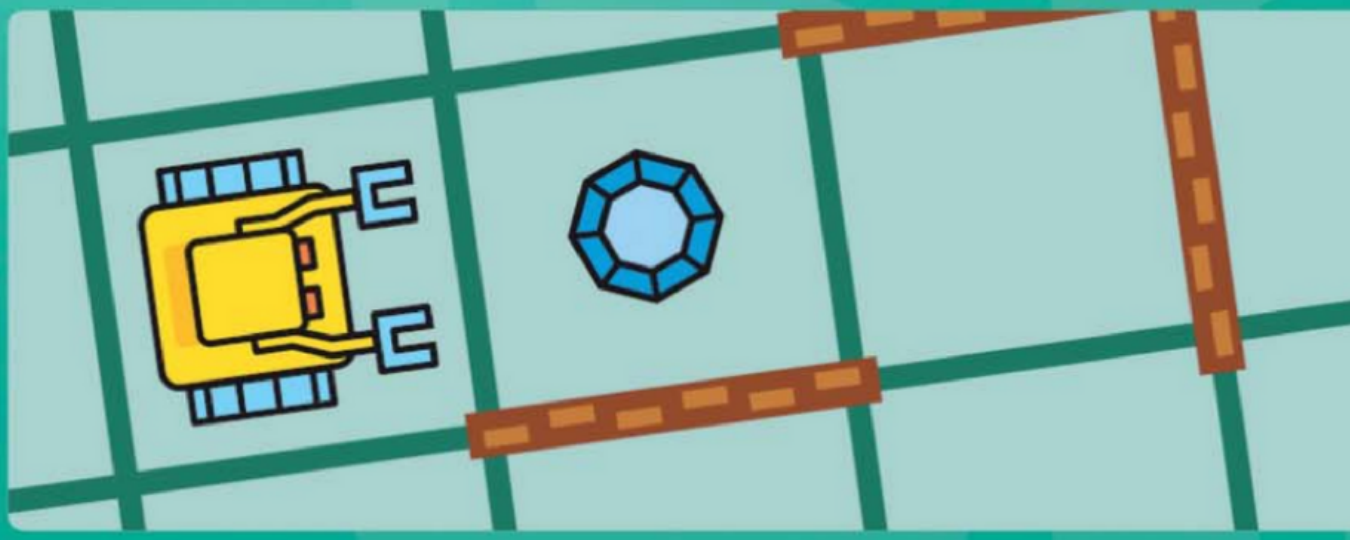
nclab

Learning for Industry 4.0

Learn how to **Think** with Karel the Robot



© Copyright 2018 NCLab

Learn How to Think
with Karel the Robot

Dr. Pavel Solin

September 24, 2018

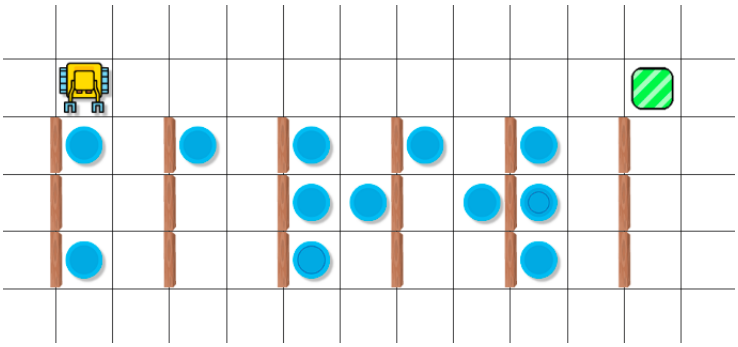
© 2011-2018 NCLab, Inc. All Rights Reserved.

Preface

Computer programming is fun. Telling a machine what to do, and then watching it actually do it, is amazing. Programming is all about breaking complex problems into simpler ones which are easier to solve. Interacting with the computer will teach you how to be accurate, use logic, solve problems, persevere, overcome failure, and get things done. These are tremendously important life skills which will help you succeed in anything you will do in the future.

Now, you might ask: *"Why should I lose time with Karel the Robot - an educational programming language - when I can start right away learning a real programming language such as Python, C++ or Java?"* The answer is that computer programming is like driving a car: One thing is to learn how to operate the vehicle - start it, shift gears, push the gas pedal and brakes, etc. But another, even more important thing is to learn the traffic rules really well: What should you do when you are coming to an intersection? Who has the right of way? What are the meanings of the traffic lights and symbols? In short - how to become a good driver. In the context of computer programming, Karel will teach you how to be such a good driver. And after that, it will be incredibly easy for you to learn how to operate various other "cars" which are other languages such as Python, Java or C++.

By the way, Karel is not a toy language at all. It can solve very hard problems including classical world-class programming challenges such as the Eight Queens puzzle which you can find on Wikipedia. Karel also can parse binary trees using recursion, implement sorting algorithms, perform statistical experiments, or read Braille text:



Karel reading his own name in Braille.

About Karel the Robot

The educational programming language Karel the Robot was created at the Stanford University by Dr. R.E. Pattis who also wrote the original textbook *Karel the Robot: A Gentle Introduction to the Art of Programming* in the 1980s. At that time, its syntax was influenced by Pascal, a major programming language of that era. We have updated the language to be compatible with Python, while preserving Dr. R.E. Pattis' original ideas. Python is a major programming language of modern engineering and science.

About the Author

Dr. Pavel Solin is Professor of Computational Science at the University of Nevada, Reno. He has used the first 8-bit computers to draw ornaments on the screen of small black and white TV when he was 9. There were no computer monitors yet, and games were stored on audio casettes. Much changed since then but Dr. Solin's passion for computers and programming remained the same. Today he is using the most powerful supercomputers to understand what happens inside of collapsing stars and other natural processes that cannot be observed or measured. He is fluent in several computer languages, wrote hundreds of thousands of lines of code, leads open source software projects, and enjoys learning new things every day.

Acknowledgment

We would like to thank educators and students for reporting bugs, suggesting new features, and providing valuable feedback. This is helping us to continuously improve the self-paced interactive Karel course in NCLab, as well as this textbook and the NCLab computing platform itself.

Contents

Preface ..... **i**
About Karel the Robot ii
About the Author ii
Acknowledgment ii

1. Introduction **1**
1.1. Karel Čapek and his 1921 play R.U.R. 1
1.2. Brief history of the Karel language 2
1.3. Various past implementations 2
1.4. New Karel language based on Python 3
1.5. Command `right` and some other new features 4
1.6. What can Karel do? 4
1.7. How does Karel differ from Python? 5
1.8. Review questions 5

2. Basic Commands **8**
2.1. Manual mode 8
2.2. Steps and operations 10
2.3. Using keyboard controls 10
2.4. Seeing the world through the robot's eyes 10
2.5. Programming mode 11
2.6. Algorithm vs. program 12
2.7. Usually, a task has more than one solution 13
2.8. How to recognize the best solution 14
2.9. Logical errors 15
2.10. Syntax rules and syntax errors 16
2.11. Debugging and where did the word "bug" come from 18
2.12. Karel's bag 19
2.13. FILO (stack) and FIFO (queue) 20
2.14. Karel is case-sensitive 21

2.15. Review questions 21

3. Counting Loop 25
3.1. Counting loop and the keyword `repeat` 25
3.2. Commenting your code 27
3.3. Repeating a sequence of commands 28
3.4. Indentation matters 29
3.5. How to figure out repeating patterns quickly and correctly 30
3.6. Stashing coins 32
3.7. Cooking potatoes 35
3.8. When it is not be possible to start a loop right away 36
3.9. Additional commands might be needed after a loop finishes 38
3.10. Nested loops 39
3.11. Revisiting previous programs 42
3.12. Crab cake 43
3.13. Starfish square 44
3.14. Triple-nested loops 47
3.15. Pearl necklace 48
3.16. Even more levels of nesting 49
3.17. Review questions 50

4. Conditions 53
4.1. What are conditions? 53
4.2. Why does Karel need them? 53
4.3. Karel's `wall` sensor and the `if` statement 54
4.4. What happens if there is no wall? 55
4.5. Other types of obstacles 56
4.6. Combining loops and conditions 57
4.7. Collectible objects 58
4.8. Collecting bananas 59
4.9. Containers 60
4.10. Collecting coconuts 61
4.11. The `else` branch 61
4.12. Hurdle race 63
4.13. Function `print` 63
4.14. Sensor `north` 64
4.15. Command `pass` 65

4.16. Finding North 65
4.17. Keyword `not` 66
4.18. Sensor `empty` 67
4.19. Keyword `and` 69
4.20. Bounty 69
4.21. Keyword `or` 70
4.22. Danger 72
4.23. Sensor `home` 73
4.24. The full `if-elif-else` statement 73
4.25. More is coming! 75
4.26. Review questions 75

5. Conditional Loop 79
5.1. Conditional loop and the keyword `while` 79
5.2. The difference between the conditional and counting loops 79
5.3. Step by step 80
5.4. Climbing a pyramid 81
5.5. Narrow escape 84
5.6. Combining the `repeat` and `while` loops 85
5.7. First Maze Algorithm (FMA) 87
5.8. Three possible failures of the FMA 89
5.9. Right-handed version of the FMA 91
5.10. Other types of mazes the FMA can handle 92
5.11. Improving Program 4.9 "Finding North" 92
5.12. Improving Program 4.12 "Emptying bag" 93
5.13. Improving Program 4.17 "Danger" 93
5.14. Improving Program 4.18 "Tunnel" 94
5.15. Other types of loops: `do-while`, `until`, and `for` 94
5.16. More is coming! 94
5.17. Review questions 95

6. Custom Commands 97
6.1. Why are custom commands useful? 97
6.2. Defining a custom command `star` 101
6.3. Assembling the solution of the main task 102
6.4. Collecting water bottles 106
6.5. The optional `return` statement 109

6.6. Arcade game 110
6.7. Second Maze Algorithm (SMA) 113
6.8. Right-handed version of the SMA 117
6.9. Rock climbing 118
6.10. Program length vs. efficiency 119
6.11. Writing monolithic code vs. using custom commands 121
6.12. Defining commands inside other commands 123
6.13. Create your own Karel language! 124
6.14. Review questions 126

7. Variables 128
7.1. What are variables and why are they useful? 128
7.2. Types of variables 129
7.3. Choosing the names of variables 129
7.4. Creating and initializing numerical variables 130
7.5. Assignment operator = ... 130
7.6. Displaying the values of variables 130
7.7. Basic math operations with variables 131
7.8. Keywords `inc` and `dec` 132
7.9. Arithmetic operators +=, −=, ⋆= and /= 132
7.10. Comparison operators ==, !=, <, <=, > and >= 133
7.11. Counting maps 135
7.12. Water supply 136
7.13. Karel and the Fibonacci sequence 137
7.14. Review questions 138

8. Functions 141
8.1. Defining and using functions 141
8.2. Builder 142
8.3. Storekeeper 144
8.4. Local variables and local scope 146
8.5. Global variables and global scope 147
8.6. Functions that accept arguments 149
8.7. Can a function change the values of its arguments? 150
8.8. Review questions 152

9. Text Strings 153

9.1. Raw text strings and text string variables 153
9.2. Assigning, displaying, and comparing text strings 154
9.3. Concatenating text strings 155
9.4. Multiplying text strings with integers 155
9.5. Length of a text string 156
9.6. Parsing a text string with the `for` loop 156
9.7. Reversing text strings 156
9.8. Extracting individual characters by their indices 157
9.9. Using negative indices 158
9.10. Slicing text strings 159
9.11. Displaying quotes in text strings 160
9.12. Checking for substrings 161
9.13. Counting occurrences of substrings 162
9.14. Finding the positions of substrings 163
9.15. Searching and replacing in text strings 164
9.16. Removing substrings 165
9.17. Swapping substrings 166
9.18. Executing text strings as code 167
9.19. Your homework 169
9.20. Review questions 169

