Wat gebeurt er in de functie `nextstring` bij de randen van de invoerstring?

Nu we weten hoe we alleen de buren meenemen, maken we een nieuwe versie van `nextstring` die enkel naar de buren kijkt. Dus: voor positie `i` tellen alleen sterren mee als ze op positie `i-1`, `i`, of `i+1` in de invoerstring staan. Een voorbeeld in- en uitvoer staat hieronder.

```
print("1. >" + nextstring("*") + "<")
print("2. >" + nextstring("a*bc*efg") + "<")
print("3. >" + nextstring("a*bc**efg") + "<")
# uitvoer:
# 1. >**<
# 2. >a**bc**efg<
# 3. >a**bc******efg<
```

OPGAVE 10.23
Pas uw programma `nextstring` aan zoals hierboven beschreven.
Hint: gebruik `range(len(string))` om de tekens van de string één voor één af te lopen.

Tot slot maken we nog één aanpassing. In plaats van extra sterren toe te voegen, zetten we nu alleen een ster in de uitvoerstring als de invoer maar één '*' heeft voor deze plaats en zijn buren. Zo nee, dan kunnen we natuurlijk niet het invoerkarakter kopiëren (want dat zou een '*' kunnen zijn). In dat geval zetten we een spatie. Een voorbeeld van in- en uitvoer staat hieronder. De spaties zijn zichtbaar gemaakt voor de duidelijkheid.

```
print("1.␣>" + nextstring("␣*␣") + "<")
print("2.␣>" + nextstring("*␣*") + "<")
print("3.␣>" + nextstring("␣**␣*␣**") + "<")
# uitvoer:
"1.␣>***<"
"2.␣>*␣*<"
"3.␣>*␣␣␣*␣␣␣<"
```

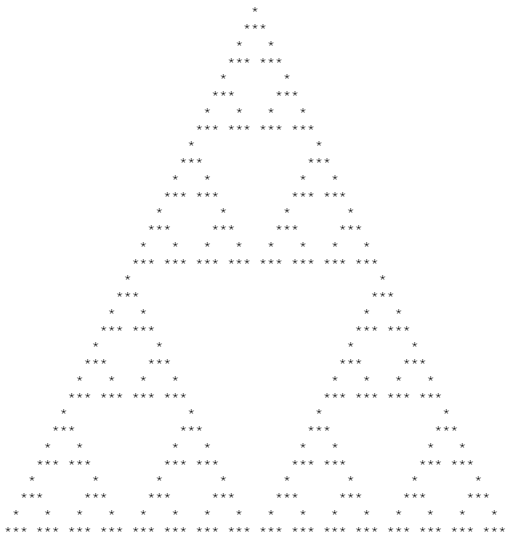
OPGAVE 10.24
Pas uw programma `nextstring(string)` aan. De uitvoer is een string waar voor iedere positie `pos` geldt:
– een '*' als in de invoerstring op deze positie en zijn twee buren precies één '*' staat
– zo nee: een spatie.

Het programma dat u in opgave 10.24 heeft gemaakt, is een voorbeeld van een *cellulaire automaat*. Een cellulaire automaat is een programma dat voor iedere plek in de uitvoer berekent wat daar moet staan, op basis van wat er op die plek en bij de directe buren staat in de invoer.

Cellulaire automaten zijn vrij eenvoudig, maar kunnen toch complexe uitvoer genereren. Met behulp van onderstaande functie kunt u een aantal strings proberen. Probeer een aantal strings uit en bekijk het resultaat.

```
def test_cell(string):
    for i in range(32):
        print(string)
        string = nextstring(string)
```

Tot slot tonen we één voorbeeld van complexe uitvoer. Als invoer gebruiken we een string van spaties, met in het midden één '*'. In figuur 10.1 ziet u wat er gebeurt als we `test_cell` toepassen op deze initiële string.



FIGUUR 10.1 De uitvoer van `test_cell` toegepast op een string van één '*' met 39 spaties aan iedere kant.

6 Programmeerparadigma's

Imperatief programmeren
Programmeerparadigma's

De programmeerstijl die we hierboven hebben gezien noemen we *imperatief programmeren*. Er zijn echter andere manieren om te programmeren: zogeheten *programmeerparadigma's*. Hier bespreken we de vier bekendste programmeerparadigma's.

