Programming Basics

Editor: Martijn Stegeman

November 21, 2019

Contents

Chapter 1

Calculations

1.1 Calculations using whole numbers

Evaluate the following expressions.

1.1

a. `1 / 5`
b. `2 / 6`
c. `3 / 8`
d. `4 / 5`
e. `5 / 6`

1.2

a. `1 % 5`
b. `2 % 6`
c. `3 % 8`
d. `4 % 5`
e. `5 % 6`

1.3

a. `3 / 1`
b. `7 / 2`
c. `7 / 3`
d. `5 / 4`
e. `8 / 5`

1.4

a. `3 % 1`
b. `7 % 2`
c. `7 % 3`
d. `5 % 4`
e. `8 % 5`

1.5

a. `2 / 1`
b. `6 / 2`
c. `12 / 3`
d. `8 / 4`
e. `25 / 5`

1.6

a. `2 % 1`
b. `6 % 2`
c. `12 % 3`
d. `8 % 4`
e. `25 % 5`

1.7

a. `3 / 8`
b. `4 / 2`
c. `8 / 9`
d. `1 / 2`
e. `5 / 7`

1.8

a. `3 % 8`
b. `4 % 2`
c. `8 % 9`
d. `1 % 2`
e. `5 % 7`

1.9

a. `14 / 11`
b. `16 / 14`
c. `5 / 16`
d. `20 / 12`
e. `15 / 17`

1.10

a. `14 % 11`
b. `16 % 14`
c. `5 % 16`
d. `20 % 12`
e. `15 % 17`

Expressions An expression is a combination of numbers and operations. Such an expression may be *evaluated* using the mathematical rules that you may already know. For example:

expression	evaluates to
`4`	`4`
`5`	`5`
`4 + 5`	`9`
`4 * 5`	`20`

Next to the more familiar operations we will explain several exceptions and particulars that come into play when doing calculations using a computer.

Integer division Computers treat *integers*, or whole numbers, differently from *floats*, numbers with a decimal point. In most situations you can expect similar results, but there are some differences. The first difference is division: the expression `1 / 2` surprisingly evaluates to `0`. This is because division of two integers is interpreted as asking the following question: *how often does the number 2 fit into the number 1*? In the following table, we list the results of dividing some numbers by 3.

`x`	0	1	2	3	4	5	6	7
`x / 3`	0	0	0	1	1	1	2	2

Modulo An operation that you might not know is modulo, often written as `%`. This operation nicely complements the integer division; for example, in the case of an expression like `5 % 2`, we should answer the following question: *2 fits 2 times into 5, how much is then left*? The answer is `1`. In the following table, we list the the results of performing a modulo 3 for some numbers. When you compare it to the previous table, you should see that it does indeed list what is "left" after performing an integer division.

`x`	0	1	2	3	4	5	6	7
`x % 3`	0	1	2	0	1	2	0	1

Divisibility We call an integer *divisible* by another integer if the result of taking the modulo is `0`, or in other words: if nothing is left after the division.

1.2 Precedence rules

Evaluate the following expressions. Note the different rules that may apply.

1.11

a. `1 / 8 * 5`
b. `5 * 3 * 6`
c. `1 * 7 / 2`
d. `5 / 5 / 5`
e. `7 / 6 / 9`

1.12

a. `5 % 1 / 3`
b. `8 % 5 % 8`
c. `3 % 1 / 8`
d. `8 / 8 % 1`
e. `7 % 8 / 6`

1.13

a. `7 + 6 * 9`
b. `7 - 5 * 3`
c. `7 * 2 - 7`
d. `5 / 1 + 5`
e. `3 / 4 + 3`

1.14

a. `1 / 5 + 6`
b. `2 / 9 - 2`
c. `6 - 8 / 8`
d. `2 * 7 + 3`
e. `8 - 6 * 1`

1.15

a. `(2 + 3) / 4`
b. `3 / (4 - 3)`
c. `(5 - 3) * 1`
d. `6 / (1 - 7)`
e. `(1 + 8) * 8`

1.16

a. `4 * (1 - 1)`
b. `(1 - 9) / 3`
c. `4 * (9 + 1)`
d. `(1 - 3) / 9`
e. `6 * (1 - 2)`

1.17

a. `(8 - 5) % 5 % 8 - 3`
b. `1 % 9 - 3 - 4 % 9`
c. `2 - (7 / 7) - 3 / 4`
d. `8 + 2 * 6 * 8 + 3`
e. `4 % 2 - 7 % 1 - 3`

1.18

a. `5 + 1 * 9 + 9 * 4`
b. `8 * 4 - 7 * 1 - 6`
c. `8 % (9 - 6) % 3 - 3`
d. `7 % 2 - 8 % 7 - 8`
e. `2 / 7 + 3 + (2 / 2)`

1.19

a. `3 / 2 - 1 / 9 - 6`
b. `9 + (7 * 3) * 4 + 9`
c. `3 % 3 - 4 - 3 % 9`
d. `1 / 6 - 7 / (3 - 2)`
e. `8 * 7 + 6 + 2 * 4`

1.20

a. `8 % 3 - 6 % 5 - 6`
b. `5 * 9 + 8 * (4 + 6)`
c. `5 + 7 * 1 + 4 * 9`
d. `(3 - 5) * 4 - 7 * 8`
e. `5 - 2 * 8 * 6 - 5`

Order of evaluation When expressions contain more than a single operator, the order of evaluation becomes important. We will formulate a few rules that define what comes first when evaluating such expressions.

The main rule is that, when we are dealing with two operators of the same kind, the operations are performed *left to right*. For example:

`1 / 2 / 2` gives `0 / 2` gives `0`

Should you evaluate this expression the other way around, so from right to left, you will see that this will give you a wholly different result: `1`. So, following these rules is important, especially when division or modulo are involved.

Operator precedence Programming languages define a list of precedence rules for operators, in order to leave no ambiguity as to the outcome of calculations. In most cases, the rules are the same as in modern mathematics. For now, we can define the following groups for basic calculations:

1. any parts of the expression contained between parentheses come first
2. then come the arithmetic operators `*`, `/` and `%`
3. and finally the operations `+` and `-` will be performed

This list does not define what you should do if you, for example, encounter an expression containing multiplication `*` and division `/`. In such a case, where two operators are from the same group, you should default to the from-left-to-right rule.

1.3 Calculations using multiple kinds of numbers

Evaluate the following expressions. Also note the type of the result.

1.21

a. `1.0 - 1.5`
b. `1.0 / 1.0`
c. `1.0 + 1.5`
d. `0.5 + 0.5`
e. `1.0 * 1.0`

1.22

a. `0.5 - 1.0 * 1.5`
b. `0.5 + 0.5 * 1.5`
c. `0.5 * 1.0 + 1.0`
d. `1.5 - 0.5 * 1.0`
e. `1.0 / 0.5 + 1.5`

1.23

a. `0.5 - 0.75`
b. `1 / 2.0`
c. `0.5 / 0.25`
d. `1 - 1.5`
e. `3 * 1.5`

1.24

a. `2 - 2.0`
b. `0.5 + 0.75`
c. `1.0 / 1.0`
d. `1 * 1.5`
e. `1 + 0.5`

1.25

a. `2 - 2 + 2`
b. `2 / 4 * 3.0`
c. `0.5 * 2 / 4`
d. `1.0 - 2 / 6`
e. `1.5 * 3.0 - 3`

1.26

a. `3 + 3 - 3`
b. `1.0 - 6 + 2.0`
c. `3 + 4.5 * 9`
d. `0.5 - 2 * 3`
e. `1 - 0.5 * 1.5`

1.27

a. `2 + 3.0 + 3.0`
b. `0.5 / 1.0 + 1`
c. `1.0 * 4 * 1.0`
d. `2 - 2 + 6`
e. `1.5 + 6 / 1.5`

1.28

a. `3 - 9 + 1.5`
b. `3 - 4 * 4`
c. `1.5 + 3.0 / 6`
d. `3 * 6 / 6`
e. `1.0 * 2.0 / 4`

1.29

a. `1 + 1.5 + 3`
b. `1 - 3 - 0.5`
c. `2 + 2 + 2.0`
d. `3 - 1.5 * 4.5`
e. `2 / 2.0 + 2.0`

1.30

a. `1.0 / 2.0 * 6`
b. `2 + 2.0 + 6`
c. `2 + 2.0 + 2.0`
d. `1.5 * 6 - 6`
e. `3 + 4.5 + 9`

Floats Numbers with a decimal point, or *floating point numbers*, generally work as you would expect from mathematics. Notably, dividing floats like `1.0 / 2.0` gives `0.5`.

Automatic conversion In expressions you may encounter combinations of floats and integers. This requires a decision as to what rule to apply. Often, this is solved by doing automatic conversion: if only a single float is involved, the other number (an integer) will be converted to a float, too. The result of the operation will then also be a float.

Note that precedence rules still define what happens first: the expression `3 / 2 + 0.25` evaluates to `1.25`. First, the subexpression `3 / 2` is evaluated, giving the integer `1`. Then, `1` and `0.25` are to be added. Only then does the automatic conversion kick in, making the result a float: `1.25`.

Chapter 2

Logic

2.1 Propositions

Evaluate the following propositions to true or false.

2.1

a. `3 == 2`
b. `1 != 1`
c. `3 == 4.0`
d. `2.2 == 2`
e. `1 != 2`

2.2

a. `1.5 * 2 == 3.0`
b. `6 == 6 * 6`
c. `3.0 == 2 * 1.0`
d. `4 == 2 * 2`
e. `2 * 3.0 == 5.0`

2.3

a. `1.0 > 1.5`
b. `3 > 1.5`
c. `2 > 3.0`
d. `1 > 1.5`
e. `0.5 < 0.5`

2.4

a. `1.0 <= 0.5`
b. `1 >= 0.5`
c. `1 >= 3.0`
d. `1.5 >= 0.75`
e. `1 >= 1.5`

2.5

a. `3 == 1 / 1`
b. `0.5 * 1 == 1.0`
c. `2 + 1.0 <= 1.0`
d. `1.0 / 0.5 <= 1`
e. `1.0 >= 3 / 1.0`

2.6

a. `1.0 <= 2 - 0.5`
b. `3 >= 2 + 3`
c. `2 == 3 - 1.5`
d. `1.0 <= 0.5 - 1`
e. `1.5 - 1.0 >= 1.5`

2.7

a. `1.5 / 0.5 == 3.0`
b. `1 * 1.5 >= 3`
c. `1.5 >= 1.0 * 1`
d. `1 >= 2 - 1.0`
e. `0.5 >= 2 + 3`

2.8

a. `1.5 == 1 * 1.5`
b. `1.0 / 2 == 1.5`
c. `3 - 1 == 2`
d. `0.5 + 0.5 >= 3`
e. `1.5 == 1 + 2`

2.9

a. `2 <= 0.5 * 3`
b. `1.5 / 1.5 <= 1.5`
c. `0.5 - 1.5 == 3`
d. `0.5 - 1.0 >= 1`
e. `3 == 0.5 - 3`

2.10

a. `1.0 + 1 >= 2`
b. `1 <= 1.0 / 1.5`
c. `1.0 + 1 == 1`
d. `1 == 1.5 * 1`
e. `2 >= 3 * 3`

Propositions are true or false For propositions like `4 == 5` we can decide if they are *true*. The number `4` does not equal `5`, which makes this proposition not true, or *false*. There are six important ways to formulate a proposition:

operator	meaning
`==`	equals
`!=`	does not equal
`<`	is smaller than
`>`	is larger than
`<=`	is smaller than or equal to
`>=`	is larger than or equal to

Each proposition formulated using one of these operators can evaluate to one of two values: `true` or `false`. These values are called *booleans*[1]. Expressions that evaluate to `true` or `false` are called *boolean expressions*.