10. Testing Your Programs 172
10.1. Morse code project - Part I 172
10.2. Testing the function `row` 174
10.3. Morse code project - Part II 176
10.4. Testing the function `english(symbol)` 178
10.5. Morse code project - Part III 180
10.6. Your homework 181
10.7. Review questions 181

11. Boolean Values, Variables, Expressions, and Functions 182
11.1. George Boole 182
11.2. Quick introduction to logic 183
11.3. What is *Boolean algebra?* 184
11.4. Boolean values and variables in Karel 185
11.5. GPS sensors `gpsx` and `gpsy` 187
11.6. Comparison operators ==, !=, <, >, <= and >= 190

11.7. Boolean functions 190
11.8. Karel's sensors are Boolean functions 192
11.9. The `if` statement revisited 193
11.10. Using the `if` statement to display debugging information 194
11.11. Another look at the `while` loop 195
11.12. Infinite loop `while True` 196
11.13. Review questions 198

12. Randomness and Probability 200
12.1. Why is randomness useful in computing 200
12.2. Generating random integers 201
12.3. Karel is rolling a die 202
12.4. Placing objects at random locations 203
12.5. Building a random skyline 205
12.6. Calculating the maximum 206
12.7. Measuring the height of a skyline 207
12.8. Measuring the height of a building 209
12.9. Calculating the minimum 211
12.10. Measuring the clearance of a cave 212
12.11. Generating random Booleans 213
12.12. Karel is tossing a coin 214
12.13. Calculating probabilities 214
12.14. Probability of events vs. their frequency 216
12.15. Karel and random walks 218
12.16. Using randomness to solve difficult tasks 219
12.17. Review questions 220

13. Lists 223
13.1. What are lists and why they are useful 223
13.2. Creating empty and nonempty lists 224
13.3. Appending items to a list 225
13.4. Measuring the length of a list 226
13.5. Accessing list items via their indices 226
13.6. Creating a list of lists 227
13.7. Parsing lists with the `for` loop 228
13.8. Checking if an item is in a list 229
13.9. Removing and returning ("popping") items from a list 231

13.10. Adding lists 232
13.11. Multiplying lists with integers 232
13.12. Deleting items from a list 233
13.13. Gardener 234
13.14. Expedition Antarctica 235
13.15. Copycat 238
13.16. Review questions 241

14. Recursion 243
14.1. What is recursion and why is it useful 243
14.2. Which tasks are suitable for recursion? 245
14.3. Collecting shields 247
14.4. Under the hood 248
14.5. Forgetting the stopping condition 249
14.6. Moving shields to boxes 251
14.7. Museum heist 251
14.8. Mutually recursive commands 252
14.9. Using variables and functions in recursion 253
14.10. Parsing lists using recursion 255
14.11. Recursion at its best 256
14.12. Review questions 257

15. Advanced Applications 258
15.1. Flood (recursion) 258
15.2. Contraband (recursion) 260
15.3. Traversing binary trees (recursion) 262
15.4. Bubble sort 263
15.5. Reading Braille 266
15.6. Cardan grille 270
15.7. Land surveyor 274
15.8. The Eight Queens puzzle 276

A. Karel App in NCLab 281
A.1. Launching the Karel app 281
A.2. Building mazes 282
A.3. Create your own game! 284
A.4. Check how difficult your game is 287

A.5. Convert worksheet into a game 288
A.6. Define game goals 290
A.7. Final test 291
A.8. Publish your game in the Internet! 292

B. Self-Paced Karel Course in NCLab 294
B.1. Age groups and prerequisites 294
B.2. Brief overview 294
B.3. What does it take to be an instructor 295
B.4. Role of the instructor 295
B.5. Instructor training 295
B.6. Course structure 295
B.7. Syllabus, lesson plans, pacing guide 295
B.8. Student journals, cheat sheets, solution manuals 296
B.9. Creative projects 296
B.10. Explore NCLab 297
B.11. Contact us 297

Index 298

1. Introduction

In this chapter you will learn:

- About a Czech writer Karel Čapek who predicted human-like machines and invented the word "robot".
- About a Stanford University professor R.E. Pattis who created an educational programming language to help his students learn logic.
- Basic facts about the Karel language, and how it differs from other programming languages.

1.1. Karel Čapek and his 1921 play R.U.R.

Karel Čapek (1890 - 1938) was a legendary Czech science fiction writer who first predicted human-like machines and invented the word "robot" in his 1921 science fiction play R.U.R. (Rossum's Universal Robots). The play was soon translated to more than 20 languages. In his visionary novels, Karel Čapek also predicted nuclear weapons, the Second World War, and consumer society. He was nominated for the Nobel Prize in Literature seven times.





Karel Čapek and his 1921 play R.U.R.

1.2. Brief history of the Karel language

The educational programming language Karel the Robot was introduced by Dr. Richard E. Pattis at the Stanford University in his book *Karel The Robot: A Gentle Introduction to the Art of Programming* in 1981.

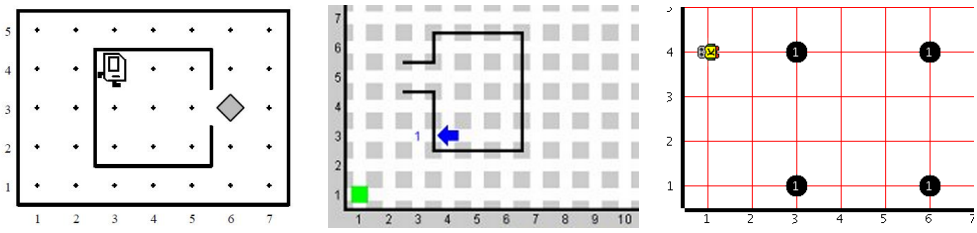


Dr. Richard E. Pattis in the 1980s and today.

R.E. Pattis created the language to help his students learn programming logic. Nowadays, Karel is used at countless schools in the world. In 2006, R.E. Pattis was recognized by the Association for Computing Machinery (ACM) as a Pioneering Innovator Who Significantly Advanced the Digital Age.

1.3. Various past implementations

There have been a number of different implementations of the Karel language since the 1980s. Some of them can still be downloaded from the web and used today while some others were designed for older operating systems (such as Windows 3.1) which no longer are in use:



Various past Karel implementations.

1.4. New Karel language based on Python

The original Karel language was based on Pascal, a popular programming language of the 1980s. For illustration, here is a sample program written using the original Karel language:

PROGRAM 1.1. Sample program written using the original Karel language

```
BEGINNING-OF-PROGRAM

DEFINE turnright AS
BEGIN
    turnleft
    turnleft
    turnleft
END

BEGINNING-OF-EXECUTION
    ITERATE 3 TIMES
    BEGIN
        turnright
        move
    END
    turnoff
END-OF-EXECUTION

END-OF-PROGRAM
```

Since Pascal is not a mainstream programming language today, we revised the Karel language and adjusted its syntax to be similar to Python, a major programming language of modern engineering and science. This change made Karel much easier to use. For illustration, here is the above program again, written in the new Karel language:

PROGRAM 1.2. Program 1.1 rewritten using the new Karel language

```
def turnright
  repeat 3
    left

repeat 3
  turnright
  go
```

1.5. Command `right` and some other new features

The new Karel language has a built-in command `right` for the right turn and the corresponding animation. In the original Karel language, the robot had to make three left turns to turn right. As a result, when solving more complex tasks, the robot was spinning a lot. In fact, he resembled a tornado passing through the maze. With this command, Program 1.2 can be reduced to just three lines:

PROGRAM 1.3. Simplified version of Program 1.2

```
1 repeat 3
2   right
3   go
```

For convenience, long keywords were replaced with shorter ones, such as `leftturn` with `left`, `move` with `go`, `pickbeeper` with `get`, `putbeeper` with `put`. Complicated syntax features such as semicolons and parentheses were removed for easier use.

And finally, the original Karel only had beepers and walls in the maze. We added many other collectible objects such as gems, nuggets, spiders and snakes. We also added many other types of obstacles (which Karel must not run into or he explodes) such as water, fire, skulls, scorpions and acid. The new Karel world also has containers (where he can drop off collectible objects) such as marks, boxes, baskets, chests and fishing nets. More about creating mazes and making games will be explained in Appendix A.

1.6. What can Karel do?

Karel is a determined robot that can cross a jungle or a desert, explore caves, collect pearls on the ocean's floor, climb icy mountains and do many other cool things:



Karel the Robot.

He can detect collectible objects beneath him and obstacles in front of him. He can detect if he is facing North and if he is at his home square. He can also collect objects and/or remember their GPS coordinates, store them in variables and lists, and place objects on the floor or in containers as needed.

1.7. How does Karel differ from Python?

Despite its playful appearance, Karel features all key concepts of modern procedural programming. Technically speaking, it is a complete Turing machine. As you will see later, the Karel programs presented in this textbook range from very simple to extremely challenging.

As we already mentioned, the new version of the Karel language is influenced by Python. Basic Python concepts such as numbers, text strings, logical values, and lists are implemented in Karel. Also, Karel knows basic Python statements. However, not every code written in Karel will work in Python and vice versa. Karel does not use the colon symbol `:` after compound statements `if`, `while`, `def`, etc. Karel has the loop `repeat` which is a simplified version of the Python `for` loop. Also, Karel has a set of its own commands and functions such as `go`, `left`, `right` which are not available in Python.

1.8. Review questions

Note: For all review questions in this book: None, one, or multiple answers may be correct.

QUESTION 1.1. *Karel Čapek was a famous Czech*

A musician
B football player
C hockey player
D writer

QUESTION 1.2. *The word "robot" was first introduced in*

A 1921
B 1291
C 1991
D 1221

QUESTION 1.3. *The word "robot" was first introduced in*

A the song Robotic Mind.

B the book Uprising of Robots.
C the theater play R.U.R.
D the movie E.T.

QUESTION 1.4. *Who created the educational programming language Karel the Robot?*
A Karel Čapek
B Pavel Solin
C Richard E. Pattis
D John von Neumann

QUESTION 1.5. *Name the university where Karel the Robot was first used!*
A MIT
B Princeton
C Harvard
D Stanford

QUESTION 1.6. *When was Karel the Robot created?*
A 1981
B 1991
C 2001
D 1967

QUESTION 1.7. *Why was Karel the Robot created?*
A To help students learn logic.
B To help students learn keyboarding.
C To help students learn geometry.
D To help students learn robotics.

QUESTION 1.8. *What major programming language influenced the original Karel language?*
A Cobol
B Basic
C Fortran
D Pascal

QUESTION 1.9. *How many times was Karel implemented in the past?*
A One time.
B Two times.
C At least three times.
D Karelw as never implemented in the past.

