Programming Basics

Editor: Martijn Stegeman

January 18, 2020

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 operations

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`

Logic operations The values `true` and `false`, or any boolean expressions, may be combined into more complex expressions using logic operations. We introduce three.

And This operation, written as `&&`, requires both operands (TODO) to be `true` in order to yield `true` itself. The behavior of the operator can be described using a truth table:

expression	yields
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Or This operation is written as `||`, and requires only one of the operands to be `true` in order to yield `true`.

expression	yields
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Not A boolean expression's value can be negated, which is expressed using an exclamation mark `!`.

expression	yields
`!true`	`false`
`!false`	`true`

Precedence The logic operations also have their place in the precedence hierarchy:

1. first, expressions in parentheses
2. then arithmetic operations like + en *
3. then boolean operations like >, <, ==, !=, >=, <=
4. then `!`
5. then `||`
6. and finally `&&`

(Note that unlike * and /, the **and** and **or** operations are not in the same group.)

2.3 Writing: conditions

In the previous sections, you have *evaluated* expressions, calculating the outcome by applying rules. Here, we ask you to write expressions yourself. This is a more creative endeavour, and it may be hard to find the "right idea". In that case, have a look at the previous sections for inspiration.

When writing such an expression, you can check it by applying the rules again. Write down your expression and try to evaluate it. You can even come up with a couple of concrete "test cases", filling in the number `n` to see if the result is what you expect.

2.21 Write an expression that tests if a number `n` is smaller than 5.

2.22 Write an expression that tests if a number `n` is between 5 and 10 (both 5 and 10 included!).

2.23 Write an expression that tests if a number `n` is divisible by 3.

2.24 Write an expression that tests if a number `n` is *even*.

2.25 Write an expression that tests if a number `n` is *odd*.

2.26 For which values does the expression `!(n > 1 && !(n > 3)) && true` yield `true`?

2.27 The expression from the previous question may be written in more simply. Do this, and check if your simplified version is indeed equivalent to the original by checking if both yield the same value when substituting different values for `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`	$\boxed{3.14}$		
`dev = 3 / 4`	3.14	$\boxed{0}$	
`wan = 0.75`	3.14	0	$\boxed{0.75}$

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`	$\boxed{\cancel{1}}$
`wei = 4`	$\boxed{4}$

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`	$\boxed{1}$	
`say = hat + 4`	1	$\boxed{5}$

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

3.2 Writing: algorithms for swapping

Swapping the values of variables We will now see how to combine a number of assignments to develop an *algorithm*. An algorithm is a general procedure that a computer can execute. It is general because it uses a number of variables that may be assigned numbers or other data depending on the situation. Let's develop an algorithm to swap the values of two variables,

`a = 54` and `b = 29`.

As a first intuition, you might think to assign the value of `b` to `a`, and the value of `a` to `b`. An algorithm might then look like this:

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

Let's trace the algorithm. We take two starting values for `a` and `b`, and then we show the effect on these two variables *line by line*. Like in the previous section, we use a box to show which variable is changed on which line of code.

	a	b
	54	29
`a = b`	$\boxed{29}$	29
`b = a`	29	$\boxed{29}$

Now note that this is not the intended effect of the algorithm! Instead of swapping, the value of `a` is overwritten with **b**'s original value, and now both contain the same number.

The problem is the following: on line three, we would like to assign `b` the value of `a`, but at that point, `a` has already been overwritten. From this analysis we may have an intuition for a solution: we need to make sure that on line 3, the original value of `a` is still available. To do this, we need to introduce a third variable, which can "save" the value of `a` before it is overwritten.

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

Tracing that modified program will look like this:

	a	b	t
	54	29	
`t = a`	54	29	$\boxed{54}$
`a = b`	$\boxed{29}$	29	54
`b = t`	29	$\boxed{54}$	54

Indeed, this is the expected result for swapping!

3.21 Write an algorithm that swaps the values of three variables, as illustrated below. After executing the algorithm, the old value of `a` should be in `b`, the old value of `b` should be in `c`, and the old value of `c` should be in `a`.

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r] & c \arrow[ll, bend left]
\end{tikzcd}$$

Note: you might substitute concrete numbers for `a`, `b` and `c` to think about this problem, but this is not necessary to create a generally useful algorithm.