Precedence rules Like with arithmetic operations, propositional operations are subject to rules of precedence.

1. any parts of the expression contained between parentheses come first
2. then come the arithmetic operators `*` and `+` (note the rules that apply between those!)
3. and finally the operations `>`, `<`, `==`, `!=`, `>=`, `<=` will be performed

Mixing floats and integers Propositions can contain integers as well as floats. If an integer is compared to a floating point number, automatic conversion takes place, making the integer into a float.

[1] After George Boole, who appears to be the first to develop an algebraic system for logic in the 19th century.

2.2 Logic operators

Evaluate the following propositions to true or false.

2.11

a. `true && 2 >= 2`
b. `1 >= 1.0 && true`
c. `2 >= 2 && false`
d. `true && 3 <= 1`
e. `2 >= 3 && false`

2.12

a. `false || 3 >= 2`
b. `0.5 == 1.5 || true`
c. `2 <= 1.5 || false`
d. `1 <= 1.0 || true`
e. `0.5 == 1.5 || true`

2.13

a. `false && !(1.5 == 0.5)`
b. `true && !(1 <= 1.5)`
c. `!(1.0 <= 1.0) || true`
d. `!(1 >= 1) || true`
e. `false || !(1 <= 3)`

2.14

a. `1 >= 3 || !(0.5 == 0.5)`
b. `2 >= 1.0 || 2 <= 1.5`
c. `1.0 == 2 || 3 >= 1`
d. `1.5 <= 2 || !(3 >= 1.5)`
e. `0.5 == 1.0 && 1.0 >= 1.0`

2.15

a. `1.0 <= 2 && 1.0 <= 0.5`
b. `!(1 <= 1.0) && 2 <= 3`
c. `1.0 <= 2 && 2 >= 0.5`
d. `!(3 == 1.0) || 2 >= 1`
e. `3 >= 2 || 1.0 <= 3`

2.16

a. `0.5 <= 2 || !(2 >= 0.5)`
b. `2 <= 1.5 && 1 <= 3`
c. `3 >= 0.5 || 1.0 >= 0.5`
d. `1 <= 1.5 || 3 >= 1`
e. `!(1.5 == 3) || 1 <= 1`

2.17

a. `3 >= 1 || 2 <= 1`
b. `1 == 0.5 || 1.5 == 1.5`
c. `2 <= 0.5 || 1.0 >= 1`
d. `1.5 >= 3 && !(3 <= 1.5)`
e. `!(0.5 >= 0.5) || 3 <= 0.5`

2.18

a. `!(0.5 <= 2) && 1.0 >= 0.5`
b. `2 <= 1.5 && 1.0 <= 3`
c. `1.5 == 1 && !(1.5 >= 3)`
d. `!(0.5 >= 0.5) || 1 == 1.5`
e. `!(3 <= 1.5) || 2 >= 0.5`

2.19

a. `3 >= 2 || 2 >= 1.0`
b. `1.0 >= 1.0 || 1.0 >= 1.5`
c. `1.5 >= 1.5 && !(2 <= 3)`
d. `!(1.0 == 3) || 3 >= 1.0`
e. `1 >= 1.5 || 2 == 1.0`

2.20

a. `2 <= 2 && !(3 == 1)`
b. `3 >= 1.5 && 1 >= 1`
c. `1.0 <= 3 && 3 >= 1.5`
d. `!(3 <= 1.0) && 1.0 == 1.5`
e. `1.5 >= 2 || 1.5 == 1`

Logische operaties Het is mogelijk om de waarden `true` en `false` te combineren tot meer complexe logische expressies. Zo zijn er de logische operaties `&&` (and) en `||` (or). Dit zijn de zogeheten waarheidstabellen voor deze operaties:

expressie	geeft
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

expressie	geeft
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Ontkenning De uitkomst van een bewering kan ook "omgedraaid" worden. Zet er dan `!` (not) voor:

`!true` geeft `false` en `!false` geeft `true`

Prioriteitsregels Ook `&&` en `||` hebben hun plek in de prioriteitsregels:

1. eerst de expressies tussen haakjes
2. dan rekenkundige operaties zoals `+` en `*`
3. dan beweringen zoals `>`, `<`, `==`, `!=`, `>=`, `<=`
4. `!`
5. `||`
6. `&&`

2.3 Exercises: conditions

In de vorige paragrafen heb je steeds expressies *geïnterpreteerd*, de uitkomst berekend op basis van de rekenregels. Nu vragen we je om zelf expressies te schrijven. Dit is wat creatiever, en het kan soms moeilijk zijn om op het juiste idee te komen. Blader dan eens terug naar de vorige delen voor inspiratie.

Als je zo'n expressie schrijft, kun je deze zelf nalopen met behulp van de regels die je kent. Schrijf je expressie op, en probeer deze te interpreteren. Je kunt ook een aantal "testgevallen" bedenken die je voor het getal `n` invult, om te kijken of de uitkomst naar verwachting is.

7.1 Schrijf een expressie die test of een getal `n` kleiner is dan 5.

7.2 Schrijf een expressie die test of een getal `n` tussen 5 en 10 zit, waarbij 5 en 10 zelf óók mogen.

7.3 Schrijf een expressie die test of een getal `n` deelbaar is door 3.

7.4 Schrijf een expressie die test of een getal `n` *even* is.

7.5 Schrijf een expressie die test of een getal `n` *oneven* is.

7.6 Voor welke waarden van `n` is de bewering `!(n > 1 && !(n > 3)) && true` waar?

7.7 De bewering uit de vorige vraag kan eenvoudiger worden geschreven. Doe dit, en controleer je vereenvoudigde versie door te kijken of deze nog steeds waar is voor dezelfde waarden van `n`.

Chapter 3

Variables

3.1 Variables

Evaluate the following code fragments and write down the final value for all variables.

3.1

```
dey = 3 + 0.5
luo = 1.5
```

3.2

```
gog = 1 * 3
oer = 3 * 0.5
```

3.3

```
eli = 1.0
soc = 1.0 + 3
```

3.4

```
jon = 1.0 - 2
aus = 2 * 0.5
```

3.5

```
abe = 1.5 / 1.0
sir = abe / 0.5
```

3.6

```
vow = 2 * 1.5
nub = vow + 1.0
```

3.7

```
gup = 1 / 2
ley = gup * 2
```

3.8

```
tez = 1.5 + 1
nap = tez * 3
```

3.9

```
jib = 1.0 - 3
jib = 2 + 3
```

3.10

```
aus = 2 - 2
aus = 3 - 1.5
```

3.11

```
bam = 3 / 2
bam = 1.0 + 1.0
```

3.12

```
tye = 1.5 / 1.5
tye = 1 * 1
```

3.13

```
nam = 1 + 1
sob = 1.0 - 1.0
nam = nam * 2
off = sob / 0.5
off = nam / 3
off = off + 3
```

3.14

```
kaw = 2 * 1
kaw = 1.5 * 1.0
zan = kaw / 2
zan = kaw * 1
kaw = 3 + 0.5
kaw = zan / 1.5
```

3.15

```
ora = 2 / 1.0
ora = 1 - 0.5
ora = ora + 0.5
ora = ora / 4
quo = ora - 1
ora = ora - 3
```

3.16

```
gup = 3 + 0.5
gup = 1 + 1.0
han = 2 / 2
gup = gup + 0.5
han = gup - 1
han = gup + 1.0
```

3.17

```
lod = 2 / 2
lod = 2 * 3
nid = lod / 1.0
mou = lod * 3
lod = lod - 1
gyp = 0.5 - 3
```

3.18

```
iwa = 3 - 2
you = iwa + 2
way = iwa - 0.5
iwa = way - 1
iwa = 2 / 2
iwa = iwa + 0.5
```

3.19

```
oka = 3 - 1
oka = oka * 1
fei = oka * 1
oka = oka * 1.5
oka = oka * 1
oka = oka / 0.5
```

3.20

```
fae = 1 + 0.5
fae = fae - 1.0
fae = 2 * 0.5
fae = fae - 1.5
fae = 0.5 - 2
fae = fae + 2
```

Assignment In the following code fragment, a *value* is assigned to a *variable*:

```
het = 2.2
```

Upon executing that line of code, a variable is created with the name `het`, and it is assigned the value `2.2`. This variable then becomes part of the *final state* after executing the code fragment. However, a code fragment can also contain multiple assignments below each other. In the following example, we assign three variables, each with their own name:

	ike	dev	wan
`ike = 3.14`	**3.14**		
`dev = 3 / 4`	3.14	**0**	
`wan = 0.75`	3.14	0	**0.75**

On the left we show the lines of code that are executed. On the right, we keep track of what happens when executing each line: we *trace* the code fragment. As one variable is being assigned, we draw a box around the new value. On the lines below, that value is retained, which we show by copying the value down. By doing this for all lines, we can read the final state of all variables on the last line: `ike`, `dev` and `wan`, with their accompanying values.

Order It's possible to assign a value to a variable for a second time (or more often). The "old" value will be *overwritten*. This makes *order* of the program important: we always process the lines from top to bottom. Take a look at the following example.

	wei
`wei = 1`	~~1~~
`wei = 4`	**4**

The variable `wei` is assigned a value two times, as is shown by the two boxes that are drawn around the values. But the final state only consists of a single variable named `wei`, with value `4`. The value `1` that was assigned earlier has disappeared when it was overwritten.

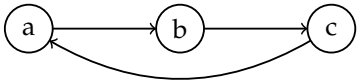
Variables in expressions Now that we have variables, we can also *use* them in calculations, referring to them by their name.

	hat	say
`hat = 1`	**1**	
`say = hat + 4`	1	**5**

The state after executing the final line of code consists of two variables: `hat = 1` and `say = 5`.

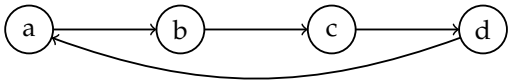
3.2 Exercises: swapping

7.1 Schrijf een algoritme dat de waarden van *drie* variabelen verwisselt, zoals in de volgende illustratie:

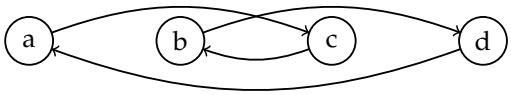


Tip: je kunt waarden verzinnen voor `a`, `b` en `c`, maar dit is niet noodzakelijk om een algemeen algoritme te maken.

7.2 Schrijf een algoritme dat de waarden van *vier* variabelen verwisselt, zoals in de volgende illustratie:



7.3 Schrijf een algoritme dat de waarden van *vier* variabelen verwisselt, zoals in de volgende illustratie:



Verwisseling Laten we eens kijken hoe we een stel toekenningen kunnen gebruiken om een algoritme te maken. Stel dat we twee variabelen hebben:

```
a = 54   en   b = 29
```

waarvan we de waarden willen verwisselen. Als je dit wil oplossen door iedere waarde aan de *andere* variabele toe te kennen, zou je het volgende algoritme kunnen voorstellen:

```
a = b
b = a
```

Laten we de werking van dit algoritme traceren. We beginnen met de twee startwaarden, en laten dan *per regel code* zien wat het effect is op onze variabelen. We laten met een kader steeds zien welke variabele wordt bijgewerkt op welke regel.

	a	b
	54	29
a = b	**29**	29
b = a	29	**29**

Let op dat dit niet het gewenste eindresultaat is! Op de derde regel willen we de waarde van `a` ook in `b` opslaan, maar de oorspronkelijke waarde is op dat moment al overschreven.