QUESTION 1.10. *The new Karel language used in this textbook is based on*

A C++
B *Lua*
C *Python*
D *Ruby*

QUESTION 1.11. *What command was newly introduced in the new Karel language?*

A *left*
B *right*
C *jump*
D *turn*

QUESTION 1.12. *Which of the following statements are true?*

A *The colon* `:` *is not used in the Karel language.*
B *Python contains all commands which are in Karel.*
C *Karel contains all commands which are in Python.*
D *Karel can use GPS coordinates.*

QUESTION 1.13. *What types of objects can be found in Karel's maze in NCLab?*

A *Walls only.*
B *Beepers only.*
C *Collectible objects, obstacles, and containers.*
D *Walls and beepers.*

QUESTION 1.14. *What can Karel do?*

A *Collect objects which are beneath him.*
B *Detect obstacles which are in front of him.*
C *Detect if he is facing North.*
D *Place objects on the floor and/or in containers.*

QUESTION 1.15. *What is the main benefit of learning programming with Karel the Robot?*

A *Karel is simplified C++.*
B *Karel is a modern version of Cobol.*
C *Karel is only for those who don't want to learn any other programming language.*
D *With Karel, one can learn programming methodology easily without struggling with the technical complexity of "real" programming languages.*

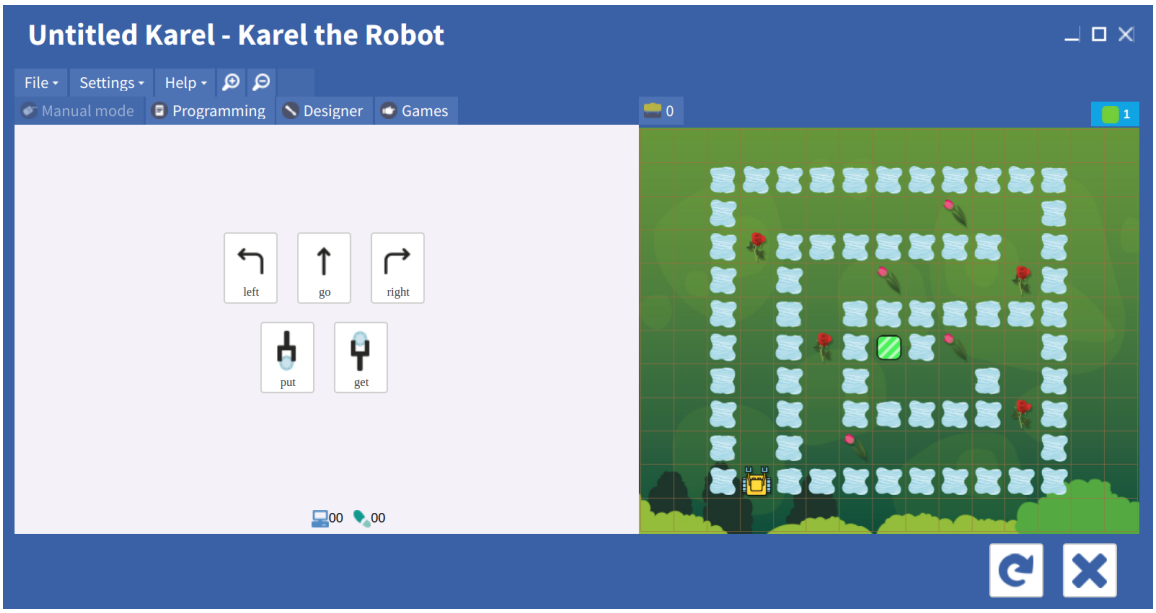
2. Basic Commands

In this chapter you will learn:

- How to control Karel in Manual mode using buttons and the keyboard.
- To see the world through the robot's eyes.
- The difference between steps and operations.
- The difference between an algorithm and a program.
- How to write first Karel programs using basic commands.
- That most tasks have multiple solutions, and how to recognize the best one.
- The difference between logical and syntax errors, and what is debugging.
- About Karel's bag, and the meaning of FILO and FIFO.

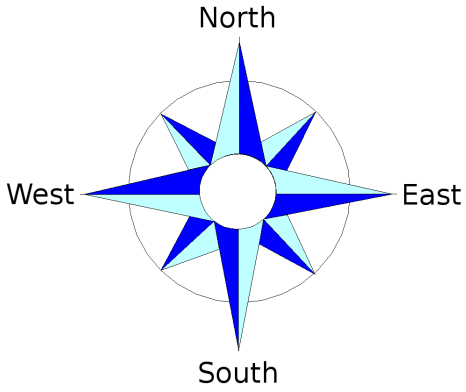
2.1. Manual mode

Let's switch the Karel app in NCLab to Manual mode. Here, Karel can be guided by pressing buttons or certain keys on the keyboard. The keywords displayed on the buttons represent the five basic commands of the Karel language - `go`, `left`, `right`, `get` and `put`:



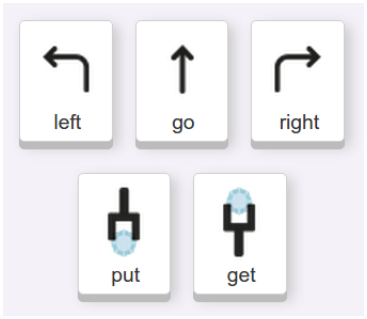
Karel application in Manual mode.

Before we go further, recall the four major directions on the compass - North, South, East and West:

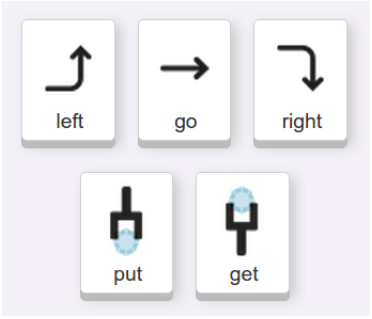


Four major directions on the compass.

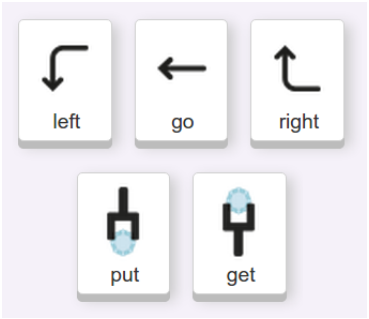
The buttons change dynamically according to the direction which the robot is facing:



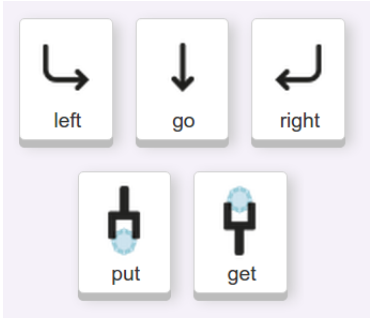
Karel faces North.



Karel faces East.



Karel faces West.



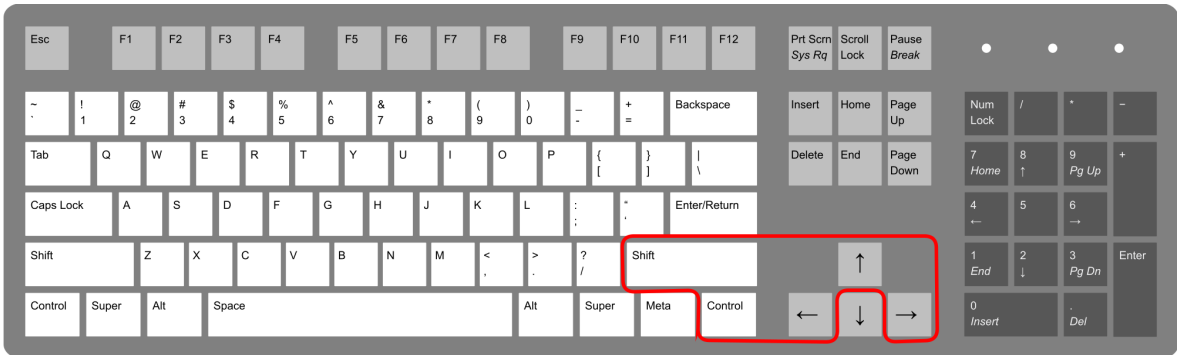
Karel faces South.

2.2. Steps and operations

On the bottom of the left panel you can see two icons that represent a computer (💻) and a trace (💊). These are the counters of operations and steps, respectively. An operation is anything the robot does - making one step forward, turning left or right, collecting an object, or placing an object on the ground. The number of operations is always greater than or equal to the number of steps. For example, when Karel makes a full 360-degree turn by executing four times the command `right`, he will do four operations but make zero steps.

2.3. Using keyboard controls

In Manual mode, the robot can also be guided using the keyboard. The corresponding keys are highlighted in the image below:



Karel's keyboard controls.

The up arrow corresponds to `go`, left arrow to `left`, right arrow to `right`. The Shift key corresponds to `get` and the Control (CTRL/CMD) key to `put`.

2.4. Seeing the world through the robot's eyes

In order to guide the robot correctly, one needs to see the world through the robot's eyes. For example, when the robot faces North, then after turning right he will face East:



When Karel faces North, his right is your right.

But when Karel faces South, then after turning right he will face West:



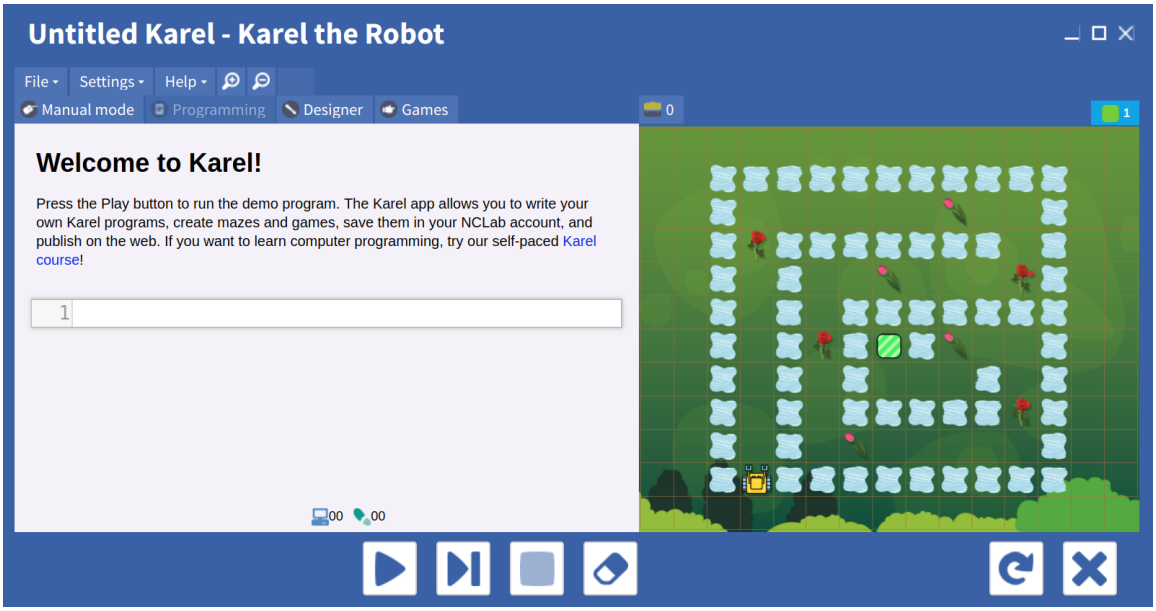
When Karel faces South, his right is your left.

This is the same skill that you need to correctly read maps.

Working with Karel will improve your ability to read maps.

2.5. Programming mode

Now, let's switch the Karel app to the Programming mode:



Karel application in Programming mode.

Here, the robot can be guided by typing commands. The code cell is located in the left panel. The control buttons are explained in the following table:

Control buttons in the Programming mode



Run the program

Stop the program

Restore the maze



Step through the program

Erase the code

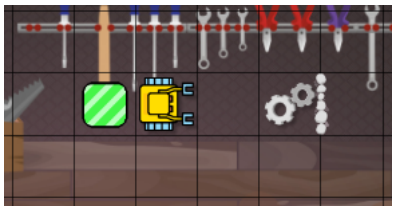
Close the Karel app

2.6. Algorithm vs. program

Karel will always follow your commands *exactly* to the letter - no exceptions. If the robot does something wrong, such as crashing into a wall, then most likely it was not his mistake but yours. Your *algorithm* was wrong.

An *algorithm* is a sequence of logical steps
that leads to the solution of the given task.

Algorithms are usually written using a common English language. For example, look at this maze where Karel needs to collect the gears and return to his home square:



Karel in his workshop.

This task can be solved using the following algorithm:

Algorithm
1: Make two steps forward.
2: Collect the gears.
3: Turn around.
4: Make three steps forward.

Translating an algorithm to a particular programming language
yields a *computer program (computer code)*.