3.22 Write an algorithm that swaps the values of four variables, as illustrated below.

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r] & c \arrow[r] & d \arrow[lll, bend left]
\end{tikzcd}$$

3.23 Write an algorithm that swaps the values of four variables, as illustrated below.

$$\begin{tikzcd}
a \arrow[rr, bend left] & b \arrow[rr, bend left] & c \arrow[l, bend left] & d \arrow[lll, bend left]
\end{tikzcd}$$

Chapter 4

Conditional statements

4.1 Tracing conditional statements

Make a trace 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
```

Conditional statements The following code fragment contains a conditional statement, which starts with the keyword `if`. There is a requirement, or *condition*, which is `a < 10`. The statement on the following line has been indented (moved to the right) which will show that the statement is dependent on the `if`.

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

We start the program on line 1 with the value `a = 0`. This means that on line 2, the condition `a < 10` has indeed been met. The result is that the instruction `a = a + 10` will be executed too. Line 4 is printed all the way to the left, which means that it is not a part of the conditional statement: it will be executed regardless. This means that the program finishes with the variable `a` having a value `9`.

Tracing Like in the previous chapter we can trace the state of the variables. A trace for the previous code fragment looks like this:

	1	2	3	4
a	$\boxed{0}$	0	$\boxed{10}$	$\boxed{9}$

In the top row we print the program's line numbers. In the row below, we keep track of the variable `a`. In exactly three places, the value of `a` has a box: these are the lines where the value of the variable changed while tracing.

Failing condition In the code fragment below, we slightly changed the condition. In this case, at the moment of evaluating the condition, it yields `false`. Because the condition has not been met, the dependent line will be skipped. This means that the program finishes with the variable `a` having a value `-1`.

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

And we skip line 3 in the trace:

	1	2	4
a	$\boxed{0}$	0	$\boxed{-1}$

4.2 Tracing with multiple variables

Make a trace 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
```

Multiple dependent statements In the next code fragment, not one but two statements are dependent on the `if` condition:

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

The trace is quite like the one we did in the previous section:

	1	2	3	4
a	$\boxed{0}$	0	$\boxed{10}$	$\boxed{20}$

Multiple variables Here we again introduce the use of multiple variables in one program. This means that more information has to be tracked while tracing the program. Take a look at this fragment:

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

To trace this program, we add rows for each variable that is in the program. The variables `b` and `c` are not initialized until later, which is why some of the values are blank.

	1	2	3	4	5	6
a	$\boxed{1}$	1	1	1	$\boxed{0}$	0
b		$\boxed{0}$	0	0	0	$\boxed{1}$
c				$\boxed{1}$	1	1

4.3 Specifying cases

Make a trace 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
```

When the condition fails Below an `if` statement you might find an `else` statement. The two statements are then connected, and the `else` statement defines what should happen in case the condition in the `if` statement yields `false`.

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

Tracing the program above would look like this:

	1	2	4	5
a	$\boxed{1}$	1	1	-9

In summary, an `if-else` combination always describes two possibilities: what should happen in case the condition is met (`a < 0`), and what should happen in case the condition is not met, `a` having a value that meets the opposite condition `a >= 0`.

Connecting multiple conditions To describe even more than two options, `if` statements can be augmented by adding `else if` statements. In the next fragment, we specify three options: `a < 0`, `a > 0` and otherwise `a = 0`. For each of the three cases, a single dependent instruction is specified, which will only be executed as that specific condition is met. All other cases are then skipped.

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

The program starts with `a` being the integer `3`, and the trace looks like this:

	1	2	4	5
a	$\boxed{3}$	3	3	$\boxed{1}$

By using even more `else if` statements, you can specify further options. The `else` statement always comes last, because it describes "any other possibility".

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
```

Looping In the following fragment a variable is changed using a `while`-loop.

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

On line 1, the variable `a` is initialized with the integer `0`. On the next line, we see the `while` keyword, and then between parentheses a condition `(a < 2)`. Now, *as long as* the condition yields `true`, any statements within the loop will be repeated. In this case, only line 3 is in the loop, but more statements might be included. It's important to realize that the condition is evaluated once per loop: on line 2, each time a decision is made if the loop can be executed again.

Tracing loops Like with sequences of assignment statements, we can trace the execution of loops. The technique is a little bit different, because the loop may be executed multiple times. For the previous fragment, the trace looks like this:

regel	1	2	3	2	3	2	4
var a	$\boxed{0}$	0	$\boxed{1}$	1	$\boxed{2}$	2	$\boxed{1}$
a < 2		true		true		$\boxed{\texttt{false}}$	

Atop the table are the line numbers for each line that we encounter, from left to right. Below that, we include lines for each variable or expression that we would like to trace. Like before, we write down a value for each variable as it is initialized, and we draw a box around each variable whenever it changes.

We included the condition for the `while`-loop as part of the trace. We only write the condition on line 2, because it is only calculated and checked on that line. The condition yields a boolean value, so the possible values are `true` and `false`.