Uit de probleemanalyse kunnen we al een beetje afleiden wat de oplossing moet zijn: we moeten er voor zorgen dat op regel 3 de oorspronkelijke waarde van `a` nog beschikbaar is. We kunnen dat doen door een extra variabele te introduceren, puur voor dat doel. We slaan daarmee de waarde van `a` alvast op, vóór deze wordt overschreven:

```
t = a
a = b
b = t
```

Als we nu het hele programma uitschrijven en weer traceren, dan ziet het er zo uit:

	a	b	t
	54	29	
t = a	54	29	**54**
a = b	**29**	29	54
b = t	29	**54**	54

Dit is het gewenste eindresultaat!

Chapter 4

Conditional statements

4.1 Tracing conditional statements

Make a trace-table for each of the following code fragments. Also note the final value of the variable a.

4.1

```
a = 0
if(a == 0)
   a = 10
a = a + 1
```

4.2

```
a = -1
if(a == 0)
   a = 10
a = a + 1
```

4.3

```
a = 0
if(a < 0)
   a = 10
```

4.4

```
a = 0
if(a > 0)
   a = 10
```

4.5

```
a = 0
if(a <= 0)
   a = 10
```

4.6

```
a = 0
if(a >= 0)
   a = 10
```

4.7

```
a = 9
if(a % 3 == 0)
   a = a * 2
```

4.8

```
a = 10
if(a % 3 == 0)
   a = a * 2
```

4.9

```
a = -1
if(a < 0 || a > 10)
   a = a * 2 - 1
```

4.10

```
a = 5
if(a > 0 && a % 2 == 0)
   a = 0
```

Voorwaardelijke opdrachten Het volgende stukje code bevat een voorwaardelijke opdracht, beginnend met `if`. Er is een voorwaarde of *condition*, namelijk `a < 10`. De instructie op de volgende regel is extra geïndenteerd, waarmee we aangeven dat deze *afhankelijk* is van de `if`.

```
1  a = 0
2  if(a < 10)
3      a = a + 10
4  a = a - 1
```

In dit geval start het programma met de waarde `a = 0`. Dus wordt aan de voorwaarde `a < 10` voldaan op het moment dat deze wordt gecontroleerd. Het gevolg is dat de instructie `a = a + 10` uitgevoerd. De laatste regel staat weer helemaal links en is dus géén onderdeel van de voorwaardelijke instructie: deze instructie wordt uitgevoerd onafhankelijk van de `if`. Dat betekent dat het programma eindigt met de waarde `a = 9`.

Traceren We kunnen hier, net als in het vorige hoofdstuk, de toestand van de variabelen traceren. Voor het bovenstaande stukje code ziet de trace er als volgt uit:

regel	1	2	3	4
a	0	0	10	9

In de eerste rij zie je de regelnummers uit het programma. In de rij eronder *traceren* we de waarde van de variabele `a`. Op drie plekken is de waarde van `a` omkaderd: op deze regels wordt inhoud van de variabele gewijzigd.

Niet waar In het fragment hieronder hebben we de voorwaarde aangepast. Op moment van evalueren van de voorwaarde blijkt dat hier niet aan is voldaan. Daarom wordt de volgende regel overgeslagen. Het programma eindigt dus met `a = -1`.

```
1  a = 0
2  if(a > 10)
3      a = a + 10
4  a = a - 1
```

De trace ziet er dan zo uit:

regel	1	2	4
a	0	0	-1

4.2 Tracing with multiple variables

Make a trace-table for each of the following code fragments. Also note the final value of all variables.

4.11

```
a = 0
if(a != 0)
   a = 10
a = a + 1
```

4.12

```
a = 0
if(a != 0)
   a = 10
   a = a + 1
```

4.13

```
a = 3
if(a <= 10)
   a = 10
   a = a * 2
```

4.14

```
a = 3
if(a <= 10)
   a = 10
a = a * 2
```

4.15

```
a = 0
b = 9
if(a < b)
   a = b
```

4.16

```
a = 0
b = 2
if(a < 0)
   b = 8
```

4.17

```
a = 0
b = 2
if(a % 2 == 0 && b % 2 == 0)
   a = -1
b = -2
```

4.18

```
a = 0
b = 3
if(a % 2 == 0 && b % 2 == 0)
   a = -1
b = -2
```

4.19

```
a = 0
b = 2
if(a == b - 2 || a == b)
   a = 6
   b = 8
```

4.20

```
a = 4
b = -1
if(a == 0 && b < 0)
   a = -2
   b = -6
```

Meer afhankelijke instructies In het volgende voorbeeld zie je dat niet één maar twee instructies afhankelijk zijn van de `if`:

```
1  a = 0
2  if(a < 10)
3      a = a + 10
4      a = a * 2
```

De trace is vergelijkbaar met eerder:

regel	1	2	3	4
a	[0]	0	[10]	[20]

Meerdere variabelen Nu introduceren we weer meerdere variabelen. Het gevolg is dat we meer informatie moeten bijhouden bij het traceren. Kijk bijvoorbeeld naar het volgende programma:

```
1  a = 1
2  b = 0
3  if(a > b)
4      c = a
5      a = b
6      b = c
```

De trace ziet er dan als volgt uit, met voor elke variabele een rij:

regel	1	2	3	4	5	6
a	[1]	1	1	1	[0]	0
b		[0]	0	0	0	[1]
c				[1]	1	1

4.3 Deciding with “else”

Make a trace-table for each of the following code fragments. Also note the final value of all variables.

4.21

```
a = 0
if(a == 0)
   a = 10
else
   a = -10
```

4.22

```
a = 1
if(a == 0)
   a = 10
else
   a = -10
```

4.23

```
a = -1
if(a < 0)
   a = 7
else if (a > 2)
   a = 8
else
   a = 9
```

4.24

```
a = 1
if(a < 0)
   a = 7
else if (a > 2)
   a = 8
else
   a = 9
```

4.25

```
a = 3
if(a < 0)
   a = 7
else if (a > 2)
   a = 8
else
   a = 9
```

4.26

```
a = -1
if(a < 0)
   a = 7
else if (a < 2)
   a = 8
else
   a = 9
```

4.27

```
a = 1
if(a < 0)
   a = 7
else if (a < 2)
   a = 8
else
   a = 9
```

4.28

```
a = 3
if(a < 0)
   a = 7
else if (a < 2)
   a = 8
else
   a = 9
```

else Bij een `if` kun je ook een `else` tegen komen. Met het commando `else` geven we aan wat moet gebeuren als *niet* aan de voorwaarde van de `if` is voldaan. De `else` hoort dus altijd bij de `if` die er vlak boven staat.

```
1  a = 1
2  if(a < 0)
3      a = a + 10
4  else
5      a = a - 10
```

Het bovenstaande programma geeft de volgende trace:

regel	1	2	4	5
a	1	1	1	-9

Merk op dat met een `if-else` altijd twee mogelijkheden worden beschreven: ofwel er is aan de voorwaarde voldaan, dus `a < 0`, ofwel `a` heeft een andere waarde, dus `a >= 0`.

else if Om meer dan twee verschillende opties te beschrijven, kunnen met `else if` nog meer mogelijkheden worden toegevoegd. In het volgende programma zijn de drie mogelijkheden `a < 0`, `a > 0` en resterend `a = 0`. Steeds is één afhankelijke instructie aanwezig, die uitgevoerd wordt als aan de juiste voorwaarde wordt voldaan.

```
1  a = 3
2  if(a < 0)
3      a = -1
4  else if(a > 0)
5      a = 1
6  else
7      a = 100
```

In dit geval start het programma met de waarde `a = 3`, en de trace ziet er als volgt uit:

regel	1	2	4	5
var a	3	3	3	1

Door meer dan één `else if` te gebruiken is het mogelijk om nog veel meer dan drie mogelijkheden te beschrijven. De `else` komt wel altijd als laatste; deze beschrijft immers de “laatste mogelijkheid”.

Chapter 5

Repetition using `while`

5.1 Tracing while-loops

Evaluate the following code fragments and write down the final value for all variables. Make a trace-table if needed.

5.1

```
a = 0
while(a < 3)
   a = a + 1
a = a * 2
```

5.2

```
a = -1
while(a < 4)
   a = a + 2
a = a * 3
```

5.3

```
a = 0
while(a <= 6)
   a = a + 2
a = a * 2
```

5.4

```
a = 2
while(a < 8)
   a = a + 2
```

5.5

```
a = 16
while(a > 1)
   a = a / 2
```

5.6

```
a = 2
while(a < 200)
   a = a * a
```

5.7

```
a = 8
while(a < 18)
   a = a / 2 + a
```

5.8

```
a = 0
while(a <= 15)
   a = a * 2 + 1
```

5.9

```
a = 0
while(a >= -15)
   a = a * 2 - 1
```

5.10

```
a = 1
while(a < 16)
   a = a * -2
```

while-loops In het volgende fragment wordt de waarde van een variabele veranderd door middel van een *while-loop*:

```
1  a = 0
2  while(a < 2)
3      a = a + 1
4  a = a / 2
```

Bij het uitvoeren van dit stuk code wordt eerst een variabele aangemaakt met de naam `a`, die op dat moment de waarde `0` krijgt toegekend. Op de regel daarna zien we de term `while`, met daarna tussen haakjes `(a < 2)`. De term geeft aan dat we het hier over een *while-loop* hebben. Het gedeelte tussen haakjes noemen we de *conditie* (of een *bewering*). Zolang de conditie `true` is, zal *alle* code die *binnen* de while-loop staat herhaald worden. Let dus op dat er ook meerdere regels code in de while-loop kunnen staan! De conditie wordt iedere keer maar éénmaal gecontroleerd: aan het begin van het codeblok dat binnen de while-loop wordt uitgevoerd.

Herhalende commando's traceren Natuurlijk kunnen we de effecten van deze code, net als voor variabelen, traceren. Voor het stuk code dat hierboven staat ziet dat er als volgt uit:

regel	1	2	3	2	3	2	4
var a	0	0	1	1	2	2	1
a < 2		true		true		false	

Bovenin de tabel staan de regelnummers die aangeven bij welke regel we op dat moment zijn. Onderin de tabel zetten we alle variabelen (in dit geval alleen `a`), met de waardes die ze hebben bij het respectievelijke regelnummer, en de conditie, met de uitkomst hiervan bij het respectievelijke regelnummer. Bij verandering zetten we een kader om de nieuwe waarde, en in de volgende kolommen nemen we alleen de waarde over. Bij het controleren van de conditie bekijken we het resultaat van de conditie (`true` of `false`).

Zoals je kan zien stopt het programma uiteindelijk met herhalen van de regels twee en drie. Dit gebeurt op regel twee wanneer de conditie `(a < 2)` niet meer waar is (`a` is namelijk `2`, en is dus niet meer kleiner dan `2`). De while-loop houdt dan op met herhalen, en de regel na de while-loop wordt uitgevoerd.

5.2 Complex conditions

Evaluate the following code fragments and write down the final value for all variables. Make a trace-table if needed.

5.11

```
a = 4
while(a == 4)
   a = a + 1
```

5.12

```
a = 4
while(a == 4 || a == 5)
   a = a + 1
```

5.13

```
a = 0
while(a + 1 < 4)
   a = a + 1
```

5.14

```
a = 0
while(a != 4)
   a = a + 2
```

5.15

```
a = 1
while(a != 5)
   a = a + 2
```

5.16

```
a = 0
while(a != 4 && a < 8)
   a = a + 2
```

5.17

```
a = 1
while(a != 4 && a < 8)
   a = a + 2
```

5.18

```
a = 4
while(a + 1 != a * 2)
   a = a - 1
```

5.19

```
a = 16
while(a < -2 || a > 2)
   a = a / -2
```

5.20

```
a = 32
while(a < -2 || a > 2)
   a = a / -2
```

Complexe condities In de opgaven hiernaast gebruiken weer complexere voorwaarden, bestaande uit beweringen en logische combinaties. Hieronder sommen we de regels voor zulke samengestelde beweringen nog eens op.