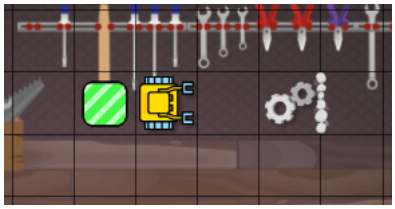
A computer program (code) is an *implementation* of the given algorithm. The process of translating an algorithm to a computer program is called *coding*. A computer program is formed by one or more commands. Here is the Karel program corresponding to the above algorithm:

PROGRAM 2.1. Collect the gears and return to the home square!

```
1 go
2 go
3 get
4 left
5 left
6 go
7 go
8 go
```

2.7. Usually, a task has more than one solution

Almost every task has several solutions. Let's stay with the previous one where Karel needs to collect the gears and return to his home square:



Karel in his workshop.

Check that this program also solves the task!

PROGRAM 2.2. A different solution to the same task

```
1 go
2 go
3 left
4 left
5 get
```

```
6 go
7 go
8 go
```

And here is yet another solution:

PROGRAM 2.3. Third solution to the same task

```
1 go
2 go
3 left
4 get
5 left
6 go
7 go
8 go
```

And another solution:

PROGRAM 2.4. Fourth solution to the same task

```
1 go
2 go
3 get
4 right
5 right
6 go
7 go
8 go
```

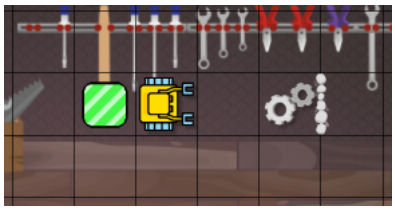
Can you find yet another solution?

Usually, one solution stands out by being the simplest or most elegant. But in this case, all the solutions are equally good.

2.8. How to recognize the best solution

If two different solutions require a substantially different numbers of operations, then the one which requires fewer operations is more efficient = better.

Check out the following solution which also solves the above task. Can you visualize in your mind step by step what the robot is doing? Here is the maze again for reference:



Karel in his workshop.

PROGRAM 2.5. A solution which requires many more operations

```
1  go
2  go
3  get
4  right
5  go
6  right
7  go
8  go
9  go
10 go
11 right
12 go
```

This program solves the task, but it needs 12 operation while all the above programs only needed 8.

In computer programming, one should always seek the most efficient solution (which requires the lowest number of operations).

2.9. Logical errors

Sometimes we think that we know how to solve a task, but our solution is wrong. Staying with the above example, let's say that we come up with the following algorithm:

Algorithm

1: Make three steps forward.
2: Collect the gears.
3: Turn around.
4: Make three steps forward.

Then we translate the algorithm to the following code, just to find out that it will not work!

PROGRAM 2.6. A program which does not work

```
1 go
2 go
3 go
4 get
5 left
6 left
7 go
8 go
9 go
```

Try to find the logical error before reading further!

An error in the algorithm is called a *logical error*.

2.10. Syntax rules and syntax errors

Every programming language has its own *syntax rules*. These are important because the code is parsed by a machine. It cannot be vague or arbitrary. For example, the first two syntax rules for Karel are:

(1) Always type one command per line.
(2) Every command must start at the beginning of line.

Return for a moment to the codes above and verify that all of them satisfy these syntax rules! Of course, programmers are human, and therefore they make all sorts of mistakes. For example the program shown below contains a syntax error because two commands are written on the first line:

PROGRAM 2.7. Syntax error (code violates Karel syntax rule #1)

```
1 go go
2 get
3 left
4 left
5 go
6 go
7 go
```

The following code violates Karel syntax rule #2 because of the indent on line 4:

PROGRAM 2.8. Syntax error (code violates Karel syntax rule #2)

```
1 go
2 go
3 get
4   left
5 left
6 go
7 go
8 go
```

Try to find three syntax errors in the following program!

PROGRAM 2.9. Karel program with three syntax errors

```
1 go
2 go
3 got
4 r1ght
5 night
6 go
7 go
8 go
```

Mistakes in syntax, such as misspelling a command, writing "1O" instead of "10", or forgetting that indentation matters are called *syntax errors*.

2.11. Debugging and where did the word "bug" come from

Mistakes of either kind (logical errors or syntax errors) are called *bugs* and the procedure of eliminating them is called *debugging*. Depending on how careful one was while preparing the algorithm and writing the program, debugging takes either a short time or a long time. It does not happen often that a program works correctly right away.

Logical errors are much harder to fix than syntax errors. Therefore, always think hard about the algorithm before coding it!

BTW, do you know why programming errors are called bugs? The first computer "bug" was a real bug — a moth. In 1947, Rear Admiral Grace Murray Hopper was working on the Harvard University Mark II Aiken Relay Calculator, an electromechanical computer. Grace Hopper tells the story: "Things were going badly; there was something wrong in one of the circuits of the long glass-enclosed computer. Finally, someone located the trouble spot and found a moth trapped between points at Relay #70 on Panel F. Using ordinary tweezers, we removed the two-inch moth. From then on, when anything went wrong with a computer, we said it had bugs in it."



Rear Admiral Grace Murray Hopper.

Grace Hopper's legacy was an inspiring factor in the creation of the Grace Hopper Celebration of Women in Computing. Held yearly, this conference is designed to bring the research and career interests of women in computing to the forefront. The USS Hopper (DDG-70) is named in honor of Rear Admiral Grace Murray Hopper.

2.12. Karel's bag

Karel has a bag where he stores all objects that he collects. Let's return one more time to his workshop. This time there are four objects on the floor - a phone, watch, radio and a light bulb:

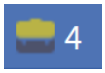


Karel in his workshop.

The robot can collect them using the following program:

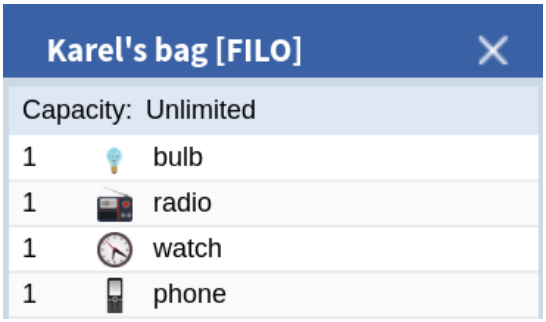
```
1 go
2 get
3 go
4 get
5 go
6 get
7 go
8 get
```

In the next section we will show you how to shorten it to only three lines. For now, we will use it in this form. When the program finishes, Karel will have four objects in his bag:



The "bag" icon shows how many objects are in Karel's bag.

When clicking on the icon, one can see all the objects. The most recently added ones are on top:

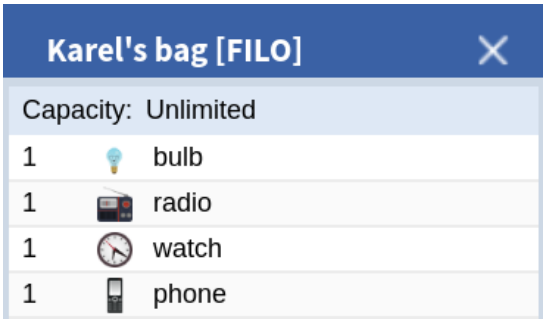


The contents of Karel's bag.

2.13. FILO (stack) and FIFO (queue)

Have you noticed the word FILO in the last image? It means that Karel's bag is in FILO mode. FILO means *stack* in computer science. Let's explain briefly what it does.

FILO stands for "First In Last Out" - objects which are inserted in some order will be removed in reverse order. In other words, the object which was inserted first will be removed last, and the object which was inserted most recently will be removed first. Recall that in the previous section, the four objects were collected in the following order (newest are on top):



Karel carries four objects.

Let's execute a program which makes Karel remove all objects from his bag and place them on the floor again:

```
1 put
2 go
3 put
4 go
5 put
6 go
7 put
8 go
```

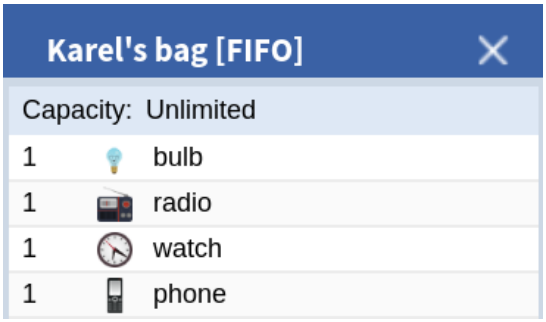
Since the bag is in FILO mode, objects are removed from the newest to the oldest. So, the light bulb is removed first and the phone last:



Outcome of the program.

FIFO mode

FIFO means *queue* in computer science. Karel's bag can be switched to FIFO mode in the Maze menu of the Designer. FIFO is an abbreviation of "First In First Out" - objects will be removed from the bag in the same order as they were inserted. Here is the contents of the bag again, with the newest items on top:



Karel's bag is now in FIFO mode.

When the last program is executed again, the outcome will be different - the phone will be removed from the bag first and the light bulb last:



Objects are removed in the same order as they were inserted.

2.14. Karel is case-sensitive

The Karel language is influenced by Python and as such, it is *case-sensitive*. This means that the case of letters matters. For instance, the Karel command to make one step forward is `go`. Any other version such as `Go`, `GO` or `gO` will not work - typing them will result into an error message stating that you are attempting to use an unknown command.

2.15. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 2.1. *In Manual mode, Karel can be controlled using*

A buttons in the Karel app.
B the keyboard.
C written programs.
D voice commands.

QUESTION 2.2. *When Karel faces East, what direction will he face after turning right?*

A North
B East
C South
D West

QUESTION 2.3. *Karel faces West. What direction will he face after turning left three times?*

A North
B East
C South
D West

QUESTION 2.4. *What command does Karel use to move one step forward?*

A forward
B step
C go
D march

QUESTION 2.5. *What command does Karel use to turn left?*

A leftturn
B turnleft
C goleft
D left

QUESTION 2.6. *What command does Karel use to place objects on the ground?*

A place
B drop
C insert
D put

QUESTION 2.7. *What command does Karel use to pick up objects from the ground?*

A collect
B get
C lift
D pick

QUESTION 2.8. *In Manual mode, the Shift key represents the command*

A go
B left

C put
D get

QUESTION 2.9. *What is the equivalent to turning left?*
A Turning right two times.
B Turning right three times.
C Turning right four times.
D Turning right five times.

QUESTION 2.10. *What is the equivalent to turning right?*
A Turning left two times.
B Turning left three times.
C Turning right five times.
D Turning left four times.

QUESTION 2.11. *The computer icon*  *represents*
A the number of operations done.
B the number of steps made.
C the number of lines in the program.
D program duration in seconds.

QUESTION 2.12. *In Programming mode the button*  *is used to*
A run the program.
B step through the program.
C stop the program.
D erase the program.

QUESTION 2.13. *Karel makes three steps forward, turns back, and returns to the square where he started. How many steps did he make?*
A 4
B 5
C 6
D 7

QUESTION 2.14. *In the previous question, how many operations did Karel do?*
A 6
B 7
C 8
D 9

QUESTION 2.15. *What statements are true about algorithms and programs?*

A An algorithm is typically written in plain English language.

B A program is typically written in plain English language.

C An algorithm is an implementation of a program.

D A program is an algorithm rewritten into a concrete programming language.

QUESTION 2.16. *What are the two basic syntax rules of the Karel language?*

A Type all commands using lowercase letters.

B Always type one command per line.

C Every command must start at the beginning of line.

D Type all commands using uppercase letters.

QUESTION 2.17. *What of the following errors are logical errors?*

A Making an indent where it should not be.

B Making a left turn instead of a right turn.

C Typing two commands on the same line.

D Making three steps instead of two.

QUESTION 2.18. *What of the following errors are syntax errors?*

A Misspelling a command, such as typing `rlght` *instead of* `right`.

B Crashing Karel into a wall.

C Typing `1O` *instead of* `10`.

D Using a command which Karel does not know.

QUESTION 2.19. *What was the first programming "bug"?*

A A mosquito.