In this case, the condition is checked three times. The first two times it yields `true`, so line 3 is run right after. The third time it yields `false`, after which the rest of the program is run — in this case only line 4.

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
```

More complex conditions In this paragraph we reintroduce more complex conditions that are combinations of propositions with logical connectors. Below you'll find a summary of the rules that apply.

Propositions

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

Logic operations

expression	yields
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

expression	yields
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Negation

expression	yields
`!true`	`false`
`!false`	`true`

The rules of precedence also apply like earlier.

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++
```

Tracing multiple variables with loops Consider the following code fragment:

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

The accompanying trace looks like this;

	1	2	3	4	5	3	4	5	3	6
var a	$\boxed{0}$	0	0	$\boxed{1}$	1	1	$\boxed{2}$	2	2	$\boxed{42}$
var b		$\boxed{0}$	0	0	$\boxed{1}$	1	1	$\boxed{2}$	2	2
a < 2			true			true			$\boxed{\texttt{false}}$	

Incrementing Loops often use a **counter**, a variable that is incremented by 1 in each loop step. Many programming languages have a special operator to do this: `++`. The statement `i++` is equivalent to `i = i + 1`. Decrementing also has a shortcut: `i--`. This command performs the same operator as `i = i - 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 The code fragment below prints the numbers 0 to 9 using a `while`-loop.

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

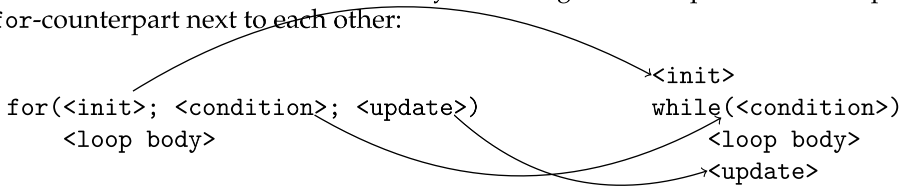
You might have noticed that many loops contain some of the same ingredients. In the program above, line 1 performs **initialization**, line 2 contains the **condition** that controls the loop, and line 4 is an **update** that ensures the loop steps towards the final goal. Because most loops comprise all three components, many programming languages have a special construct to do the same: the *counting for-loop*. The fragment below shows the same program, but instead using a `for`-loop:

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

The same three ingredients, initialization, condition and update, are present in the first line of the loop. By convention, the initialization (`i = 0`) is performed only once, at the loop's start. The condition (`i < 0`) is evaluated each time we restart at the top of the loop. The update (`i++`) is also performed each time, but only *after* all statements in the loop have been performed. So in this example, the *order of execution* is:

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

Because the `for`-loop is a shorthand, it may not be immediately obvious which part is used at which moment. Another way of looking at it, is to put a `while`-loop and its `for`-counterpart next to each other:



```
                                                         <init>
for(<init>; <condition>; <update>)                       while(<condition>)
    <loop body>                                              <loop body>
                                                             <update>
```

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
```

Tracing In the following code fragment we use a `for`-loop to manipulate a variable called a.

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

As soon as we try to trace the loop using previous techniques, we realize that line 2 contains many parts that are used at other points in the code. One way to more easily perform the trace is to first translate the code into a `while`-loop, which is fine. We can also augment our tracing technique to better accommodate `for`-loops. Let's make the initialization (`i = 0`), condition (`i < 2`) and update (`i++`) explicit in the trace:

regel	1	2	2	3	2	2	3	2	2	4
		(i = 0)	(i < 2)		(i++)	(i < 2)		(i++)	(i < 2)	
var i		$\boxed{0}$	0	0	$\boxed{1}$	1	1	$\boxed{2}$	2	
var a	$\boxed{-10}$	-10	-10	$\boxed{0}$	0	0	$\boxed{2}$	2	2	$\boxed{8}$
i < 2			true			true			$\boxed{\text{false}}$	

Once you are used to the form of the `for`-loop, you might reduce the trace again:

regel	1	2	2	3	2	2	3	2	2	4
var i		$\boxed{0}$	0	0	$\boxed{1}$	1	1	$\boxed{2}$	2	
var a	$\boxed{-10}$	-10	-10	$\boxed{0}$	0	0	$\boxed{2}$	2	2	$\boxed{8}$
i < 2			true			true			$\boxed{\text{false}}$	

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)
```

Rewriting This is the side-by-side comparison of `for`- and `while`-loops you saw earlier:

```
                                             <init>
for(<init>; <condition>; <update>)           while(<condition>)
    <loop body>                                  <loop body>
                                                 <update>
```

You can use it to translate `for`-loops to `while`-loops and the other way around. After doing it, you can check your work by tracing both versions and comparing the results.