Beweringen

operatie	**betekenis**
`==`	is gelijk aan
`!=`	is niet gelijk aan
`<`	is kleiner dan
`>`	is groter dan
`<=`	is kleiner dan of gelijk aan
`>=`	is groter dan of gelijk aan

Logische combinaties

expressie	geeft
`true && false`	`false`
`true && true`	`true`
`false && true`	`false`
`false && false`	`false`

expressie	geeft
`true || false`	`true`
`true || true`	`true`
`false || true`	`true`
`false || false`	`false`

Ontkenning

`!true` geeft `false` en `!false` geeft `true`

Vergeet bovendien niet de prioriteitsregels voor het rekenen.

5.3 Multiple variables

Evaluate the following code fragments and write down the final value for all variables. Make a trace-table if needed.

5.21

```
a = 0
b = 0
while(a < 2)
   b = b + 2
   a = a + 1
```

5.22

```
a = 0
b = 0
while(a < 2)
   a = a + 1
   b = a * 2
```

5.23

```
a = 0
b = 0
while(a < 2)
   a = a + 1
   b = a - 2
```

5.24

```
a = 0
b = 1
while(a > -2)
   a = a - 1
   b = b * 2
```

5.25

```
a = 2
b = 1
while(a >= b)
   a = a + 1
   b = b * 2
```

5.26

```
a = 0
b = 0
while(a <= 8)
   a = b * 2
   b = b + 1
```

5.27

```
a = 0
b = 0
while(a < 6 && b < 2)
   a = a + 2
   b = b++
```

5.28

```
a = 0
b = 0
while(a < 6 || b < 2)
   a = a + 2
   b = b++
```

Meerdere variabelen traceren We kunnen `while`-loops ook traceren als er meerdere variabelen in het spel zijn. Kijk naar het volgende codefragment:

```
1  a = 0
2  b = 0
3  while(a < 2)
4      a = a + 1
5      b = b + 1
6  a = 42
```

De bijbehorende trace er als volgt uit:

regel	1	2	3	4	5	3	4	5	3	6
var a	0	0	0	1	1	1	2	2	2	42
var b		0	0	0	1	1	1	2	2	2
a < 2			true			true			false	

Increment In loops is het vaak zo dat je een teller hebt, een variabele die je bij elke iteratie eentje wil ophogen. Omdat dit zo vaak voorkomt hebben veel programeertalen hier een speciale operatie voor: `++`. De opdracht `i++` is equivalent aan `i = i + 1`. De tegenhanger hiervan is `i--`. Hiermee verlagen we `i` juist met `1`.

Chapter 6

Repetition using `for`

6.1 Counting loops

Write down what is printed when running each of the following code fragments.

6.1

```
i = 0
while(i < 3)
   print(i)
   i++
```

6.2

```
i = 12
while(i < 15)
   print(i)
   i++
```

6.3

```
i = 3
while(i >= 0)
   print(i)
   i--
```

6.4

```
i = 14
while(i > 10)
   print(i)
   i--
```

6.5

```
for(i = 0; i < 3; i++)
   print(i)
```

6.6

```
for(i = 3; i >= 0; i--)
   print(i)
```

6.7

```
i = 12
for(; i < 15 ; i++)
   print(i)
```

6.8

```
i = 14
for(; i > 10; i--)
   print(i)
```

6.9

```
for(i = 14; i >= 10; )
   i--
   print(i)
```

6.10

```
for(i = 8; i <= 10;)
   i++
   print(i)
```

6.11

```
for(i = 14; i >= 8; i = i - 2)
   print(i)
```

6.12

```
for(i = 0; i < 10; i = i + 2)
   print(i)
```

For-loops Onderstaand codefragment print de getallen 0 t/m 9 met een `while`-loop.

```
1 i = 0
2 while(i < 10)
3     print(i)
4     i++
```

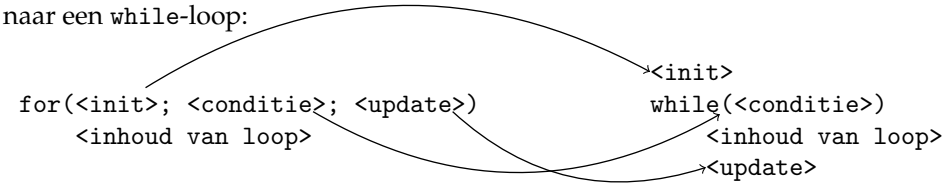
Zoals je misschien is opgevallen bevatten veel loop-constructies min of meer dezelfde ingredienten: Een **initialisatie** (`i = 0`), een **conditie** (`i < 10`), en een **update** (`i++`). Omdat dit patroon zo veel voorkomt, hebben de meeste programmeertalen hier een speciale constructie voor: de `for`-loop. Het onderstaande stukje code laat precies hetzelfde programma zien, maar dan geschreven als een `for`-loop.

```
1 for(i = 0; i < 10; i++)
2    print(i)
```

De `for`-loop bevat dezelfde drie ingredienten als de `while`-loop: De initialisatie (`i = 0`), de conditie (`i < 10`), en de update (`i++`). De initialisatie (`i = 0`), wordt alleen de eerste keer uitgevoerd. De conditie (`i < 10`) wordt elke keer voor het begin van de iteratie uitgevoerd. De update (`i++`) wordt elke keer *aan het einde* van de iteratie uitgevoerd. Dus, de volgorde van executie in dit voorbeeld:

- regel 1, `i = 0` (initialisatie)
- regel 1, `i < 10` (conditie)
- regel 2, `print(i)`
- regel 1, `i++` (update)
- regel 1, `i < 10` (conditie)
- regel 2, `print(i)`
- etc.

Het lastige hierbij is dat regel 1 van de `for`-loop dus eigenlijk 3 verschillende commando's heeft, die ook op verschillende momenten worden uitgevoerd. De makkelijktse manier om te zien hoe een `for`-loop wordt uitgevoerd is door hem te vertalen naar een `while`-loop:



6.2 Tracing for-loops

Evaluate the following code fragments and write down the final value for all variables. Make a trace-table if needed.

6.13

```
for(i = 0; i < 3; i++)
   a = i * 2
```

6.14

```
for(i = 3; i >= 0; i--)
   a = i * 2
```

6.15

```
for(i = 3; i >= 0; i = i - 1)
   a = i * 2
```

6.16

```
a = 16
for(i = 0; i < 3; i = i + 1)
   a = a / 2
```

6.17

```
a = 16
for(i = 0; i <= 2; i = i + 1)
   a = a / 2
```

6.18

```
a = 1
for(i = 2; i <= 8; i = i * 2)
   a = a + 1
```

6.19

```
a = 0
for(i = 1; a < 3; i = i * 2)
   a = a + 1
```

6.20

```
for(i = 0; i < 6; i = i + 1)
   i = i + 1
```

For-loops traceren In het volgende stuk code wordt de waarde van een variabele veranderd binnen een `for`-loop:

```
1  a = -10
2  for(i = 0; i < 2; i++)
3      a = i * 2
4  a = a + 6
```

Als we dit fragment willen traceren hebben we een probleem. Regel 2 bestaat eigenlijk uit meerdere opdrachten: initialisatie (`i = 0`), conditie (`i < 2`) en update (`i++`). We kunnen dit expliciet opnemen in de trace:

regel	1	2 (i = 0)	2 (i < 2)	3	2 (i++)	2 (i < 2)	3	2 (i++)	2 (i < 2)	4
var i		0	0	0	1	1	1	2	2	
var a	-10	-10	-10	0	0	0	2	2	2	8
i < 2			true			true			false	

We hebben in dit voorbeeld voor de duidelijkheid de betreffende commando's onder de labels van de `for`-loop gezet en de waarheidswaarden van conditie expliciet gemaakt. Je kan dit natuurlijk ook wat beknopter opschrijven:

regel	1	2	2	3	2	2	3	2	2	4
var i		0	0	0	1	1	1	2	2	
var a	-10	-10	-10	0	0	0	2	2	2	8

6.3 For and while loops

Rewrite these `for`-loops as `while`-loops.

6.21

```
b = 8
for(a = 5; a > 3; a = a - 1)
   b = b / 2
```

6.22

```
for(e = 0; e < 4; e = e + 1)
   print(e)
```

6.23

```
for(i = 4; i >= 0; i = i - 1)
   print(i)
```

6.24

```
b = 0
for(i = 0; i < 4; i = i + 2)
   b = i * 2
```

Rewrite these `while`-loops as `for`-loops.

6.25

```
a = 1
while(a < 6)
   print(a)
   a = a + 2
```

6.26

```
a = 8
b = 0
while(a > 1)
   b = b + 1
   a = a / 2
```

6.27

```
a = 8
b = 0
while(b < 3)
   a = a / 2
   b = b + 1
```

6.28

```
a = 8
b = 0
while(a > 1 || b <= 3)
   b = b + 1
   a = a / 2
   print(a)
```

Herschrijven Dit is de vergelijking van `for`- en `while`-loops die je al gezien hebt:

```
for(<init>; <conditie>; <update>)
    <inhoud van loop>
```

```
<init>
while(<conditie>)
    <inhoud van loop>
    <update>
```

Die kun je gebruiken om `for`-loops naar `while`-loops om te zetten en andersom. Als je dit doet, kun je beide versies traceren en de uitkomst vergelijken. Zo weet je dat het omzetten precies is gelukt.

Chapter 7

Algorithms: sequences

7.1 Sequences

7.1 Schrijf een algoritme dat de eerste 10 waarden van de volgende reeks print.

```
0 2 4 6 8 10 ...
```

7.2 Schrijf een algoritme dat de eerste 12 waarden van de volgende reeks print.

```
1 3 5 7 9 11 ...
```

7.3 Schrijf een algoritme dat de eerste 7 waarden van de volgende reeks print.

```
4 5 6 7 8 9 ...
```

7.4 Schrijf een algoritme dat de eerste 9 waarden van de volgende reeks print.

```
4 8 12 14 16 ...
```

7.5 Schrijf een algoritme dat de eerste 10 waarden van de volgende reeks print.

```
0 1 4 9 16 25 36 ...
```

7.6 Schrijf een algoritme dat de eerste 15 waarden van de volgende reeks print.

```
1 2 5 10 17 26 37 ...
```

7.7 Schrijf een algoritme dat de eerste 9 waarden van de volgende reeks print.

```
5 4 3 2 1 0 -1 -2 -3 ...
```

Standaard-reeksen Met een loop kun je reeksen getallen printen. Met de volgende `for`-loop printen we de getallen 0 tot en met 9:

```
for(i = 0; i < 10; i++)
    print(i)
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Om andere reeksen te printen moet je achterhalen wat de regelmaat in de reeks is. Vaak kun je je loop dan baseren op het voorbeeld hierboven. Wil je bijvoorbeeld tien getallen uit de reeks $0, 4, 8, 12, 16, ...$ printen, dan zou je kunnen opmerken dat elk van die getallen steeds vier keer zo groot is als het getal uit de reeks hierboven. Dus kunnen we ons programma zo aanpassen:

```
for(i = 0; i < 10; i++)
    print(i * 4)
```

```
0, 4, 8, 12, 16, 20, 24, 28, 32, 36
```

Voor de reeks $1, 2, 3, 4, ...$ kunnen we het volgende programma gebruiken. Vergelijk het weer met het eerste programma hierboven:

```
for(i = 0; i < 10; i++)
    print(i + 1)
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Aantal stappen Je kunt de lengte van de reeks variëren door de conditie van de loop zelf aan te passen. Met `i < 10` printen we 10 getallen, maar dat kan net zo goed een ander aantal zijn.