B A butterfly.

C A moth.

D A bee.

QUESTION 2.20. *When Karel's bag is in FILO mode (stack), then:*

A Object which was inserted first will be removed last.

B Object which was inserted first will be removed first.

QUESTION 2.21. *When Karel's bag is in FIFO mode (queue), then:*

A Object which was inserted first will be removed last.

B Object which was inserted first will be removed first.

3. Counting Loop

In this chapter you will learn:

- What the counting loop is.
- How to use the keyword `repeat` to repeat single commands.
- How to comment your code, and why commenting your code is important.
- How to use the `repeat` loop to repeat sequences of commands.
- How to quickly and correctly identify a repeating pattern.
- What to do if a repeating pattern does not start right away.
- What to do if additional operations are needed after a repeating pattern ends.
- How to use nested `repeat` loops.

3.1. Counting loop and the keyword `repeat`

The concept of *counting loop* is present in all procedural programming languages such as Python, C/C++, Java, Javascript, Fortran and others. Counting loops allow us to repeat a command, or a sequence of commands, a given number of times.

Importantly, the number of repetitions (cycles) must be **known in advance**.

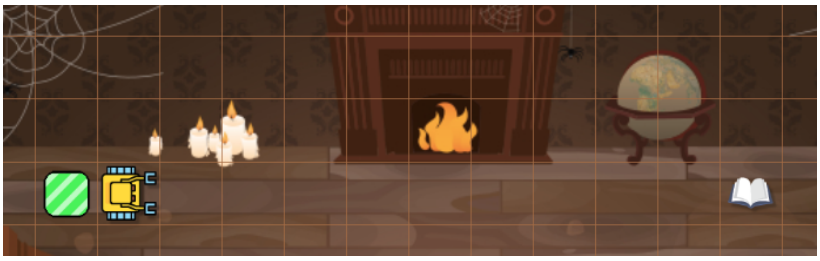
In the Karel language, the counting loop is defined using the keyword `repeat`. For example, the following code will repeat 10 times the command `go`:

PROGRAM 3.1. Make 10 steps forward!

```
1 repeat 10
2   go
```

Notice that the keyword `repeat` is followed by the number of repetitions (cycles), and that the body of the loop is **indented**. By *body of the loop* we mean the command or sequence of commands to be repeated.

Most "real" languages define the counting loop using the keyword `for`. The Karel language also uses the keyword `for`, in the same way Python does - this will be explained later. For now let us now look at the following maze:



Karel is in the library.

Suppose that the robot needs to collect the book and return to his home square. Here is the algorithm for that:

Collect the book and return to the home square!

1: Make 10 steps forward.
2: Collect the book.
3: Turn around.
4: Make 11 steps forward.

If we did not have the `repeat` loop, we would implement the algorithm as shown below. The hash symbol # is used to introduce comments - the rest of the line after this symbol is ignored by the interpreter.

PROGRAM 3.2. Implementation without the `repeat` loop

```
1  # Make 10 steps forward:
2  go
3  go
4  go
5  go
6  go
7  go
8  go
9  go
10 go
11 go
12 # Collect the book:
13 get
14 # Turn around:
15 left
16 left
17 # Make 11 steps forward:
```

```
18 go
19 go
20 go
21 go
22 go
23 go
24 go
25 go
26 go
27 go
28 go
```

However, this program is too long. Let's make it shorter!

PROGRAM 3.3. Program 3.2 rewritten using the `repeat` loop

```
1  # Make 10 steps forward:
2  repeat 10
3    go
4  # Collect the book:
5  get
6  # Turn around:
7  repeat 2
8    left
9  # Make 11 steps forward:
10 repeat 11
11   go
```

Again, notice that the command `go` on lines 3 and 11, as well as the command `left` on line 8 are indented. This is very important because it makes it clear which commands belong to the body of the loop.

Forgetting to indent the body of a loop is a syntax error.

3.2. Commenting your code

Good programmers use many comments in their programs. Namely, while writing your program, you know exactly what every line does (at least you should). But when you go do something else, and return to it after several weeks, the program might as well be written by somebody else. Then, the comments will help you to quickly refresh your

memory. And of course, comments are absolutely crucial when you work in a team and other team members are using your code.

> Comments and empty lines do not count towards the length of the program.

So, the above Program 3.2 has 24 lines, and the shorter Program 3.3 only has 7.

3.3. Repeating a sequence of commands

The body of a loop can contain a sequence of commands, as long as all commands in this sequence are indented. Then the entire sequence is repeated. For illustration, here is a maze with an anchor, chest, and three gold coins:



Karel is on a ship.

Karel's task is to collect the three gold coins, and put them in the chest at the end. This can be done using the following program. Step through it in your mind to make sure it works correctly!

PROGRAM 3.4. Collect the gold coins and put them in the chest!

```
1 # Collect the coins:
2 repeat 3
3   go
4   get
5   go
6   left
7 # Put them in the chest:
8 repeat 3
9   put
```

After the program ends, the robot is standing over the chest, facing South:



Karel's position after the program ends.

3.4. Indentation matters

Forgetting one indent can change the outcome of your program completely. For a moment, let's assume that we committed a syntax error and forgot to indent the command `left` on line 5:

PROGRAM 3.5. Missing indent on line 5

```
1 repeat 3
2   go
3   get
4   go
5 left
6 repeat 3
7   put
```

But the forgotten indent means that the command `left` is no longer part of the loop! This will change everything, and the robot will crash into the wall:



One mistake in indentation caused the robot to crash.

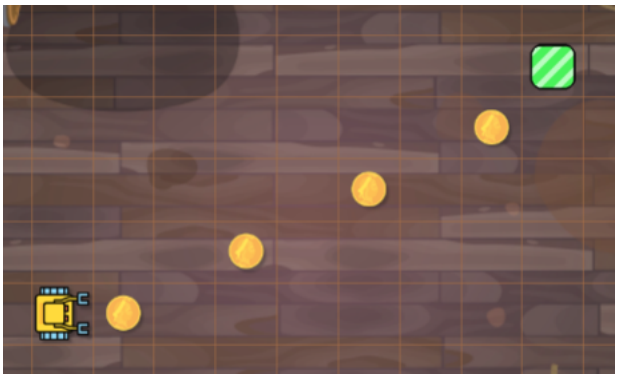
The following screenshot shows the corresponding error message, including the line where the error occurred:

```
Run-time error on line 2:
Ouch, you crashed me!
```

Error message generated by Program 3.5.

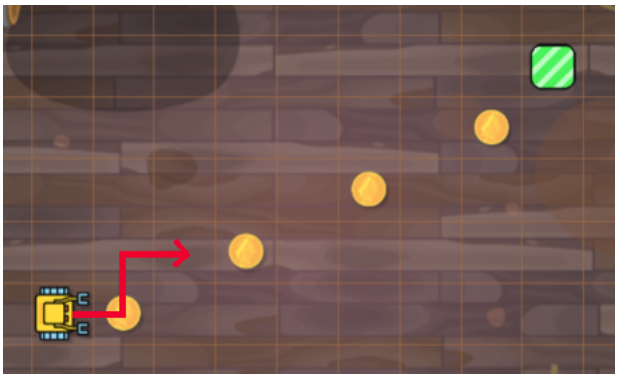
3.5. How to figure out repeating patterns quickly and correctly

Identifying the repeating pattern (the body of the loop) is a fundamental skill of computer programming. Any time spent practicing this skill is time well spent. Let's look at the following example, where Karel's task is to collect four coins and enter the home square:



Karel needs to collect four coins.

It is clear that there will be four cycles because there are four coins. But what commands should be repeated? The following image helps us to answer this question:



First cycle - Karel must collect one coin and get ready in front of the next one.

At the beginning, Karel stands in front of a coin. Therefore, at the end of the first cycle, he must also stand in front of a coin (the next one). And he must have collected one coin on the way. Only then he can repeat the same sequence of commands again. Hence, we have just figured out the body of the loop!

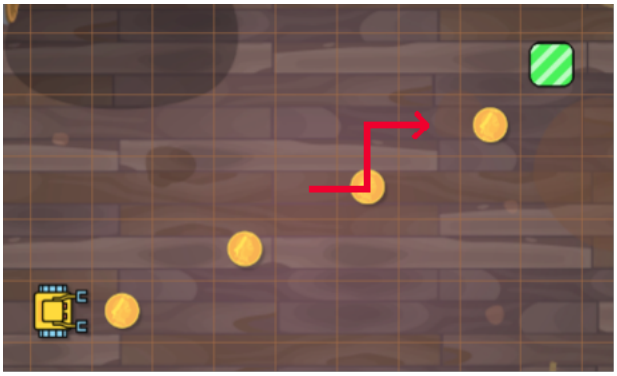
```
go
get
left
go
right
go
```

When stepping through these six lines for a second time, the robot collects the second coin and moves to the third one:



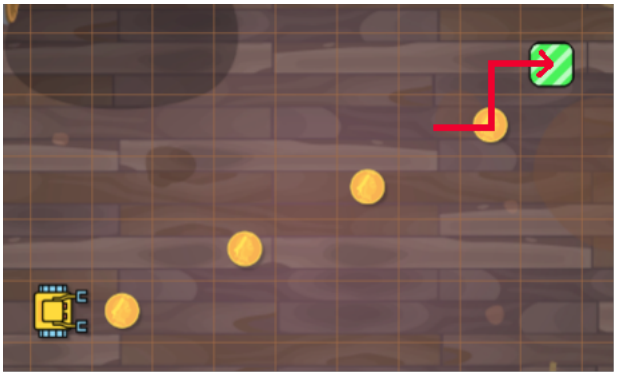
Second cycle.

This is the third cycle - Karel collects the third coin and moves to the last one:



Third cycle.

And one last time:



Fourth and last cycle.

We already know that there are four cycles, so the first line of the program will be `repeat 4`. After adding the above six lines as the body of this loop, we obtain the final program:

PROGRAM 3.6. Collect four coins and enter the home square!

```
1 repeat 4
2   go
3   get
4   left
5   go
6   right
7   go
```

3.6. Stashing coins

Let's look at another example where Karel needs to stash four coins in the chests:



Stashing coins.

Clearly, there will be four cycles, because there are four coins and four chests. But what commands will form the body of the loop?

Notice that Karel stands with his back to a chest. That must also be his final position after finishing the first cycle - except it will be the next chest which is to his right. And moreover, one coin must be already stashed in this chest. This tells us where Karel needs to go:



First cycle.

By now we have figured out the repeating pattern which is formed by the following 10 commands:

```
right
go
right
go
go
get
left
go
put
go
```

To verify that these 10 lines are correct, let's step through them in our mind for a second time:



Second cycle.

For a third time:



Third cycle.

And finally, one last time:



Fourth cycle.

Using the above 10 lines as the body of a `repeat` loop with four cycles, we obtain the final program:

PROGRAM 3.7. Move the four coins to the chests!

```
repeat 4
  right
  go
  right
  go
  go
  get
  left
  go
  put
  go
```

In order to figure out a repeating pattern, look where Karel is initially, and where he must be at the end of the first cycle.

3.7. Cooking potatoes

Karel's next task is to cook some potatoes.



Karel must put all potatoes in the pots.

With what you already know, it is easy to see the repeating pattern: At the end of the first cycle, the first potato must be in the first pot and Karel must be ready in front of the second one:



Repeating pattern.

Now we know how the body of the `repeat` loop will look like:

```
right
go
get
left
go
left
go
put
right
go
```

Using these 10 lines as the body of a `repeat` loop with 4 cycles yields the final code:

PROGRAM 3.8. Put all potatoes in the pots!

```
1  repeat 5
2    right
3    go
4    get
5    left
6    go
7    left
8    go
9    put
10   right
11   go
```

3.8. When it is not be possible to start a loop right away

Sometimes the repeating pattern does not start immediately. For example, consider the last example, but now with Karel standing at a different initial position:



Karel's initial position changed.