Chapter 7

Algorithms: sequences

7.1 Sequences

7.1 Write an algorithm that prints the first `10` numbers of the following sequence.

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

7.2 Write an algorithm that prints the first `12` numbers of the following sequence.

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

7.3 Write an algorithm that prints the first `7` numbers of the following sequence.

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

7.4 Write an algorithm that prints the first `9` numbers of the following sequence.

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

7.5 Write an algorithm that prints the first `10` numbers of the following sequence.

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

7.6 Write an algorithm that prints the first `15` numbers of the following sequence.

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

7.7 Write an algorithm that prints the first `9` numbers of the following sequence.

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

Generating sequences You can use a loop to print number sequences. This is a `for`-loop that prints the numbers from 0 up to and including 5:

```
for(i = 0; i < 6; i++)          0
    print(i)                    1
                                2
                                3
                                4
                                5
```

To print other sequences we need to find out the rhythm of the sequence. Quite often, you can base your loop on the example above. Let's say we would like to print six numbers from the sequence $0, 4, 8, 12, ...$, we might notice that each of those numbers is exactly four times the number from the earlier sequence. This means that we could adapt our code as follows:

```
for(i = 0; i < 10; i++)         0
    print(i * 4)                4
                                8
                                12
                                16
                                20
```

If we would like to print the sequence $1, 2, 3, 4, ...$, starting with 1 instead of 0, we could use the following code. Compared to the first example, each number is *one more*:

```
for(i = 0; i < 6; i++)          1
    print(i + 1)                2
                                3
                                4
                                5
                                6
```

Writing an algorithm To design an algorithm is to create a general procedure that can be easily changed to accommodate different situations. In this case, there are two things that can be changed. The *rhythm* of the sequence is defined by the formula in the `print`-statement. The *length* of the sequence can be controlled by changing the loop condition. By using `i < 6` we have been printing sequences of six numbers, but you can change it as required.

Tracing To check the correctness of your programs, you can trace them after writing. Most likely, you will not even have to trace the full extent of the program. Instead, tracing a couple of steps is enough, as long as you can be sure that the rhythm of the generated sequence is as expected.

7.2 More complex sequences

7.8 Write an algorithm that prints the first `12` numbers of the following sequence.

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

7.9 Write an algorithm that prints the first `7` numbers of the following sequence.

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

7.10 Write an algorithm that prints the first `10` numbers of the following sequence.

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

7.11 Write an algorithm that prints the first `9` numbers of the following sequence.

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

7.12 Write an algorithm that prints the first `10` numbers of the following sequence.

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

7.13 Write an algorithm that prints the first `7` numbers of the following sequence.

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

7.14 Write an algorithm that prints the first `9` numbers of the following sequence.

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

Generating more complex sequences This sequence is not easily translated into a combination of the sequences in the previous paragraph:

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

However, the rhythm is not too complicated: each number in the sequence is the previous number, times 2. To generate the sequence, we need more than just a formula: we need to add a variable that *tracks* the state of the sequence while generating it.

```
1 number = 1
2 for(i = 0; i < 10; i++)
3     print(number)
4     number = number * 2
```

The loop now has a few more components. On line 1, the *initial* number is set to 1, the first number of our sequence. Like before, line 2 controls the amount of numbers that are generated by the loop. Line 3 only prints the *current* number, without doing any calculations. Line 4 *updates* the number, calculating the next number from the current one.

Do not forget to trace the algorithms that you write, to ensure that they work well!

7.3 Filtering sequences

7.15 Write an algorithm that prints the first 14 numbers of the following sequence.

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

7.16 Write an algorithm that prints the first 12 numbers of the following sequence.

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

7.17 Write an algorithm that prints the first 12 numbers of the following sequence.

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

7.18 Write an algorithm that prints the first 12 numbers of the following sequence.

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

7.19 Write an algorithm that prints the first 12 numbers of the following sequence.

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

Filtering Let's study this sequence:

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

It looks quite a bit like some of our earlier sequences. However, each number that is divisible by 3 is replaced by a `*`. To generate this sequence, we start from the code for a simpler sequence $0, 2, 4, 6, ...$, and introduce an `if-else` statement:

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

Chapter 8

Strings

8.1 Strings and indexing

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) - 1
print(s[alb * 2])
```

8.14

```
s = "Kolmogorov"
kul = length(s) - 2
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 A string is composed of several *symbols*, also called *characters*. Let's define a string `s` that comprises 13 characters:

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

The characters in such a string have a *position* or *index*. In most programming languages, the index is 0-bases, or in other words, we start counting at zero.

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

Indexing We can retrieve separate characters from a string by *indexing into* it:

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

This fragment prints the character from a string `"Hello, world!"` that is found at position 1. Because we count from 0, this is the letter `e`, one of the characters that computers can represent.