Imperatief programmeren

Imperatief programmeren: eerst dit, dan dat

Statement

Imperatief programmeren is net als een recept volgen: eerst dit, dan dat. Het programma is een lijst instructies (*statements*), die door de computer in de opgegeven volgorde uitgevoerd worden.

Functioneel programmeren

Functioneel programmeren: Evalueer de functie

Expressie

Bij functioneel programmeren zijn de programma's functies die berekeningen uitvoeren. Er is dus geen 'volgend' statement zoals bij imperatief programmeren, er zijn enkel functies en berekeningen (*expressies*), er zijn geen statements. NB: functies zoals die in Python worden aangemaakt door middel van `def functienaam()` zijn niet noodzakelijk een voorbeeld van functioneel programmeren – alleen als ze geen statements hebben, maar enkel functies en expressies, zijn het ook functionele programma's.

Een simpel voorbeeld van een functioneel programma staat hieronder. Dit programma berekent de faculteit van een opgegeven getal n (het product van de getallen 1 tot en met n).

```
1 factorial:: Int->Int
2 factorial 0 = 1
3 factorial n = n * factorial (n - 1)
```

Ter illustratie: Als dit programma wordt uitgevoerd met `factorial 3`, dan retourneert het de waarde 6 (namelijk $3 \cdot 2 \cdot 1 \cdot 1$).

Objectgeörienteerd programmeren: objecten praten met elkaar

Objectgeörienteerd programmeren

In objectgeörienteerd programmeren bestaat de wereld uit 'objecten'. Deze objecten wisselen berichten met elkaar uit. Denk bijvoorbeeld aan een online spel: er lopen vele spelers rond, en vele vijanden vanuit het spel. In plaats van dat het spel op één centrale plaats alle details (hitpoints, wapen, kracht, ...) van iedereen bijhoudt, houdt ieder 'object' (hier: spelers en vijanden) dat zelf bij. Hieronder staat een voorbeeld in Java. In dit voorbeeld kan een speler een ander aanvallen door middel van `Player.attack()`, dat vervolgens een bericht `damage(power)` stuurt naar de aangevallen speler.

```
public class Player {
     private int hitpoints;
     private int power;

     public void damage(int dmg) {
          this.hitpoints = this.hitpoints - dmg;
     }

     public void attack(Player vijand) {
          vijand.damage(power);
     }
}
```

Logisch programmeren: antwoorden afleiden

Logisch programmeren

Logisch programmeren verschilt behoorlijk van de andere hier besproken paradigma's. Bij logisch programmeren geeft de programmeur een aantal feiten op (een kennis-database), waarna hij/zij vragen kan stellen. In het voorbeeld hieronder zien we een aantal dieren, waarvan sommige huisdier zijn en sommige kunnen vliegen. Verder is er nog een regel die zegt dat een vliegend huisdier een huisdier is dat kan vliegen.

```
huisdier(hond).
huisdier(vogel).
insect(vlieg).
vliegt(vlieg).
vliegt(vogel).

vlieghuisdier(X) :-
    huisdier(X),
        vliegt(X).
```

Nadat deze kennis-database aan het systeem is gegeven, kunnen er vragen gesteld worden, zoals 'is een vlieg een vliegend huisdier': `vlieghuisdier(vlieg)`. Het systeem leidt dan het antwoord af uit de gegeven informatie (hier: nee).

Een programmeertaal kan meerdere paradigma's ondersteunen. Verschillende paradigma's kunnen zelfs in één programma gemengd worden. Python ondersteunt bijvoorbeeld het imperatieve paradigma, maar ook het objectgeörienteerde paradigma en het functionele paradigma.

SAMENVATTING

Programmeren is het maken van programma's. Programma's zijn de instructies die een computer uit moet voeren. Computers gebruiken intern machinetaal – reeksen cijfers die rechtstreeks naar de processor gestuurd worden. Dit is voor mensen vrijwel onleesbaar. De eerste programma's werden geschreven in assembly. Assembly blijft heel dicht bij machinetaal, maar gebruikt woorden in plaats van cijfers. Vanuit dit primitieve beginpunt zijn programmeertalen ver geëvolueerd. Programma's worden tegenwoordig in een algemene programmeertaal geschreven, waarna de computer het omzet in machinetaal.

Variabelen zijn stukjes geheugen waarin een programma data kan opslaan en later weer gebruiken. Het *type* van een variabele wordt bepaald door wat voor soort data er opgeslagen is. De behandelde datatypen zijn *int* voor gehele getallen, *bool* voor True/False, *str* voor strings, en *lijsten* voor rijtjes van data.