This is not a good initial position to start a loop. Therefore, let's change it! One way to do it is make two steps forward, turn right, make another step forward, and then turn left:

```
go
go
right
go
left
```

Then the robot's position will match his initial position from the last example, and we will be able to use the previous code:



Adjusting Karel's initial position so that a loop can be started.

Hence, the program to solve the new task will consist of the five commands shown above, followed by Program 3.8:

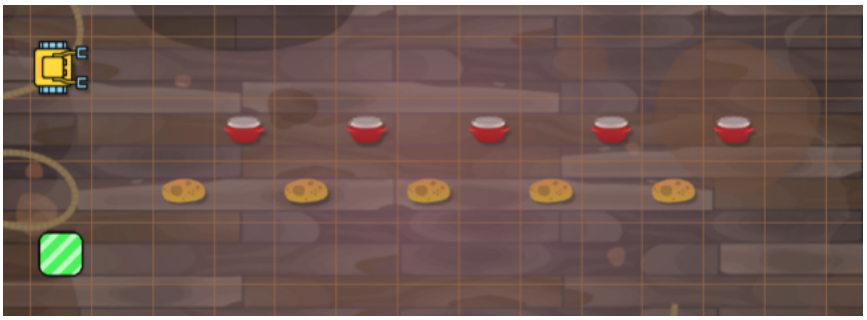
PROGRAM 3.9. Adjust your initial position and put all potatoes in the pots!

```
1  go
2  go
3  right
4  go
5  left
6  repeat 5
7    right
8    go
9    get
10   left
11   go
12   left
13   go
14   put
15   right
```

```
16    get
17    go
```

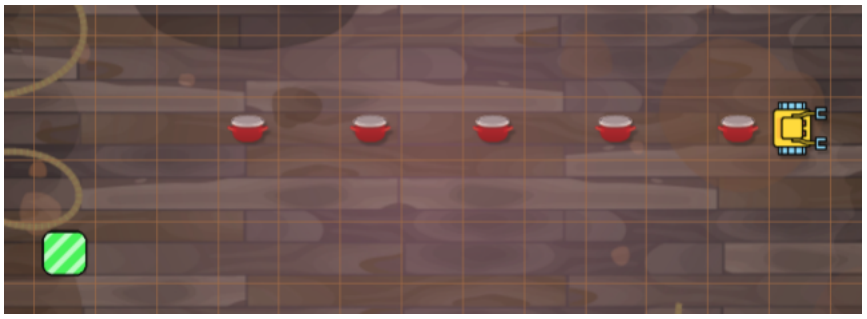
3.9. Additional commands might be needed after a loop finishes

Let's stay with the last example, but we added a home square where Karel needs to go after putting the potatoes in the pots:



Karel moreover needs to enter the home square at the end.

Clearly, we will be able to reuse Program 3.9 but some new commands will have to be added at the end. To figure out what these commands are, we must know where Karel will be after the loop finishes. The best way to find out is to step through the program in your mind. Then you will see that Karel will finish the loop standing on the right of the last pot, looking East:



Karel's position after finishing the loop.

Now it is easy to see that in order to get home, Karel needs to turn right, make two steps, turn right again, and make 12 more steps:

```
right
go
go
```

```
right
repeat 12
  go
```

Hence, here is the final program which consists of Program 3.9 and the above 6 lines:

PROGRAM 3.10. Get home after fulfilling the task!

```
1  go
2  go
3  right
4  go
5  left
6  repeat 5
7    right
8    go
9    get
10   left
11   go
12   left
13   go
14   put
15   right
16   get
17   go
18 right
19 go
20 go
21 right
22 repeat 12
23   go
```

3.10. Nested loops

Sometimes a repeating pattern can contain another repeating pattern. Such as in the following task where Karel needs to collect nine pearls which are grouped into three rows:



Karel is on the bottom of the ocean.

These tasks lead to *nested loops* - loops which contain other loops in their body. Writing programs with nested loops requires practice, so don't get frustrated or give up when you struggle initially. This is perfectly normal. However, some things can be done to make the solution of these tasks easier.

First, always start from the smaller repeating pattern - in this case, let's collect the first three pearls first:

```
repeat 3
  go
  get
```

It is important to step through this loop in your mind, to see where the robot will be after this loop finishes:



After finishing the loop.

As you can see, Karel cannot just start the same loop right away (because after making one step forward, there would be no pearl to collect. To get ready for it, he must make one step forward first:



After moving one step forward.

Then he can use the same loop as before to collect the next three pearls:

```
repeat 3
  go
  get
```

When the second loop is finished, he stands where the sixth pearl was:



After finishing the second loop.

Again, Karel needs to make one step forward to move in front of the last row of pearls:



After moving one step forward.

To collect the last three pearls, the robot can do the same as he already did twice before:

```
repeat 3
  go
  get
go
```

At the end, Karel's position will be:



Final position after completing the task.

Hence the final code to solve this task contains the above four lines, repeated three times:

PROGRAM 3.11. Collect three rows of pearls!

```
1 repeat 3
2   repeat 3
3     go
4     get
5   go
```

Notice that the commands in the inner loop are indented twice.

3.11. Revisiting previous programs

With the knowledge of nested loops, we can revisit and improve some of our previous programs. Let's start with Program 3.7 from page 35:

```
1  repeat 4
2    right
3    go
4    right
5    go
6    go
7    get
8    left
9    go
10   put
11   go
```

Here, lines 2 and 3 are repeated two times. Therefore, we can use a loop for them:

PROGRAM 3.12. Program 3.7 improved using a nested loop

```
1  repeat 4
2    repeat 2
3      right
4      go
5    go
6    get
7    left
8    go
9    put
10   go
```

Also Program 3.8 from page 36 can be improved in a similar way:

```
1  repeat 5
2    right
3    go
4    get
5    left
6    go
7    left
8    go
9    put
```

```
10    right
11    go
```

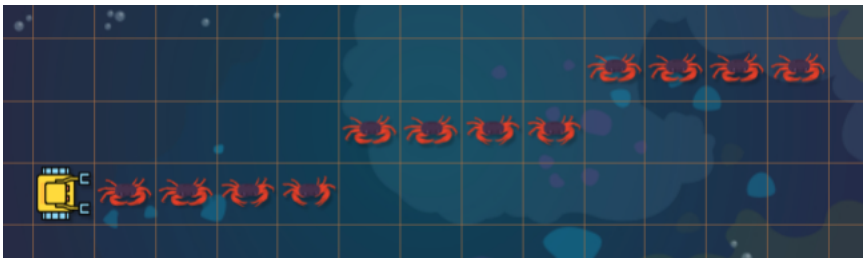
Here, lines 5 and 6 are repeated two times. Therefore we can shorten it using a loop:

PROGRAM 3.13. Program 3.8 improved using a nested loop

```
1  repeat 5
2    right
3    go
4    get
5    repeat 2
6      left
7      go
8    put
9    right
10   go
```

3.12. Crab cake

Today Karel needs to collect three rows of crabs:



Karel is collecting crabs.

Each row has length four. To collect the first one, Karel can use the following loop:

```
repeat 4
  go
  get
```

Then, to get ready in front of the next row, he needs to make a left turn, one step forward, and a right turn:

```
left
go
```

```
right
```

Finally, the solution code for this task will consist of the above two codes put together, and repeated three times:

PROGRAM 3.14. Collect three rows of crabs!

```
1 repeat 3
2   repeat 4
3     go
4     get
5   left
6   go
7   right
```

3.13. Starfish square

This time Karel needs to collect starfish which form a 6×6 square:



Karel is collecting starfish.

This task is a bit tricky. Clearly, the square has four sides, and therefore the outer loop will have four cycles. Each side has length six, but in order to create a repeating pattern, Karel will only be collecting five starfish on each side. Think about it for a moment: If he collected six starfish on the first side, then on the next one he would only have five. So he would not be able to collect six as in the first cycle. That would not work!

Moreover, look at Karel's initial position above. If we wanted to start the inner loop right away, then Karel's position at its end would have to be as follows:



Inoptimal position outside of the square.

This could be done, but the final code would be longer and more complicated than necessary. It is far better to move Karel one step forward to a better starting position as the first thing:



Better initial position.

Now, after collecting five starfish and turning left, he is ready for a second cycle of the outer loop! Notice that there is a starfish beneath the robot as it was at the beginning of the first cycle:



After collecting the first five starfish and turning left, Karel is ready to repeat.

The following image shows Karel after finishing the nested loops. He stands on the left of the home square and points North:



After finishing the nested loops.

Therefore, after the nested loops are finished, he needs to turn right and make one step forward:

```
right
go
```

Here is the final code:

PROGRAM 3.15. Collect all the starfish!

```
go
repeat 4
  repeat 5
    get
    go
  left
right
go
```

Always think about the algorithm first. Implementing a good algorithm is joy. Implementing a bad algorithm is a nightmare.

3.14. Triple-nested loops

Some tasks can involve triple-nested repeating patterns. For illustration, let's take the previous starfish square, but now with three starfish per grid square instead of one. Karel's task is again to collect all of them:



Now there are three starfish per grid square.

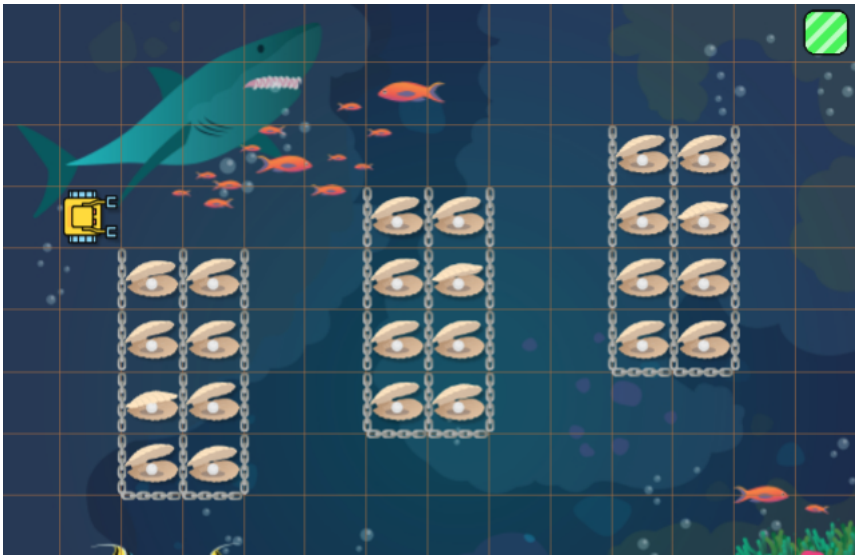
The solution program will be very similar to Program 3.15 except that the `get` command needs to be replaced with a `repeat` loop with three cycles. As a result, the program will contain a triple-nested loop:

PROGRAM 3.16. Collect all the starfish!

```
go
repeat 4
  repeat 5
    repeat 3
      get
    go
  left
right
go
```

3.15. Pearl necklace

Let's show one more example of a task that leads to triple-nested loops. This time Karel needs to collect pearls from three boxes:



Three boxes of pearls.

With what you already know, it should not be difficult to step through the following program in your mind, and verify that it works correctly:

PROGRAM 3.17. Collect three boxes of pearls!

```
repeat 3
  repeat 2
    go
```

```
    right
    repeat 4
      go
      get
    repeat 2
      left
    repeat 4
      go
    right
  repeat 2
    go
  left
  go
  right
```

In theory, the number of nesting levels is unlimited. But even in the most complicated real-world programs it hardly ever exceeds three.

3.16. Even more levels of nesting

To show just one example which involves four levels of nested loops, let's return to our three boxes of pearls. But this time, there will be five pearls per grid square instead of one.



Previous task, but with five pearls per grid square.