Using variables for indexing To allow us to create algorithms that use strings, it's possible to index into a string using variables or more complex expressions:

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

Boundaries Reading or even writing characters outside the *boundaries* or a string will commonly bring about an *out-of-bounds error* or a *segmentation fault*. For example:

```
s = "Hello, world!"                    error
print(s[20])
```

So keep track of string boundaries!

8.2 Looping with 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])
```

Strings with loops By using loops we can extract information from strings. For example, we can write a loop that prints only some of the characters from a string:

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

On line `3` we use the `length` function to determine the size or length of a string `s` (the result is 6 in this case). Most programming languages have such a built-in command.

As soon as we are going to trace a code fragment like the above, it is useful to annotate the positions in the string, because we will need to *look up* the characters by position quite often.

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

Tracing The key parts to keep track of when tracing this code are the variable `i`, the character at position `i` in the string. We will also want to track the *moments* that the print statement occurs, so we can eventually see what will be printed by the program.

To keep the trace as small as possible, we will not note the values of the variables that do not change: the string `s` itself, and the length of the string, as stored in the variable `len`.

`regel`	`1`	`2`	`3`	`4`	`5`	`6`	`4`	`5`	`6`	`4`	`5`	`6`	`4`
`var i`		$\boxed{0}$	`0`	`0`	`0`	$\boxed{2}$	`2`	`2`	$\boxed{4}$	`4`	`4`	$\boxed{6}$	`6`
`s[i]`		`H`	`H`	`H`	`H`	`o`	`o`	`o`	`i`	`i`	`i`		
`print`					`H`			`o`			`i`		

We can now conclude that the code will print the letters `Hoi` to our screens.

8.3 Looping 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 this code fragment we define an *array* of integers:

```
ooy = [5, 6, 7]          5
print(ooy[0])            6
print(ooy[1])            7
print(ooy[2])
```

The first line defines the array to contain the numbers 5, 6 and 7 and assigns it to the variable ooy. The lines below print each element from the array. Like with strings, *array indexes* start at 0.

Types Used in this way, arrays look a lot like strings. Indeed, in some programming languages, a string is nothing more than an array of characters. The big difference is that arrays allow storage for other things than characters. We will be using arrays to store numbers: both integers and floats.

`ooy = [5, 6, 7]`	→	*array of integers*
`lia = [5.0, 6.0, 7.0]`	→	*array of floats*
`aps = ['e', 'f', 'g']`	→	*array of charakters (often the same as a string)*

Indexing Like with strings, you can use variables for indexing into an array.

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

And again it is possible to use more complex expressions as an index.

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

Mutating Arrays are *mutable*, which means that once we have created an array, we can still make changes later. Usually, that does not mean that the *length* of the array can be changed, but each of its elements can. Take this example:

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

First we create an array containing three 0's, after which we change the second element of the array to `5`. So after executing this code fragment, we have an array containing `[0, 5, 0]`.

9.3 Looping with 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]
```

Loops Arrays, like strings, are often used with loops to extract information, to change whole arrays or to summarize them.

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

After running this code fragment the array `eus` contains values `0`, `2`, `4` and `6`. We use the `length` command to calculate the array's size. Note that not all languages allow you to simply retrieve the length of an array; in those case you will need to keep track of that separately, for example using an integer in a variable.

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()
```

Defining a function For functions there is a separation between the *definition*, which describes what actions the function will perform, and the function *call*, which signals that our function should be run. This also means that as we define a function, it will not automatically be run, allowing us to postpone running it until we need it. This, in turn, allows us to call a function multiple times in different parts of our programs. In this book, a function definition will look like this:

```
<type> <name of function>():
    <function body>
```

For now, we will use `void` for the `<type>`. This word `void` signals us that the function is intended to perform some action (later, we will meet functions that are intended to calculate something). One example of a function that performs an action is a printing function:

```
void print_reassuring_message():
    print("I'm still here!")
```

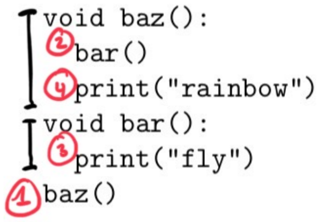
Calling a function Calling a function looks like this:

```
<name of function>()
```

As you can see, the parentheses `()` are an important part of the function definition, as well as of the function call. Most programming languages use these to discern the use of functions from other elements of the program.

Tracing Function calls can be traced, but because definition and calls are separate, we need a clear notation that is different from before. Say we take the program below. The execution of the program starts at the first line that is not a function definition (the last line). From there, we jump to the function `baz` that is being called, which makes us jump to the next function `bar`. The key is to strictly follow the top-down sequence of statements, unless there is a function call.

In our trace we put lines next to the two functions to have a clearly visible separation:



```
void baz():
  ② bar()
  ④ print("rainbow")
void bar():
  ③ print("fly")
① baz()
```

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 Functions often have *parameters*. When calling the function, we supply *concrete* values for these parameters, which is called *parameter passing*. From the function's perspective, these values are assigned names the are specified in the *parameter list* of the function definition.