Traceren Om te controleren of de programma's kloppen, kun je een trace maken. Je hoeft waarschijnlijk niet de hele loop uit te schrijven; een aantal stappen is genoeg, zolang je kunt verifiëren dat de eerste paar getallen precies kloppen.

7.2 More complex sequences

7.8 Schrijf een algoritme dat de eerste 12 waarden van de volgende reeks print.

```
1 2 4 8 16 ...
```

7.9 Schrijf een algoritme dat de eerste 7 waarden van de volgende reeks print.

```
1 3 9 27 81 ...
```

7.10 Schrijf een algoritme dat de eerste 10 waarden van de volgende reeks print.

```
2 4 6 8 10 ...
```

7.11 Schrijf een algoritme dat de eerste 9 waarden van de volgende reeks print.

```
16 8 4 2 1 0 0 ...
```

7.12 Schrijf een algoritme dat de eerste 10 waarden van de volgende reeks print.

```
1000 100 10 1 0 0 ...
```

7.13 Schrijf een algoritme dat de eerste 7 waarden van de volgende reeks print.

```
21 10 5 2 1 0 0 ...
```

7.14 Schrijf een algoritme dat de eerste 9 waarden van de volgende reeks print.

```
5 4 3 2 1 0 -1 -2 -3 ...
```

Standaard-reeksen De volgende reeks is niet zomaar af te leiden uit de reeksen in het vorige hoofdstuk:

```
1 2 4 8 16 32 ...
```

Elk getal van de reeks is het voorgaande getal, keer 2. Om de reeks te genereren, introduceren we daarom een aparte variabele `getal`, waarin we de tussenstand van de berekening bijhouden. De tussenstand vermenigvuldigen we steeds met 2, en dan printen we het resultaat:

```
getal = 1
for(i = 0; i < 10; i++)
    print(getal)
    getal = getal * 2
```

Vergeet niet je uitwerkingen te traceren.

7.3 Filtering sequences

7.15 Schrijf een algoritme dat de eerste 14 waarden van de volgende reeks print.

```
1 2 * 4 5 * 7 8 * 10 ...
```

7.16 Schrijf een algoritme dat de eerste 12 waarden van de volgende reeks print.

```
1 2 3!  4 5 6!  7 8 9!  10 ...
```

7.17 Schrijf een algoritme dat de laatste 12 waarden van de volgende reeks print.

```
...  0 16 14 0 10 8 0 4 2 0
```

7.18 Schrijf een algoritme dat de eerste 12 waarden van de volgende reeks print.

```
1 2 # 8 16 # 64 128 # 512 ...
```

7.19 Schrijf een algoritme dat de eerste 12 waarden van de volgende reeks print.

```
1 2 2 8 16 5 64 128 8 512 ...
```

Filter Neem de volgende reeks:

```
* 2 4 * 8 10 * 14 16 * ...
```

De reeks lijkt erg op de eenvoudige standaardreeks, maar elk getal dat deelbaar is door 3, is vervangen door een `*`. Om deze reeks te genereren, gebruiken we een `if-else`:

```
for(i = 0; i < 10; i++)
    getal = i * 2
    if(getal % 3 == 0)
        print("*")
    else
        print(getal)
```

Chapter 8

Strings

8.1 Indexing into strings

Write down what is printed when running each of the following code fragments.

8.1

```
s = "Hello, world!"
print(s[0])
```

8.2

```
s = "Hello, world!"
print(s[4])
```

8.3

```
s = "Hello, world!"
print(s[5])
```

8.4

```
s = "Hello, world!"
print(s[6])
```

8.5

```
s = "Hello, world!"
print(s[11])
```

8.6

```
s = "Hello, world!"
print(s[10 + 1])
```

8.7

```
s = "Leibniz"
a = 2
print(s[a])
```

8.8

```
s = "von Neumann"
x = 5
print(s[x])
```

8.9

```
s = "Fourier"
foo = 3 * 2
print(s[foo])
```

8.10

```
s = "Fourier"
muz = 3
print(s[muz * 2])
```

8.11

```
s = "Riemann"
fol = 5
print(s[fol / 2])
```

8.12

```
s = "Laplace"
dal = 2
print(s[5 / dal + 1])
```

8.13

```
s = "Hamilton"
alb = length(s) - 4
print(s[alb * 2])
```

8.14

```
s = "Kolmogorov"
kul = length(s) - 1
print(s[kul / 2])
```

8.15

```
s = "Bernoulli"
lop = length(s)/2
print(s[lop])
```

8.16

```
s = "Wrap"
len = length(s)
print(s[2 % len])
```

8.17

```
s = "Wrap"
len = length(s)
print(s[3 % len])
```

8.18

```
s = "Wrap"
len = length(s)
print(s[4 % len])
```

Strings Een string bestaat uit losse *tekens* of *characters*. We definiëren een string `s` die bestaat uit 13 tekens:

```
s = "Hello, world!"
```

De tekens in zo'n string hebben een *positie* of *index*. Daarbij wordt altijd vanaf 0 geteld:

teken	`H`	`e`	`l`	`l`	`o`	`,`		`w`	`o`	`r`	`l`	`d`	`!`
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Indexering We kunnen de losse tekens van een string ophalen door te indexeren:

```
s = "Hello, world!"
print(s[1])
```
```
e
```

Dit stukje code print het onderdeel van de string `"Hello, world!"` dat zich bevindt op plaats 1. Omdat we vanaf 0 tellen, is dit de letter `e`, zoals in de tabel hierboven.

Variabelen als index We kunnen natuurlijk ook prima variabelen of expressies gebruiken als index van een string:

```
s = "Hello, world!"
i = 7
print(s[(i - 1) / 2]
```
```
l
```

Grenzen Lezen of schrijven buiten de grenzen van een string levert een *out of bounds*-fout op of een *segmentation fault*. Bijvoorbeeld bij het opvragen van de waarde op positie 10 van de string `"Error"`. Hou de grenzen dus goed in de gaten!

8.2 Repetition and strings

Write down what is printed when running each of the following code fragments. Make a trace-table if needed.

8.19

```
s = "Hello, world!"
i = 0
len = length(s)
while(i < len)
   print(s[i])
   i = i + 1
```

8.20

```
s = "Hello, world!"
i = 0
len = length(s)
while(i < len)
   print(s[i])
   i = i + 2
```

8.21

```
s = "Hello, world!"
i = length(s) - 1
while(i >= 0)
   print(s[i])
   i = i - 1
```

8.22

```
s = "Hello, world!"
l = length(s)
for(i = 0; i < l; i = i + 1)
   print(s[i])
```

8.23

```
s = "guret"
l = length(s)
for(i = l - 1; i >= 0; i = i - 1)
   print(s[i])
```

8.24

```
s = "!yenskle!s"
l = length(s)
for(i = 1; i < l; i = i * 2)
   print(s[i])
```

8.25

```
s = "ok craab  borg"
for(i = 13; i >= 0; i = i - 2)
   print(s[i])
```

8.26

```
s = "inn dreaxx"
for(i = 0; i < 5; i = i + 1)
   print(s[i * 2])
```

8.27

```
s = "args"
for(i = 0; i < 6; i = i + 1)
   print(s[i / 2])
```

8.28

```
s = "hi!"
len = length(s)
for(i = 0; i < 6; i = i + 1)
   print(s[i % len])
```

8.29

```
s = "h!i"
len = length(s)
for(i = 0; i < 12; i = i + 2)
   print(s[i % len])
```

Herhaling en strings Je kan strings natuurlijk ook combineren met loops.

```
1 s = "Hsopil"
2 i = 0
3 len = length(s)
4 while(i < len)
5     print(s[i])
6     i = i + 2
```

Op regel 3 wordt het commando `length` gebruikt om de lengte van de string `s` te bepalen (in dit geval 6). Vrijwel alle programmeertalen hebben zo een ingebouwde opdracht.

Als we willen weten wat er geprint wordt tijdens het uitvoeren van deze code kunnen we, net als in eerdere hoofdstukken, een trace maken. Voor dat we dat doen, is het handig om eerste een tabel te maken met de indices van string `s`:

index	0	1	2	3	4	5
karakter	H	s	o	p	i	l

Trace In de trace willen we bijhouden wat de waarde van de variabele `i` is en wat er geprint wordt. Voor het gemak houden we ook de waarde van `s[i]` bij. Om de trace een beetje compact te houden laten we de waardes van de variabelen `s = "Hsopil"` en `len = 6` buiten beschouwing. Deze veranderen toch niet gedurende de loop.

regel	1	2	3	4	5	6	4	5	6	4	5	6	4
var i		0	0	0	0	2	2	2	4	4	4	6	6
s[i]		H	H	H	H	o	o	o	i	i	i		
print					H			o			i		

Deze code print dus de text `Hoi` naar het scherm.

8.3 More repetition with strings

Write down what is printed when running each of the following code fragments. Make a trace-table if needed.

8.30

```
s1 = "Hlo ol"
s2 = "el,wrd"
i = 0
while(i < 6)
   print(s1[i])
   print(s2[i])
   i = i + 1
```

8.31

```
s1 = "asedm"
s2 = "!artm"
i = 0
while(i < 5)
   print(s1[i])
   print(s2[5 - i])
   i = i + 1
```

8.32

```
i = 0
s = "srblcabe"
while(i < 4)
   print(s[i])
   print(s[i + 4])
   i = i + 1
```

8.33

```
s1 = "ab"
s2 = "123"
j = 1
for(i = 0; i < 2; i = i + 1)
   print(s1[i])
   print(s2[j])
   j = j + 1
```

8.34

```
s1 = "ab"
s2 = "123"
for(i = 0; i < 3; i = i + 1)
   print(s1[i / 2])
   print(s2[i])
```

8.35

```
s = "args"
for(i = 0; i < 6; i = i + 1)
   print(s[i / 3])
```

8.36

```
s = "hi!"
for(i = 0; s[i] != '!'; i = i + 1)
   print(s[i])
```

Chapter 9

Arrays

9.1 Arrays

Write down what is printed when running each of the following code fragments. Make a trace-table if needed.

9.1

```
rop = [8, 5, 4]
print(rop[1])
```

9.2

```
fifi = [0.1, 3.1, 2.1]
print(fifi[2])
```

9.3

```
bol = ['x', 'b', 'g']
print(bol[0])
```

9.4

```
p = [16, 2, 0]
print(p[2 - 1])
```

9.5

```
aaa = [4, 2, 0]
print(aaa[10 % 3])
```

9.6

```
yup = [0.4, 0.3, 0.1]
print(yup[1 * 2])
```

9.7

```
uil = [8, 2, 0]
print(uil[2] - 1)
```

9.8

```
lala = [4, 10, 0]
print(lala[1] % 3)
```

9.9

```
tir = [0.4, 0.3, 0.1]
print(tir[1] * 2)
```

9.10

```
ala = [1, 2, 0]
i = 2
print(ala[i])
```

9.11

```
jul = [4.0, 2.5, 8.1]
a = 1
print(jul[a + 1])
```

9.12

```
j = [1.5, 1.2, 3.1]
vi = 2
print(j[vi] * 2)
```

9.13

```
u = [10, 14, 6]
t = 2
print(u[(t + 5) % 3])
```

9.14

```
hi = [4, 3, 2]
x = 1
print(hi[x * 2] + 2)
```

9.15

```
f = [14, 12]
l = 2
print(f[l + 11 % 2] % 6)
```

9.16

```
l = [13, 11]
i = 0
print(l[i] - l[i + 1])
```

9.17

```
lom = [11, 12, 13]
i = lom[0] % 3
print(lom[i] - 10)
```

9.18

```
s = "Wrap"
len = length(s)
print(s[4 % len])
```

Arrays In het volgende codefragment wordt een array van integers gedefiniëerd:

```
ooy = [5, 6, 7]   →  array van integers
print(ooy[0])     →  print '5'
print(ooy[1])     →  print '6'
print(ooy[2])     →  print '7'
```

De eerste regel maakt een array met de getallen `5`, `6` en `7` en kent deze toe aan de variabele `ooy`. De regels eronder printen vervolgens de individuele elementen uit deze array. Let op, net als bij strings beginnen de indices van arrays met tellen bij `0`.