And this is the corresponding solution program. It was obtained from Program 3.17 by replacing the `get` command with a `repeat` loop with five cycles:

PROGRAM 3.18. A program with four levels of nested loops

```
1  repeat 3
2    repeat 2
3      go
4      right
5      repeat 4
6        go
7        repeat 5
8          get
9      repeat 2
10       left
11     repeat 4
12       go
13     right
14   repeat 2
15     go
16   left
17   go
18   right
```

3.17. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 3.1. *Is commenting your code a good practice?*

A *Yes, because it helps you understand what you code does.*

B *Yes, because it helps others understand what you code does.*

C *No, because comments make the code too long.*

D *No, because writing comments takes precious time.*

QUESTION 3.2. *What symbol uses Karel to introduce a comment?*

A `%`

B `//`

C `$`

D `#`

QUESTION 3.3. *What keyword does Karel use for the counting loop?*

A `loop`
B `repeat`
C `cycle`
D `while`

QUESTION 3.4. *What is the* body *of the counting loop?*

A *The number of repetitions.*
B *The first command to be repeated.*
C *One or more commands which are repeated.*
D *The last command to be repeated.*

QUESTION 3.5. *How is the body of the counting loop defined in Karel?*

A *The command(s) are enclosed in square brackets.*
B *The command(s) are enclosed in curly braces.*
C *The command(s) are written to begin at the beginning of the line.*
D *The command(s) are indented.*

QUESTION 3.6. *Karel is facing North. What direction will he face when the following program finishes?*

```
1 repeat 4
2   go
3   left
```

A *West*
B *East*
C *North*
D *South*

QUESTION 3.7. *Karel is facing North. What direction will he face when the following program finishes?*

```
1 repeat 4
2   go
3 left
```

A *West*
B *East*
C *North*
D *South*

QUESTION 3.8. *Karel faces West. Where will he face when the following program finishes?*

```
repeat 365
  left
```

A North
B South
C East
D West

QUESTION 3.9. *How many steps will Karel make?*

```
repeat 4
  go
repeat 2
  go
```

A 6
B 7
C 8
D 9

QUESTION 3.10. *How many steps will Karel make now?*

```
repeat 4
  repeat 2
    go
```

A 6
B 7
C 8
D 9

QUESTION 3.11. *And finally, how many steps will Karel make with this program?*

```
repeat 3
  repeat 2
    repeat 2
      go
```

A 7
B 8
C 12
D 24

4. Conditions

In this chapter you will learn:

- What are conditions and why they are used in computer programming.
- Why does Karel need conditions.
- How to use the `if` statement and the `else` branch
- How to detect obstacles, collectible objects, and containers.
- New function `print` and a new command `pass`.
- About the sensors `north`, `empty` and `home`.
- How to use the full `if-elif-else` statement.
- How to combine two or more conditions using the keywords `not`, `and`, and `or`.

4.1. What are conditions?

Sometimes we need our program to contain parts which are not executed always, but only in certain situations (conditionally). Imagine, for example, that our program asks the user to enter a credit card number. Users are known to make mistakes. So, when the credit card number is received, the program needs to execute a condition, checking whether the number has 16 digits. If so, then the program can proceed to other tests (is the credit card number valid?) But if not, then the program needs to ask the user to correct his/her input. Conditions are present in every programming language, not only to check user data but for many other types of tests as well.

4.2. Why does Karel need them?

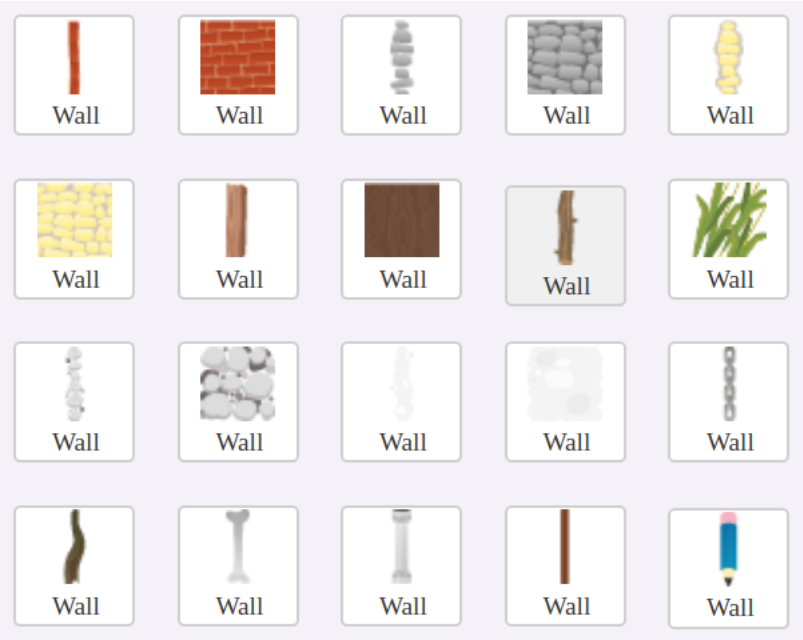
Karel needs conditions to check his sensors. You will learn more about them soon. With the help of his sensors, he can:

- Detect collectible objects and containers on the ground beneath him,
- detect and avoid obstacles in front of him,
- find out which way is North (and from there, any other direction),
- check his GPS coordinates and know where he is in the maze,
- check whether he is at his home square,

- check whether his bag is empty,
- pass through unknown mazes, etc.

4.3. Karel's `wall` sensor and the `if` statement

Karel's mazes can feature many different types of walls:

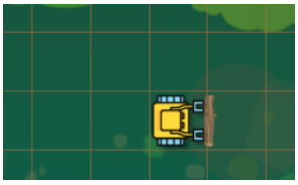


Various types of walls in Karel's mazes.

The robot has a sensor named `wall` which helps him detect a wall in front of him. Here "in front of him" means that the wall is *right in front of him* - he would crash into it if he made one more step forward.

Karel's `wall` sensor can only detect a wall which is right in front of the robot.

Now imagine that Karel is in a new environment that he does not know. He wants to make one step forward. Keep in mind that he does not have eyes, so he cannot see what you see. So, he cannot know if there is a wall in front of him or not:



Karel is in the jungle.

He knows that he can't just execute

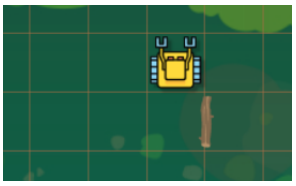
```
1 go
```

because if there was a wall in front of him, he would blindly crash into it. So, instead, he uses the keyword `if` to check his sensor `wall` before making the step:

PROGRAM 4.1. Karel is checking for a wall before making a step

```
1 if wall
2   left
3 go
```

And this was a very good decision indeed, because otherwise he would have crashed! The outcome of Program 4.1 is:



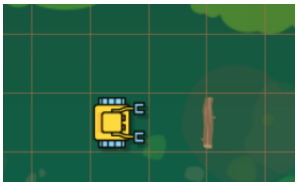
Karel detected and avoided a wall.

Notice that the keyword `left` was **indented** in Program 4.1 - that's because it forms the *body of the condition*.

The body of a condition is only executed when the condition is satisfied.

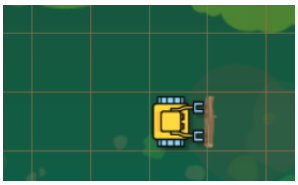
4.4. What happens if there is no wall?

Now let's see what happens when the condition is not satisfied:



Karel's `wall` sensor cannot detect a wall which is one step away.

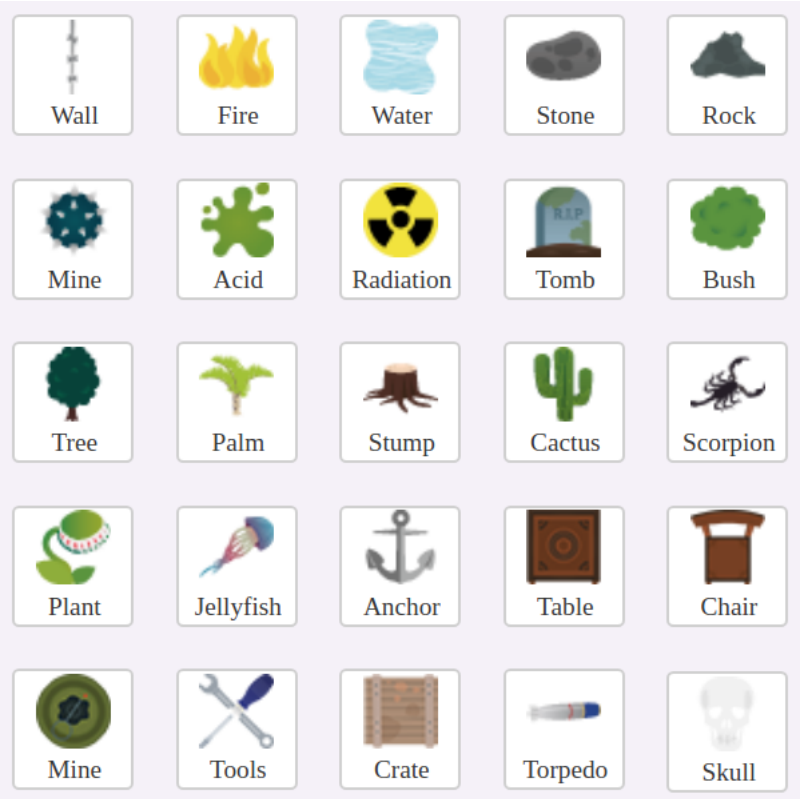
When Program 4.1 is executed, the condition is not satisfied. Therefore the conditional code - in this case just one command `left` on line 2 - is skipped. Hence, Karel just executes the command `go` on line 3 and makes one step forward:



Karel safely made one step forward.

4.5. Other types of obstacles

Besides walls, one can find in Karel's mazes various other *obstacles*:



Other types of obstacles.

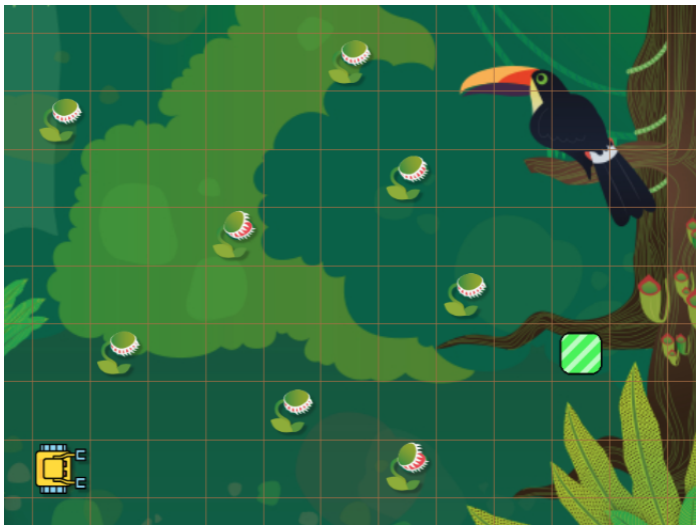
Karel has a dedicated sensor to detect each one. The names of the sensors exactly correspond to the names of the obstacles. For example, the `fire` sensor helps him detect fire, the `water` sensor helps him detect water, etc. All obstacle sensors work analogously to the `wall` sensor, so the robot can detect fire only when the fire is right in front of him, etc.

There are two types of obstacles - "thin ones" which only fill an edge between two grid squares, and "thick ones" which fill an entire grid square.

All of the thin ones are walls. Some of the thick ones are walls as well, but mainly these are other objects such as fire, water, plant, scorpion, tools, skull, etc.

4.6. Combining loops and conditions

As you have seen in Section 3.10 (page 39), loops can contain other loops. Then we talk about *nested loops*. In this section we want to show you that loops can also contain conditions. To illustrate it, let's have Karel play the Game of Arrows. How is this game played?



The Game of Arrows.