```
<type> <name of function>(<parameter list>):
    <function body>
```

Now consider these two function definitions. Both have two parameters named in the parameter list.

```
void date_1(day, month):            void date_2(month, day):
    print(day)                          print(day)
    print(month)                        print(month)
```

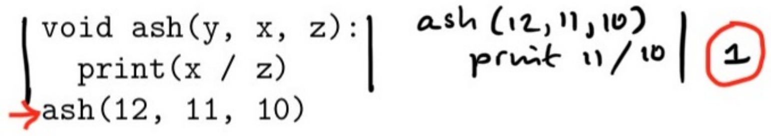
In the left definition we specify the parameters `day` and `month`. In the right definition, we specify `month` and `day`, so in the reverse order. This *order* has an effect on what names are given to the concrete parameters that are passed when calling the function. Let's call the functions:

```
date_1(21, 6)                       date_2(6, 21)
```

The output of the functions would then be:

```
21                                  21
6                                   6
```

Tracing Keeping track of all values when passing parameters can easily become very tedious, which is why we often need to trace them explicitly. Below, like before, we put a line next to the one function that is defined. We also mark the starting line with a little arrow.



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

On that starting line, the function `ash` is called. To the right of the definition of that function, we draw a line, and copy the function definition, while substituting the concrete values from the function call.

Combining the original function definition and the substituted version, we can infer that in the function `y = 12`, `x = 11` and `z = 10`. We use this information to substitute those values in the right places of the `print` line.

Then, only a small calculation is left, of which the result will be printed. When we evaluate the expression using the basic rules, we get the number that will be printed.

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)
```

Passing variables Functions can also accept the values from variables as concrete values for their parameters. Let's study the following function:

```
void twice(text):
    print(text)
    print(text)
fruits = "pineapple blueberry"
twice(fruits)
```

```
pineapple blueberry
pineapple blueberry
```

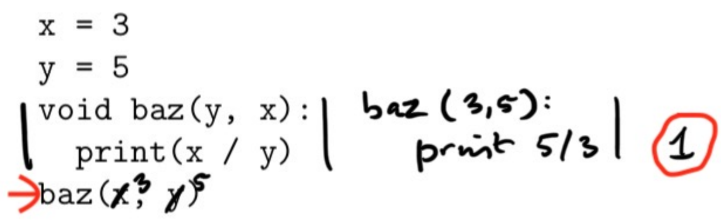
We passed a single variable called `fruits` to the function. Its contents are named `text` inside the function definition. It is this text that is printed on the screen twice.

Like before, the order of the parameters will be consistent between the definition and any function calls that are performed. This is especially important because sometimes, parameters may have the same name as variables found elsewhere in the code. Consider this fragment:

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

Because the values of `y` and `x` are passed to the function in that order, inside the function we will have the concrete parameters `x = 40` and `y = 10`. Hence, the result that is printed will be 30.

Tracing Because of the potential confusion between variable and parameter names, we add an explicit step to our function tracing technique: substituting values in the function call. We cross out the variable name and write its value next to it. We can do this before considering the function's definition at all. Now that we have substituted the concrete values, it's easy to copy them into the concrete function call like in earlier sections.



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

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))
```

Returning To *return* means to transfer back some value from the function to where it was called. Most programming languages use the keyword *return* for this. The value that is returned is sometimes called the *result* of the function. In the code where the function is called, the function call might be replaced by its result, in the same way that we use concrete values each time we calculate using variables.

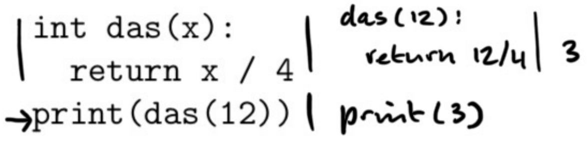
Types Right at the start of this chapter we showed the following prototype of a function definition:

```
<type> <name of function>(<parameter list>):
  <function body>
```

Up until now we have only used `void` for `<type>`. We can now define that `void` means that a function *returns nothing*, meaning that the function will not contain a `return` statement.

Now if we would like to have the function return something, we will specify as `<type>` what kind of value the function will be returning. In this book, we will use the type `int`, `float` and `string`.

Tracing Functions that calculate and return something can be traced much like before. We add a *back substitution* that fills in the concrete result of the function into the place where it was called. Below, the function call `das(12)` returns the integer 3, and this is substituted into the `print` statement.