Types Zoals je ziet, lijken arrays erg op strings. Dat is ook niet zo gek, een string is niets anders dan een array van karakters. Het grote verschil is dat we in een array getallen kunnen opslaan. We kunnen dus een array van integers of floating point getallen maken. De voorbeelden hieronder laten verschillende type arrays zien.

```
ooy = [5, 6, 7]          →  array van integers
lia = [5.0, 6.0, 7.0]    →  array van floats
aps = ['e', 'f', 'g']    →  array van karakters (in veel talen equivalent aan een string)
```

Indices Je kan natuurlijk gewoon variabelen gebruiken voor de indices:

```
elk = [1, 2, 3]
i = 1
print(elk[i])
```

print: 2

En, net als met strings, is het mogelijk om hele berekeningen te gebruiken als index:

```
das = [1, 2, 3]
i = 1
print(das[(i + 1) / 2])
```

print: 2

9.2 Mutating arrays

Evaluate the following code fragments and write down the final value for all variables.

9.19

```
lop = [10, 20, 30]
lop[1] = 5
```

9.20

```
kol = ['a', 'o', 'i']
kol[0] = 'h'
```

9.21

```
fof = [0.1, 0.2, 0.3]
fof[2] = 1.0
```

9.22

```
dido = [20, 15, 10]
i = 0
dido[i] = 2
```

9.23

```
l = [5, 1, 2]
hil = 2
l[hil] = hil
```

9.24

```
l = [0.1, 0.2, 0.3]
hihi = 10 / 5
l[hihi / 2] = 1.0
```

9.25

```
yoyo = [2.0, 1.5, 1.0]
i = 0
yoyo[i] = yoyo[i + 1]
```

9.26

```
yoyo = [2.0, 1.5, 1.0]
i = 0
yoyo[i + 1] = yoyo[i]
```

9.27

```
yoyo = [2.0, 1.5, 1.0]
yoyo[0] = yoyo[1]
yoyo[1] = yoyo[2]
```

9.28

```
hipp = [42, 3, 101]
y = 1
hipp[y] = hipp[y] + 1
```

9.29

```
fle = [7, 6, 5]
s = 2
fle[s] = fle[s] * 2
```

9.30

```
k = [3.4, 2.0, 0.6]
m = 1
k[m - 1] = k[m + 1] / 2
```

9.31

```
r = [0, 0, 0, 0]
p = r[0]
r[p + 1] = r[p] + 1
```

9.32

```
zi = ['a', 'b', 'c']
p = 2
zi[(p + 1) % 3] = 's'
```

9.33

```
b = [false, true, false]
l = 0
b[l + 2] = b[0] || b[1]
```

9.34

```
rol = [1, 2, 3, 4]
o = 3
rol[o] = rol[o] % 4
```

9.35

```
j = [0, -1, -2]
v = j[1] + 2
j[v] = j[v] * 2
```

9.36

```
r = [4, 2, 0]
r[r[0] / 2] =
   r[r[2 / 2] / 2] / 2
```

Aanpassen Arrays zijn modificeerbaar. Dat wil zeggen dat als we eenmaal een array hebben gemaakt, we later nog aanpassingen kunnen doen. Neem het volgende voorbeeld:

```
lan = [0, 0, 0]
lan[1] = 5
```

Eerst maken we een array met drie nullen. Daarna maken we van het eerste element van de array een `5`. Na het uitvoeren van dit codefragment is array `lan` $\rightarrow$ `[0, 5, 0]`.

9.3 Repetition and arrays

Write down what is printed when running each of the following code fragments.

9.37

```
klo = [1, 2, 3, 4, 5]
l = length(klo)
for(i = 0; i < l; i = i + 1)
   print(klo[i])
```

9.38

```
vlo = [1, 2, 3, 4, 5]
l = length(vlo)
for(i = l - 1; i >= 0; i = i - 1)
   print(vlo[i])
```

9.39

```
tik = [1, 2, 3, 4, 5]
i = 0
while(tik[i] < 4)
   print(tik[i])
   i = i + 1
```

9.40

```
hal = [10, 20, 30]
i = 2
while(hal[i] / 10 > 1)
   print(i)
   i = i - 1
```

Evaluate the following code fragments and write down the final value for all variables.

9.41

```
lap = ['a', 'b', 'c', 'd', 'e']
l = length(lap)
for(i = l - 1; i >= 0; i = i - 1)
   lap[i] = 'x'
```

9.42

```
gop = [4, 2, 5, 1, 8]
l = length(gop)
for(i = 0; i < l - 1; i = i + 1)
   gop[i] = gop[i + 1]
```

9.43

```
gop = [4, 2, 5, 1, 8]
l = length(gop)
for(i = 0; i < l - 1; i = i + 1)
   gop[i] = gop[i] + 1
```

9.44

```
gop = [4, 2, 5, 1, 8]
l = length(gop)
for(i = 0; i < l - 1; i = i + 1)
   gop[i + 1] = gop[i]
```

Herhaling In de praktijk komen arrays veel voor in combinatie met loops.

```
1 eus = [0, 0, 0, 0]
2 for(i = 0; i < length(eus); i++)
3     eus[i] = i * 2
```

Na het uitvoeren van dit codefragment bevat array `eus` de waardes `0`, `2`, `4` en `6`. Deze code maakt gebruik van het commando `length()` om de lengte van de array te bepalen. Let erop dat niet alle programmeertalen zo'n commando hebben (bijvoorbeeld de programmeertaal C). In zulke gevallen zal je als programmeur zelf de lengte van een array moeten bijhouden.

Chapter 10

Functions

10.1 Functions that print

Write down what is printed when running each of the following code fragments.

10.1

```
void baz():
   print("fly")
baz()
```

10.2

```
void bar():
   print("jump")
```

10.3

```
void foo():
   print("fly")
void bar():
   print("jump")
foo()
bar()
bar()
```

10.4

```
void baz():
   print("jump")
void foo():
   print("bicycle")
baz()
```

10.5

```
void foo():
   print("pie")
void baz():
   foo()
   print("bicycle")
foo()
baz()
```

10.6

```
void baz():
   bar()
   print("rainbow")
void bar():
   print("fly")
baz()
```

10.7

```
void foo():
   baz()
   print("cake")
void baz():
   print("jump")
baz()
```

10.8

```
void bar():
   print("bicycle")
void baz():
   print("jump")
   bar()
baz()
```

Functiedefinitie Bij functies onderscheiden we de *definitie*, waarin wordt beschreven wat de functie moet doen, van de *aanroep*, die vertelt dat de functie daadwerkelijk moet worden uitgevoerd. Dat betekent meteen dat als we een functie definiëren de functie niet direct wordt uitgevoerd. Een functiedefinitie ziet er in dit boek als volgt uit:

```
<type> <naam van functie>():
  <inhoud van functie>
```

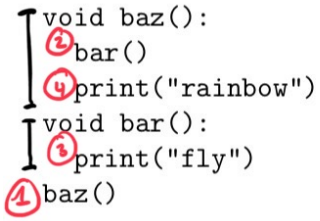
Voorlopig zullen we `void` invullen als `<type>`. In de praktijk is dat een functie die een opdracht uitvoert (zoals `print`). Een voorbeeld van een functie die iets doet:

```
void ding_printer():
  print("iets")
```

Functieaanroep Van zichzelf doet een definitie niet anders dan een functie definiëren onder een gestelde naam. Het aanroepen van de functie gebeurt dan als volgt:

```
<naam van functie>()
```

Traceren Functieaanroepen kun je traceren. We starten bij de allereerste regel die geen functiedefinitie is, maar een aanroep, en van daar stappen we naar de functie die wordt aangeroepen. Om duidelijk te maken welke functies er zijn, hebben we de definities gemarkeerd met een streep in de kantlijn:



Zo zie je in welke volgorde de twee `print`-statements worden uitgevoerd.

10.2 Parameters

Write down what is printed when running each of the following code fragments.

10.9

```
void hay(x):
   print(x)
hay("fly")
```

10.10

```
void dal(y):
   print(y)
dal("jump")
```

10.11

```
void eau(x, y):
   print(x / y)
eau(10, 12)
```

10.12

```
void pow(y, x):
   print(x / y)
pow(10, 12)
```

10.13

```
void gam(x, z):
   print(z / x)
gam(10, 12)
```

10.14

```
void zek(x, z):
   print(x / z)
zek(12, 10)
```

10.15

```
void eid(x, y, z):
   print(z / x)
eid(10, 11, 12)
```

10.16

```
void ash(y, x, z):
   print(x / z)
ash(12, 11, 10)
```

10.17

```
void bar(z, y, x):
   print(y / x * z)
bar(10, 11, 12)
```

10.18

```
void duo(x, z, y):
   print(y / z)
duo(12, 11, 10)
```

10.19

```
void oca(y, z, x):
   print(x / y)
oca(10, 11, 12)
```

10.20

```
void tug(z, x, y):
   print(x / z * y)
tug(12, 11, 10)
```

Parameters Functies hebben vaak parameters. Bij het aanroepen van de functie worden *concrete* waarden ingevuld voor deze parameters. Vanuit het perspectief van de functie, krijgen deze waarden namen die gespecificeerd staan in de *parameterlijst*. Zo hebben we deze definities:

```
void datum_1(dag, maand):
  print(dag)
  print(maand)
```

```
void datum_2(maand, dag):
  print(dag)
  print(maand)
```

In het eerste geval hebben we de parameters `dag` en `maand`. In het tweede geval `maand` en `dag`, in de omgekeerde volgorde. Deze volgorde heeft effect op hoe de meegegeven parameters behandeld worden. We roepen de functies aan in deze voorbeelden:

```
datum_1(21, 6)
```

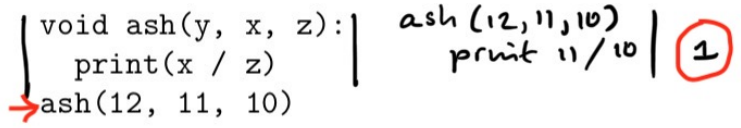
```
datum_2(6, 21)
```

De uitvoer is dan:

```
21
6
```

```
21
6
```

Traceren Het volgen van alle waarden bij een functie met parameters kan snel ingewikkeld worden. Daarom is het ook nu handig te traceren. Allereerst strepen we de functies aan (in dit geval ééntje). Dan markeren we de startregel met een pijltje.



Op de startregel staat een functieaanroep. Deze aanroep nemen we over rechts van de functiedefinitie. De functiedefinitie nemen we ook over, maar we vullen alle waarden zo **concreet** mogelijk in. De parameters `y`, `x` en `z` hebben in de concrete versie waarden gekregen, dus vullen we die ook in op de `print`-regel.