Een *functie* is een stukje programma dat een bewerking verricht en eventueel data retourneert aan het aanroepende programma. Functies hebben nul of meer *argumenten*: stukjes data die ze in hun bewerking kunnen gebruiken.

Een functie is één *blok* van programmacode. In Python worden blokken aangegeven door in te springen. Blokken bepalen de *scope* van variabelen: een variabele bestaat in het blok waarin de variabele is gedefiniëerd in de daaronderliggende blokken, zolang daar niet een andere variabele met dezelfde naam wordt gedefiniëerd. Variabelen die in het buitenste blok zijn gedefiniëerd heten *globale* variabelen, variabelen die in een lager blok zijn gedefiniëerd zijn lokale variabelen.

In een programmeertaal kunnen we herhaling en keuzes gebruiken. Eén manier om herhaling in Python aan te geven is door middel van `for i in LIJST`, waarbij de lijst kan worden opgebouwd door `range`, bijvoorbeeld `range(GETAL)` of `range(START, EIND, STAPGROOTTE)`. Keuzes in Python kunnen worden gemaakt door middel van `if conditie:` en `else:`.

Python kent een aantal manieren om strings te bewerken. Een *substring* bepalen kan door middel van `string[begin:eind]`, het i^e karakter van een string is gegeven door `string[i-1]`. Omgekeerd kan de positie van een karakter binnen een string worden gezocht door `STRING.index(ZOEKSTRING)`. Python kan een string aflopen door `for i in range(STRING)`.

Dit alles kan worden gecombineerd om ingewikkeldere programma's op te bouwen. We hebben twee voorbeelden van complexere programma's behandeld, een encryptieprogramma en een cellulaire automaat. Het programma `caesar` is een simpel encryptieprogramma dat teksten versleuteld door verschuiving in het alfabet. Als uitsmijter heeft u een *cellulaire automaat* gebouwd. Dat is een programma waar de waarde van iedere plaats in de uitvoer afhangt van de originele invoer op die plek en de directe buren.

Tot slot bespraken we verschillende concepten waarop geprogrammeerd kan worden, zogenaamde programmeerparadigma's. We hebben vier paradigma's behandeld.

- *Imperatief programmeren* lijkt sterk op recepten schrijven: de instructies, ofwel statements, worden in volgorde opgeschreven.
- *Functioneel programmeren* blijft dicht bij de wiskunde: programma's bestaan uit niets anders dan functies. In tegenstelling tot imperatief programmeren zijn er geen statements, er zijn enkel expressies, ofwel berekeningen.
- In *objectgeörienteerd programmeren* bestaan er objecten, die zelf hun eigen toestand bijhouden en berichten met elkaar uitwisselen.
- Bij *logisch programmeren* geeft de programmeur een kennis-database in, waarna er antwoorden op vragen over de kennis-database automatisch kunnen worden afgeleid.

ZELFTOETS

1 Schrijf een programma `plek_in_alfabet(letter)` die voor de invoerletter teruggeeft op welke plek deze letter in het alfabet staat, waarbij a = 0, b = 1, c = 2, . . . , z = 25.

2 Het volume van een rechthoekige piramide is een derde van de oppervlakte van de rechthoek maal de hoogte (oppervlakte $\cdot \frac{1}{3}h$). Maak onderstaand programma af, zodat het correct de inhoud van de piramide retourneert.

```
1 def opp_pyramid(lengte, breedte, hoogte):
2     opp = opp_rechthoek(lengte, breedte)
3     return opp *...
```

3 Schrijf een programma `schuif_letter(lettera, letterb)` dat als invoer twee kleine letters krijgt, en als uitvoer `letterb` verschuift met `plek_in_alfabet(lettera)` plaatsen.
Bijvoorbeeld: `schuif_letter('a', letter)` retourneert `letter` (omdat `plek_in_alfabet('a')` 0 retourneert).

4 Beschouw onderstaand programma, dat gebruikmaakt van `schuif_letter`.

```
1 def watdoeik(key, plaintext):
2     retstr = ""
3     for i in range(len(plaintext)):
4         origletter = plaintext[i]
5         shiftletter = key[i % len(key)]
6         retstr += schuif_letter(shiftletter, origletter
              )
7     return retstr
```

a Wat is de scope van variabele `key`? En van variabele `shiftletter`? Geef uw antwoord in regelnummers.
b Wat doet het programma?