```
|int das(x):        | das(12):
|  return x / 4     |   return 12/4 | 3
->print(das(12))    | print(3)
```

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

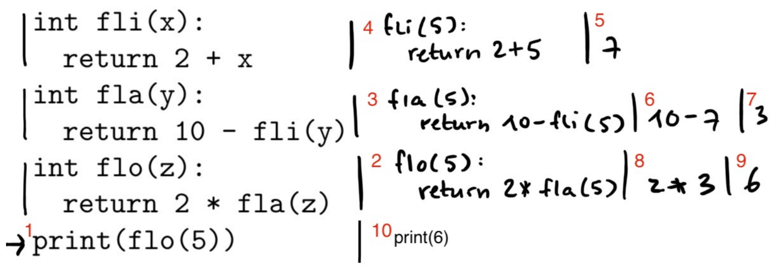
```
int loo(y, z)
   return z / 3
int aam(z, y)
   return loo(y, z) * 6
int wah(y, z, x)
   return aam(y, z) - 2
z = 2
x = 5
y = 4
print(wah(x, y, z))
```

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))
```

Stacked function calls Functions that calculate and return something can also call other functions that return something. Tracing might look like below. Using red digits, we indicate the order in which the trace is done.



```
int fli(x):                  | 4 fli(5):                    | 5
  return 2 + x               |     return 2+5              |   7
int fla(y):                  | 3 fla(5):                   | 6          | 7
  return 10 - fli(y)         |     return 10-fli(5)        |   10-7     |   3
int flo(z):                  | 2 flo(5):                   | 8          | 9
  return 2 * fla(z)          |     return 2* fla(5)        |   2*3      |   6
→1 print(flo(5))             | 10 print(6)
```

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	1.27	1.28	1.29	1.30
a.	`3`	`8.0`	`-4.5`	`5.5`	`3.0`
b.	`-3.0`	`1.5`	`-13`	`-2.5`	`10.0`
c.	`43.5`	`4.0`	`2.0`	`6.0`	`6.0`
d.	`-5.5`	`6`	`3`	`-3.75`	`3.0`
e.	`0.25`	`5.5`	`0.5`	`3.0`	`16.5`

	2.1	2.2	2.3	2.4	2.5
a.	`FALSE`	`TRUE`	`FALSE`	`FALSE`	`FALSE`
b.	`FALSE`	`FALSE`	`TRUE`	`TRUE`	`FALSE`
c.	`FALSE`	`FALSE`	`FALSE`	`FALSE`	`FALSE`
d.	`FALSE`	`TRUE`	`FALSE`	`TRUE`	`FALSE`
e.	`TRUE`	`FALSE`	`FALSE`	`FALSE`	`FALSE`

	2.6	2.7	2.8	2.9	2.10
a.	`TRUE`	`TRUE`	`TRUE`	`FALSE`	`TRUE`
b.	`FALSE`	`FALSE`	`FALSE`	`TRUE`	`FALSE`
c.	`FALSE`	`TRUE`	`TRUE`	`FALSE`	`FALSE`
d.	`FALSE`	`TRUE`	`FALSE`	`FALSE`	`FALSE`
e.	`FALSE`	`FALSE`	`FALSE`	`FALSE`	`FALSE`

	2.11	2.12	2.13	2.14	2.15
a.	`TRUE`	`TRUE`	`FALSE`	`FALSE`	`FALSE`
b.	`TRUE`	`TRUE`	`FALSE`	`TRUE`	`FALSE`
c.	`FALSE`	`FALSE`	`TRUE`	`TRUE`	`TRUE`
d.	`FALSE`	`TRUE`	`TRUE`	`TRUE`	`TRUE`
e.	`FALSE`	`TRUE`	`FALSE`	`FALSE`	`TRUE`

	2.16	2.17	2.18	2.19	2.20
a.	`TRUE`	`TRUE`	`FALSE`	`TRUE`	`TRUE`
b.	`FALSE`	`TRUE`	`FALSE`	`TRUE`	`TRUE`
c.	`TRUE`	`TRUE`	`FALSE`	`FALSE`	`TRUE`
d.	`TRUE`	`FALSE`	`FALSE`	`TRUE`	`FALSE`
e.	`TRUE`	`FALSE`	`TRUE`	`FALSE`	`FALSE`

3.1	3.2	3.3	3.4	3.5
`dey = 3.5`	`gog = 3`	`eli = 1.0`	`jon = -1.0`	`abe = 1.5`
`luo = 1.5`	`oer = 1.5`	`soc = 4.0`	`aus = 1.0`	`sir = 3.0`