Tot slot blijft er nog een berekening over in het `print`-statement. Als we die evalueren, krijgen we het getal dat er geprint zal worden.

10.3 Calling functions with variables

Write down what is printed when running each of the following code fragments.

10.21

```
void foo(x):
   print(x)
inv = "fly"
foo(inv)
```

10.22

```
void bar(name):
   print(name)
foo = "skip"
bar("jump")
```

10.23

```
var = "bear"
void una(var):
   print(var)
una(var)
```

10.24

```
name = "pieter"
void greet(name):
   print("hello")
greet(name)
```

10.25

```
x = 3
y = 5
void foo(a, b):
   print(a / b)
foo(x, y)
```

10.26

```
x = 3
y = 5
void baz(y, x):
   print(x / y)
baz(x, y)
```

10.27

```
x = 10
void bar(x, z):
   print(z / x)
bar(x, 12)
```

10.28

```
var1 = 2
void dra(var1, var2):
   print(var1 + var2)
dra(var1, 8)
```

10.29

```
x = "fly"
var2 = "skip"
void shout(var1, var2):
   print(var1 + var2)
shout(x, "jump")
```

10.30

```
x = 3
a = 2
void magic(a, b):
   print(a - b)
magic(x, a)
```

Aanroepen met variabelen Functies kunnen ook worden aangeroepen met variabelen als concrete waarden voor de parameters. Zo hebben we bijvoorbeeld de volgende functie:

```
void twice(text):
  print(text)
  print(text)
```

Als we deze functie als volgt aanroepen:

```
fruits = "pineapple blueberry"
twice(fruits)
```

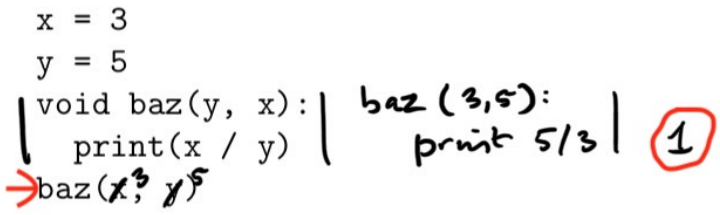
dan krijg je twee keer de tekst `pineapple blueberry` op je scherm geprint. De waarde van de variabele `fruits`, dat is de string `"pineapple blueberry"`, wordt namelijk meegegeven aan de functie `twice`. Op de regel met `twice(fruits)` wordt de huidige waarde van fruits ingevuld. Dit kan je je voorstellen alsof de regel veranderd wordt naar `twice("pineapple blueberry")`. Binnen de functie `twice` wordt de string `"pineapple blueberry"` toegewezen aan `text`. Daarna wordt die waarde geprint.

Ook bij het aanroepen van functies met variabelen maakt de volgorde van de parameters uit:

```
x = 10
y = 40
void minus(x, y):
  print(x - y)
minus(y, x)
```

Omdat de waardes van de variabelen worden meegegeven aan de functie wordt hier `30` geprint en niet `-30`. Je kan de regel `minus(y, x)` vervangen door `minus(40, 10)`, ofwel de waardes van de variabelen `x` en `y` op dat moment. Binnen de functie wordt de waarde `40` aan de variabele `x` en de waarde `10` aan de variabele `y` toegewezen. Dan wordt er `print(x - y)` uitgevoerd, ofwel `print(40 - 10)`.

Traceren We voegen nu expliciet een element aan de trace toe: het vervangen van de waarden in de functieaanroep. Bij de startregel strepen we de variabelenamen door, en vervangen deze door de waarden die eerder toegekend zijn. (Hierbij letten we nog helemaal *niet* op de functiedefinitie!)



Nu de concrete waarden ingevuld zijn, kunnen we de trace verder opschrijven zoals in de vorige paragraaf.

Chapter 11

Functions that calculate

11.1 Functions that return something

Write down what is printed when running each of the following code fragments.

11.1

```
string foo(x):
   return x
print(foo("hard"))
```

11.2

```
string bar(name):
   return "jump"
print(bar("hi"))
```

11.3

```
int foo(x, y):
   return x / y
print(foo(10, 12))
```

11.4

```
int baz(y, x):
   return x / y
print(baz(10, 12))
```

11.5

```
amd = 4
float oei(dam, mad):
   return dam * mad + amd
sim = oei(amd, amd)
print(sim)
```

11.6

```
x = 1
int una(x):
   return x + 1
int zab(y):
   return y - 1
print(zab(2) + una(x))
```

11.7

```
string mak(x, y, z):
   return y
mis = "cal"
res = mak("sol", mis, "toa")
print(res)
```

11.8

```
void una(x, z):
   print(x / z)
int kam(y):
   return y * 3
print(una(kam(4), 10))
```

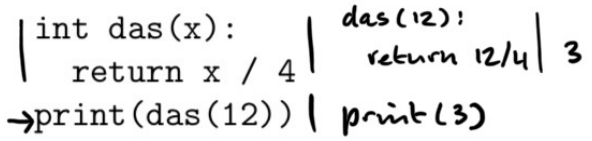
Retourneren *Retourneren* is een ander woord voor *teruggeven*. Het is mogelijk voor functies om een waarde terug te geven naar vanwaar de functie wordt uitgevoerd. Dit doe je met het commando `return`. De waarde die hier teruggegeven wordt noemen we ook wel het *resultaat* van de functie. Het resultaat van de functie vervangt als het ware de functieaanroep zelf. Dit werkt net als het gebruik van variabelen; die worden vervangen door hun waardes.

Types Helemaal in het begin van dit hoofdstuk lieten we het volgende zien:

```
<type> <naam van functie>():
  <inhoud van functie>
```

Tot nog toe hebben we alleen `void` in de plaats van `<type>` gebruikt. `void` staat voor `niets` en dit betekent dat onze functie niets teruggeeft. Dan hoef je dus ook geen `return` commando te geven. Als je wel iets wil teruggeven moet je het specifieke `type` aanduiden. In onze voorbeelden zullen we alleen de types `int`, `float`, en `string` gebruiken.

Traceren Functies die iets uitrekenen traceren we op vrijwel dezelfde manier als voorheen. Daar komt bij dat er iets *terug*gaat van de functie naar de startregel. In het volgende voorbeeld zie je dat de functie het getal 3 uitrekent en `returnt`. Dit kunnen we uiteindelijk invullen in de `print`.



11.2 Functies that call other functions

Write down what is printed when running each of the following code fragments.

11.9

```
int gus(z, y)
   return dim(z, y) / 2
int yap(z, x)
   return gus(x, z) - 3
int dim(y, x)
   return x + 4
x = 6
y = 4
print(yap(x, y))
```

11.10

```
int uru(x, y)
   return 2 * y
int sal(z, y)
   return 4 / uru(y, z)
int hit(z, x)
   return 2 + sal(z, x)
y = 5
x = 6
print(hit(x, y))
```

11.11

```
int dip(y, x)
   return y * 4
int mux(y, z)
   return 5 / dip(y, z)
int bad(x, z)
   return mux(z, x) + 3
y = 4
x = 5
print(bad(x, y))
```

11.12

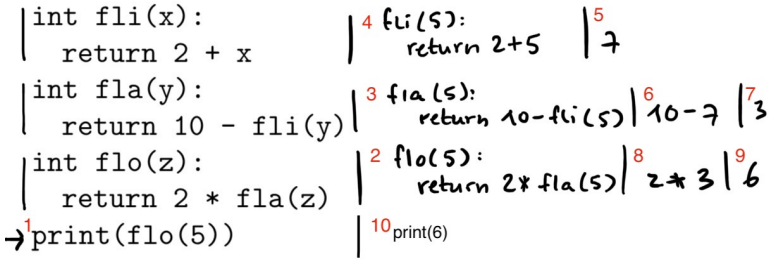
```
int loo(y, z)
   return z / 3
int aam(z, y)
   return loo(y, z) * 6
int wah(y, z, x)
   return aam(y, z) - 2
z = 2
x = 5
y = 4
print(wah(x, y, z))
```

11.13

```
int nae(z, y)
   return 2 * y
int bus(x, z)
   return nae(z, x) + 3
int ann(z, y, x)
   return bus(y, z) / 2

y = 5
x = 3
z = 6
print(ann(x, y, z))
```

Traceren Als er sprake is van meerdere functies die iets uitrekenen, dan kan het traceren er als volgt uitzien.



Met cijfers is aangegeven in welke volgorde de trace ingevuld wordt.

Answers

Exercises

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
a. 0	a. 1	a. 3	a. 0	a. 2	a. 0	a. 0	a. 3	a. 1	a. 3
b. 0	b. 2	b. 3	b. 1	b. 3	b. 0	b. 2	b. 0	b. 1	b. 2
c. 0	c. 3	c. 2	c. 1	c. 4	c. 0	c. 0	c. 8	c. 0	c. 5
d. 0	d. 4	d. 1	d. 1	d. 2	d. 0	d. 0	d. 1	d. 1	d. 8
e. 0	e. 5	e. 1	e. 3	e. 5	e. 0	e. 0	e. 5	e. 0	e. 15

1.11	1.12	1.13	1.14	1.15	1.16	1.17	1.18	1.19	1.20
a. 0	a. 0	a. 61	a. 6	a. 1	a. 0	a. 0	a. 50	a. -5	a. -5
b. 90	b. 3	b. -8	b. -2	b. 3	b. -2	b. -6	b. 19	b. 102	b. 125
c. 3	c. 0	c. 7	c. 5	c. 2	c. 40	c. 1	c. -1	c. -7	c. 48
d. 0	d. 0	d. 10	d. 17	d. -1	d. 0	d. 107	d. -8	d. -7	d. -64
e. 0	e. 1	e. 3	e. 2	e. 72	e. -6	e. -3	e. 4	e. 70	e. -96

1.21	1.22	1.23	1.24	1.25
a. -0.5	a. -1.0	a. -0.25	a. 0.0	a. 2
b. 1.0	b. 1.25	b. 0.5	b. 1.25	b. 0.0
c. 2.5	c. 1.5	c. 2.0	c. 1.0	c. 0.25
d. 1.0	d. 1.0	d. -0.5	d. 1.5	d. 1.0
e. 1.0	e. 3.5	e. 4.5	e. 1.5	e. 1.5