3.6	3.7	3.8	3.9	3.10

`vow = 3.0`
`nub = 4.0`
	`gup = 0`
	`ley = 0`
		`tez = 2.5`
		`nap = 7.5`
			`jib = 5`	`aus = 1.5`

3.11	3.12	3.13	3.14	3.15

`bam = 2.0`	`tye = 1`	`nam = 4`
		`sob = 0.0`
		`off = 4`
			`kaw = 1.0`
			`zan = 1.5`
				`quo = -0.75`
				`ora = -2.75`

3.16	3.17	3.18	3.19	3.20

`gup = 2.5`
`han = 3.5`
	`lod = 5`
	`nid = 6.0`
	`mou = 18`
	`gyp = -2.5`
		`you = 3`
		`way = 0.5`
		`iwa = 1.5`
			`fei = 2`
			`oka = 6.0`
				`fae = 0.5`

4.1	4.2	4.3	4.4	4.5
`a = 11`	`a = 0`	`a = 0`	`a = 0`	`a = 10`
4.6	4.7	4.8	4.9	4.10
`a = 10`	`a = 18`	`a = 10`	`a = -3`	`a = 5`
4.11	4.12	4.13	4.14	4.15

`a = 1`	`a = 0`	`a = 20`	`a = 20`	`a = 9`
				`b = 9`

4.16	4.17	4.18	4.19	4.20

`a = 0`
`b = 2`
	`a = -1`
	`b = -2`
		`a = 0`
		`b = -2`
			`a = 6`
			`b = 8`
				`a = 4`
				`b = -1`

4.21	4.22	4.23	4.24	4.25	4.26	4.27	4.28
`a = 10`	`a = -10`	`a = 7`	`a = 9`	`a = 8`	`a = 7`	`a = 8`	`a = 9`

5.1	5.2	5.3	5.4	5.5
`a = 6`	`a = 15`	`a = 16`	`a = 8`	`a = 1`
5.6	5.7	5.8	5.9	5.10
`a = 256`	`a = 18`	`a = 31`	`a = -31`	`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	8.20	8.21	8.22
`Hello, world!`	`Hlo ol!`	`!dlrow ,olleH`	`Hello, world!`

8.23	8.24	8.25	8.26	8.27	8.28	8.29
`terug`	`yes!`	`go back`	`index`	`aarrgg`	`hi!hi!`	`hi!hi!`

8.30	8.31	8.32	8.33
`Hello, world`	`bounds err`	`scrabble`	`a2b3`
8.34	8.35	8.36	
`a1a2b3`	`aaarrr`	`hi`	

9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9	9.10
`5`	`2.1`	`x`	`2`	`2`	`0.1`	`-1`	`1`	`0.6`	`0`
9.11	9.12	9.13	9.14	9.15		9.16	9.17	9.18	
`8.1`	`6.2`	`14`	`4`	`bounds err`		`2`	`3`	`3`	

9.19	9.20	9.21
`lop = [10, 5, 30]`	`kol = [‘h’, ‘o’, ‘i’]`	`fof = [0.1, 0.2, 1.0]`
9.22	9.23	9.24
`dido = [2, 15, 10]`	`l = [5, 1, 2]`	`l = [0.1, 1.0, 0.3]`
9.25	9.26	9.27
`yoyo = [1.5, 1.5, 1.0]`	`yoyo = [2.0, 2.0, 1.0]`	`yoyo = [1.5, 1.0, 1.0]`
9.28	9.29	9.30
`hipp = [42, 4, 101]`	`fle = [7, 6, 10]`	`k = [0.3, 2.0, 0.6]`
9.31	9.32	9.33
`r = [0, 1, 0, 0]`	`zi = [‘s’, ‘b’, ‘c’]`	`b = [false, true, true]`
9.34	9.35	9.36
`rol = [1, 2, 3, 0]`	`j = [0, -2, -2]`	`r = [4, 2, 1]`

9.37	9.38	9.39
`1 2 3 4 5`	`5 4 3 2 1`	`1 2 3`
9.40	9.41	
`2 1`	`lap = [‘x’, ‘x’, ‘x’, ‘x’, ‘x’]`	

9.42	9.43	9.44
`gop = [2, 5, 1, 8, 8]`	`gop = [5, 3, 6, 2, 8]`	`gop = [4, 4, 4, 4, 4]`

10.1	10.2	10.3	10.4
`fly`		`fly jump jump`	`jump`
10.5	10.6	10.7	10.8
`pie pie bicycle`	`fly rainbow`	`jump`	`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`