1.26

a. 3
b. -3.0
c. 43.5
d. -5.5
e. 0.25

1.27

a. 8.0
b. 1.5
c. 4.0
d. 6
e. 5.5

1.28

a. -4.5
b. -13
c. 2.0
d. 3
e. 0.5

1.29

a. 5.5
b. -2.5
c. 6.0
d. -3.75
e. 3.0

1.30

a. 3.0
b. 10.0
c. 6.0
d. 3.0
e. 16.5

2.1

a. FALSE
b. FALSE
c. FALSE
d. FALSE
e. TRUE

2.2

a. TRUE
b. FALSE
c. FALSE
d. TRUE
e. FALSE

2.3

a. FALSE
b. TRUE
c. FALSE
d. FALSE
e. FALSE

2.4

a. FALSE
b. TRUE
c. FALSE
d. TRUE
e. FALSE

2.5

a. FALSE
b. FALSE
c. FALSE
d. FALSE
e. FALSE

2.6

a. TRUE
b. FALSE
c. FALSE
d. FALSE
e. FALSE

2.7

a. TRUE
b. FALSE
c. TRUE
d. TRUE
e. FALSE

2.8

a. TRUE
b. FALSE
c. TRUE
d. FALSE
e. FALSE

2.9

a. FALSE
b. TRUE
c. FALSE
d. FALSE
e. FALSE

2.10

a. TRUE
b. FALSE
c. FALSE
d. FALSE
e. FALSE

2.11

a. TRUE
b. TRUE
c. FALSE
d. FALSE
e. FALSE

2.12

a. TRUE
b. TRUE
c. FALSE
d. TRUE
e. TRUE

2.13

a. FALSE
b. FALSE
c. TRUE
d. TRUE
e. FALSE

2.14

a. FALSE
b. TRUE
c. TRUE
d. TRUE
e. FALSE

2.15

a. FALSE
b. FALSE
c. TRUE
d. TRUE
e. TRUE

2.16

a. TRUE
b. FALSE
c. TRUE
d. TRUE
e. TRUE

2.17

a. TRUE
b. TRUE
c. TRUE
d. FALSE
e. FALSE

2.18

a. FALSE
b. FALSE
c. FALSE
d. FALSE
e. TRUE

2.19

a. TRUE
b. TRUE
c. FALSE
d. TRUE
e. FALSE

2.20

a. TRUE
b. TRUE
c. TRUE
d. FALSE
e. FALSE

3.1

```
dey = 3.5
luo = 1.5
```

3.2

```
gog = 3
oer = 1.5
```

3.3

```
eli = 1.0
soc = 4.0
```

3.4

```
jon = -1.0
aus = 1.0
```

3.5

```
abe = 1.5
sir = 3.0
```

3.6

```
vow = 3.0
nub = 4.0
```

3.7

```
gup = 0
ley = 0
```

3.8

```
tez = 2.5
nap = 7.5
```

3.9

```
jib = 5
```

3.10

```
aus = 1.5
```

3.11

```
bam = 2.0
```

3.12

```
tye = 1
```

3.13

```
nam = 4
sob = 0.0
off = 4
```

3.14

```
kaw = 1.0
zan = 1.5
```

3.15

```
quo = -0.75
ora = -2.75
```

3.16

```
gup = 2.5
han = 3.5
```

3.17

```
lod = 5
nid = 6.0
mou = 18
gyp = -2.5
```

3.18

```
you = 3
way = 0.5
iwa = 1.5
```

3.19

```
fei = 2
oka = 6.0
```

3.20

```
fae = 0.5
```

4.1

```
a = 11
```

4.2

```
a = 0
```

4.3

```
a = 0
```

4.4

```
a = 0
```

4.5

```
a = 10
```

4.6

```
a = 10
```

4.7

```
a = 18
```

4.8

```
a = 10
```

4.9

```
a = -3
```

4.10

```
a = 5
```

4.11

```
a = 1
```

4.12

```
a = 0
```

4.13

```
a = 20
```

4.14

```
a = 20
```

4.15

```
a = 9
b = 9
```

4.16

```
a = 0
b = 2
```

4.17

```
a = -1
b = -2
```

4.18

```
a = 0
b = -2
```

4.19

```
a = 6
b = 8
```

4.20

```
a = 4
b = -1
```

4.21

```
a = 10
```

4.22

```
a = -10
```

4.23

```
a = 7
```

4.24

```
a = 9
```

4.25

```
a = 8
```

4.26

```
a = 7
```

4.27

```
a = 8
```

4.28

```
a = 9
```

5.1

```
a = 6
```

5.2

```
a = 15
```

5.3

```
a = 16
```

5.4

```
a = 8
```

5.5

```
a = 1
```

5.6

```
a = 256
```

5.7

```
a = 18
```

5.8

```
a = 31
```

5.9

```
a = -31
```

5.10

```
a = 16
```

5.11

```
a = 5
```

5.12

```
a = 6
```

5.13

```
a = 3
```

5.14

```
a = 4
```

5.15

```
a = 5
```

5.16

```
a = 4
```

5.17

```
a = 9
```

5.18

```
a = 1
```

5.19

```
a = -2
```

5.20

```
a = 2
```

5.21

```
a = 2
b = 4
```

5.22

```
a = 2
b = 4
```

5.23

```
a = 2
b = 0
```

5.24

```
a = -2
b = 4
```

5.25

```
a = 5
b = 8
```

5.26

```
a = 10
b = 6
```

5.27

```
a = 4
b = 2
```

5.28

```
a = 6
b = 3
```

6.1

```
0 1 2
```

6.2

```
12 13 14
```

6.3

```
3 2 1 0
```

6.4

```
14 13 12 11
```

6.5

```
0 1 2
```

6.6

```
3 2 1 0
```

6.7

```
12 13 14
```

6.8

```
14 13 12 11
```

6.9

```
13 12 11 10 9
```

6.10

```
9 10 11
```

6.11

```
14 12 10 8
```

6.12

```
0 2 4 6 8
```

6.13

```
i = 3
a = 4
```

6.14

```
i = -1
a = 0
```

6.15

```
i = -1
a = 0
```

6.16

```
i = 3
a = 2
```

6.17

```
i = 3
a = 2
```

6.18

```
i = 16
a = 4
```

6.19

```
i = 8
a = 3
```

6.20

```
i = 6
```

6.21

```
b = 8
a = 5
while(a > 3)
   b = b / 2
   a = a - 1
```

6.22

```
e = 0
while(e < 4)
   print(e)
   e = e + 1
```

6.23

```
i = 4
while(i >= 0)
   print(i)
   i = i - 1
```

6.24

```
b = 0
i = 0
while(i < 4)
   b = i * 2
   i = i + 2
```

6.25

```
for(a = 1; a < 6; a = a + 2)
   print(a)
```

6.26

```
b = 0
for(a = 8; a > 1; a = a /2)
   b = b + 1
```

6.27

```
a = 8
for(b = 0; b < 3; b = b + 1)
   a = a / 2
```

6.28

```
for(a = 8, b = 0;
   a > 1 || b <= 3;
   a = a / 2, b = b + 1)
   print(a)
```

7.1

```
for(i = 0; i < 10; i++)
   print(i * 2)
```

7.2

```
for (i = 0; i < 12; i++)
   print(getal * 2 + 1)
```

7.3

```
for (i = 0; i < 7; i++)
   print(getal + 4)
```

7.4

```
for (i = 0; i < 9; i++)
   print(getal * 4 + 4)
```

7.5

```
for(i = 0; i < 10; i++)
   print(i * i)
```

7.6

```
for(i = 0; i < 15; i++)
   print(i * i + 1)
```

7.7

```
for(i = 0; i < 9; i++)
   print(i * -1 + 5)
```

7.8

```
getal = 1
for(i = 0; i < 12; i++)
   print(getal)
   getal = getal * 2
```

7.9

```
getal = 1
for(i = 0; i < 7; i++)
   print(getal)
   getal = getal * 3
```

7.10

```
getal = 2
for(i = 0; i < 10; i++)
   print(getal)
   getal = getal + 2
```

7.11

```
a = 16
for(i = 0; i < 9; i++)
   print(a)
   a = a / 2
```

7.12

```
b = 1000
for(i = 0; i < 10; i++)
   print(b)
   b = b / 10
```

7.13

```
c = 21
for(i = 0; i < 7; i++)
   print(c)
   c = c / 2
```

7.14

```
getal = 1
for(i = 0; i < 14; i++)
   if(getal % 3 == 0)
        print("*")
   else
        print(getal)
   getal = getal + 1
```

7.15

```
getal = 1
for(i = 0; i < 12; i++)
   if(getal % 3 == 0)
        print(getal + "! ")
   else
        print(getal)
   getal = getal + 1
```

7.16

```
getal = 34
for(i = 0; i < 20; i++)
   if(getal % 6 == 0)
        print(0)
   else
        print(getal)
   getal = getal - 2
```

7.17

```
getal = 1
for(i = 1; i <= 12; i++)
   if(i % 3 == 0)
        print("#")
   else
        print(getal)
   getal = getal * 2
```

7.18

```
getal = 1
for(i = 0; i < 12; i++)
   if((i + 1) % 3 == 0)
        print(i)
   else
        print(getal)
   getal = getal * 2
```

8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.8	8.9
H	o	,	(space)	d	d	i	e	r

8.10	8.11	8.12	8.13	8.14	8.15	8.16	8.17	8.18
r	e	l	bounds err	o	o	a	p	W

8.19
```
Hello, world!
```

8.20
```
Hlo ol!
```

8.21
```
!dlrow ,olleH
```

8.22
```
Hello, world!
```

8.23
```
terug
```

8.24
```
yes!
```

8.25
```
go back
```

8.26
```
index
```

8.27
```
aarrgg
```

8.28
```
hi!hi!
```

8.29
```
hi!hi!
```

8.30
```
Hello, world
```

8.31
```
bounds err
```

8.32
```
scrabble
```

8.33
```
a2b3
```

8.34
```
a1a2b3
```

8.35
```
aaarrr
```

8.36
```
hi
```

9.1
```
5
```

9.2
```
2.1
```

9.3
```
x
```

9.4
```
2
```

9.5
```
2
```

9.6
```
0.1
```

9.7
```
-1
```

9.8
```
1
```

9.9
```
0.6
```

9.10
```
0
```

9.11
```
8.1
```

9.12
```
6.2
```

9.13
```
14
```

9.14
```
4
```

9.15
```
bounds err
```

9.16
```
2
```

9.17
```
3
```

9.18
```
3
```

9.19
```
lop = [10, 5, 30]
```

9.20
```
kol = [‘h’, ‘o’, ‘i’]
```

9.21
```
fof = [0.1, 0.2, 1.0]
```

9.22
```
dido = [2, 15, 10]
```

9.23
```
l = [5, 1, 2]
```

9.24
```
l = [0.1, 1.0, 0.3]
```

9.25
```
yoyo = [1.5, 1.5, 1.0]
```

9.26
```
yoyo = [2.0, 2.0, 1.0]
```

9.27
```
yoyo = [1.5, 1.0, 1.0]
```

9.28
```
hipp = [42, 4, 101]
```

9.29
```
fle = [7, 6, 10]
```

9.30
```
k = [0.3, 2.0, 0.6]
```

9.31
```
r = [0, 1, 0, 0]
```

9.32
```
zi = [‘s’, ‘b’, ‘c’]
```

9.33
```
b = [false, true, true]
```

9.34

```
rol = [1, 2, 3, 0]
```

9.35

```
j = [0, -2, -2]
```

9.36

```
r = [4, 2, 1]
```

9.37

```
1 2 3 4 5
```

9.38

```
5 4 3 2 1
```

9.39

```
1 2 3
```

9.40

```
2 1
```

9.41

```
lap = [‘x’, ‘x’, ‘x’, ‘x’, ‘x’]
```

9.42

```
gop = [2, 5, 1, 8, 8]
```

9.43

```
gop = [5, 3, 6, 2, 8]
```

9.44

```
gop = [4, 4, 4, 4, 4]
```

10.1

```
fly
```

10.2

```

```

10.3

```
fly jump jump
```

10.4

```
jump
```

10.5

```
pie pie bicycle
```

10.6

```
fly rainbow
```

10.7

```
jump
```

10.8

```
jump bicycle
```

10.9	10.10	10.11	10.12	10.13	10.14
fly	jump	0	1	1	1

10.15	10.16	10.17	10.18	10.19	10.20
1	1	0	0	1	0

10.21	10.22	10.23	10.24	10.25	10.26
fly	jump	bear	hello	0	1

10.27	10.28	10.29	10.30
1	10	fly jump	1

11.1	11.2	11.3	11.4	11.5	11.6
hard	jump	0	1	20	3

11.7	11.8
cal	1

11.9	11.10	11.11	11.12	11.13
2	2	3	4	6