Karel knows that his home is exactly 33 steps away. Dangerous carnivorous plants are located randomly in the maze. He only knows that when he encounters one, he must turn left - so in fact, each plant serves as a left arrow.

Here is a program which solves this puzzle:

PROGRAM 4.2. Karel plays the Game of Arrows

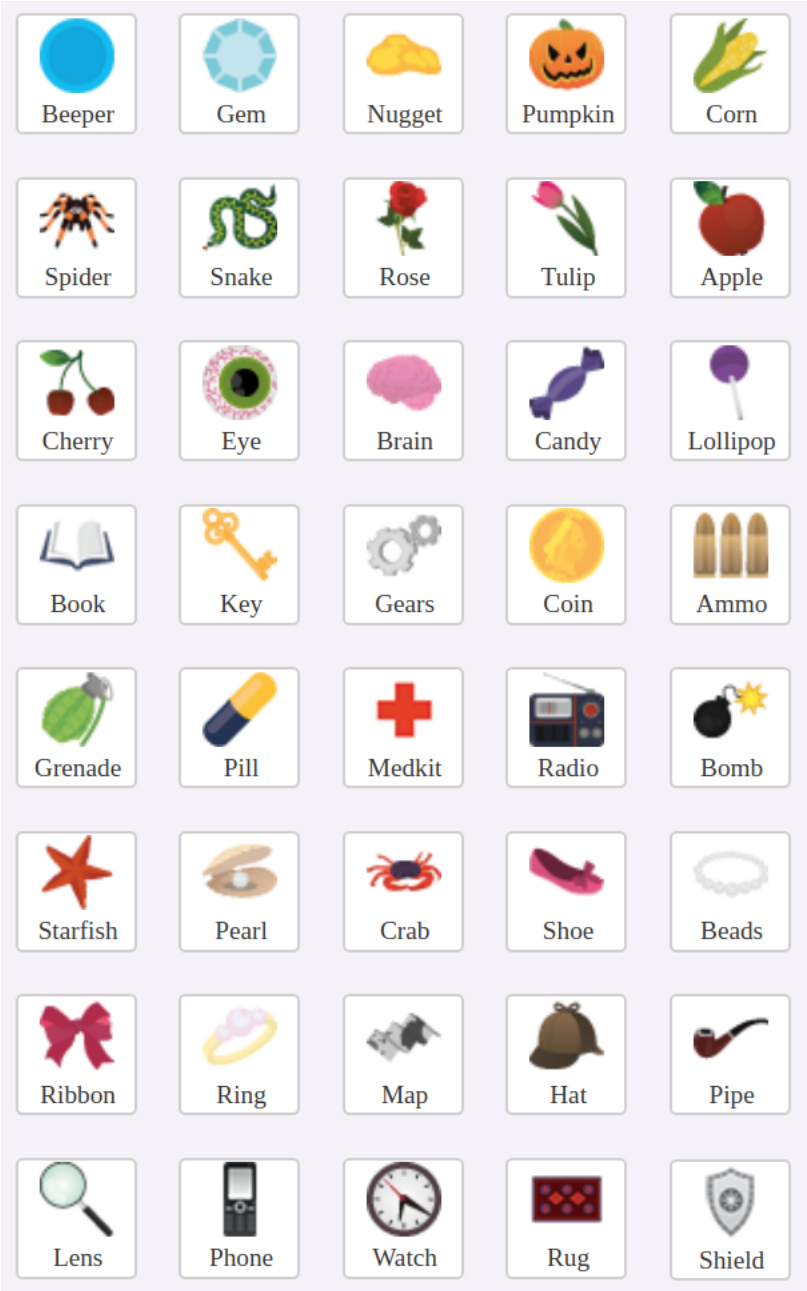
```
1 repeat 33
2   go
3   if plant
4     left
```

The point of this example was to show you that when a loop contains a condition, then the body of the condition must be **indented twice**. We are talking about the command `left` on line 4. You already have seen the accumulation of indents in nested loops in Section

3.10 (page 39). Of course, conditions can contain loops, and conditions can also contain other conditions. And all these can be nested. We will see more examples later.

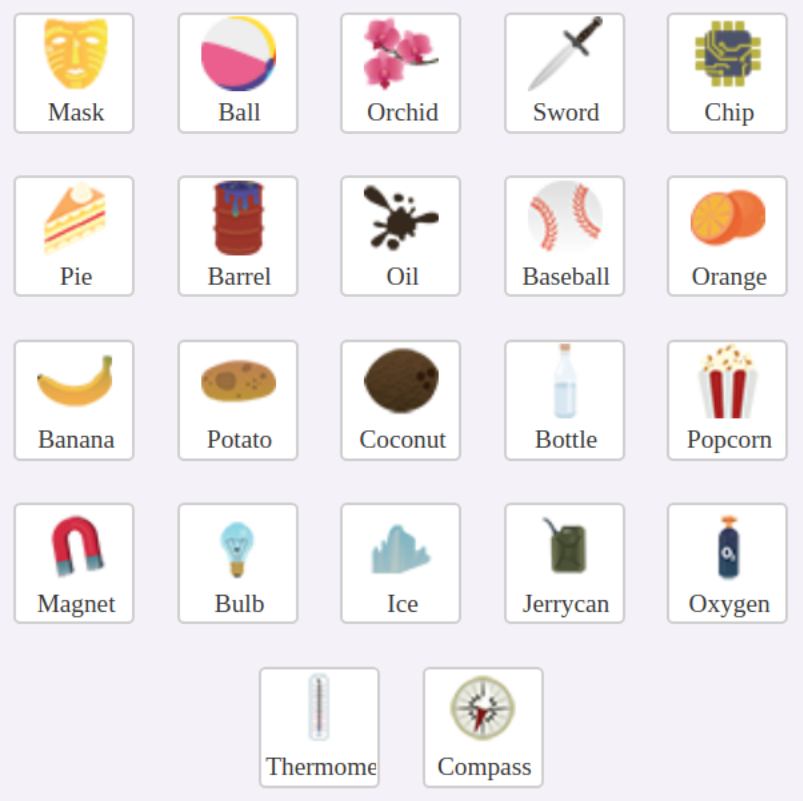
4.7. Collectible objects

In addition to obstacles, one can find various *collectible objects* in Karel's mazes:



Collectible objects - part 1.

Detecting collectible objects works differently from obstacles - Karel must stand above a collectible object (in the same square) to be able to detect it. He has a dedicated sensor for each collectible object, and the name of the sensor exactly corresponds to the name of the object. For example, he can use the sensor `beeper` to detect beepers on the ground, the sensor `spider` to detect spiders, etc.



Collectible objects - part 2.

4.8. Collecting bananas

For illustration, let's look at the following maze where several bananas lie at random positions between Karel and his home square. The home square is 12 steps ahead. The robot's task is to collect them all:



Karel is collecting bananas.

The robot has a sensor `banana` which helps him detect bananas on the ground. Here is a program that solves this task:

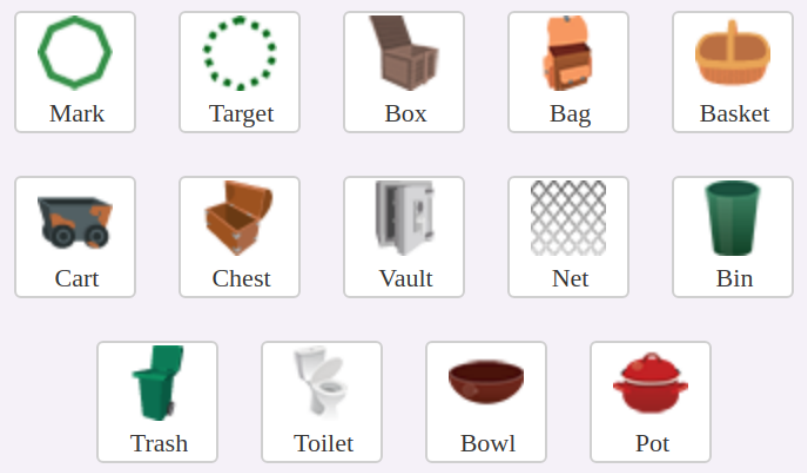
PROGRAM 4.3. Collect all the bananas!

```
1 repeat 12
2   if banana
3     get
4   go
```

Notice that the condition is part of the body of a `repeat` loop - therefore, the command `get` on line 3 is indented twice.

4.9. Containers

The third type of objects one can find in Karel's mazes are *containers*. Containers are useful when one wants Karel to place collectible objects at a designated place in the maze which he does not know in advance.



Karel can place objects in these containers.

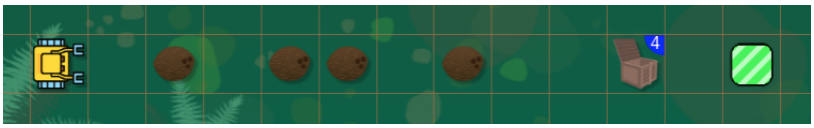
When installing a container into the maze, one can define its capacity (number of objects the container can store) as well as the type of objects which can be stored in it.

Containers are especially useful when creating games,
where one of the possible game goals is *fill all containers*.

The process of building mazes and creating games will be explained in detail in Appendices A.2 and A.3.

4.10. Collecting coconuts

You already saw Karel collect bananas, now let's enlist his help collecting coconuts! There are four coconuts placed randomly between the robot and his home square. The home square is 12 steps away. Also, there is a box somewhere between the last coconut and the home square. The blue number 4 located close to the box indicates that the box can store up to four bananas. Karel's task is to collect all the coconuts and put them into the box:



Karel is collecting coconuts.

The program to solve this task is shown below. Notice that it contains a loop which contains a condition which contains another loop. Hence, the command `put` on line 6 is indented three times:

PROGRAM 4.4. Put all the coconuts in the box!

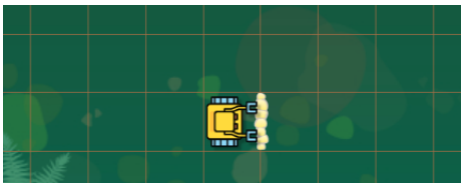
```
1 repeat 12
2   if coconut
3     get
4   if box
5     repeat 4
6       put
7   go
```

4.11. The `else` branch

So far we have used the `if` statement to just skip some code when the condition was not satisfied. But sometimes one needs to also execute alternative code. This is exactly what the `else` branch does!

> The `else` branch contains alternative code to be executed when the condition is not satisfied.

Imagine that Karel wants to move by one square to the East. If there is a wall in front of him, he needs to go around it. If not, he can just make one step forward:



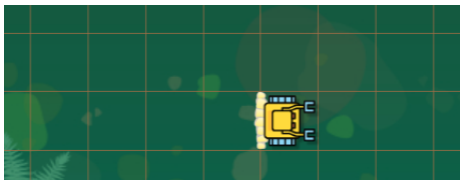
Karel needs to move by one square East, bypassing a wall if necessary.

Here is the corresponding code:

PROGRAM 4.5. Move by one square East, bypassing a wall if necessary!

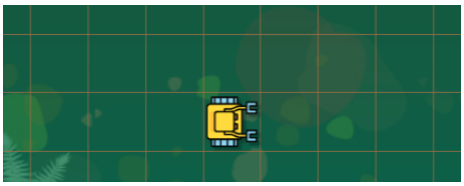
```
1 if wall
2   left
3   go
4   repeat 2
5     right
6     go
7   left
8 else
9   go
```

Since the condition on line 1 was satisfied, lines 2-7 were executed and line 9 was skipped. This is Karel's position after the program ends:



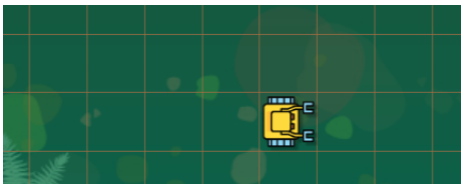
Karel went around the wall successfully.

Now let's see what the program will do when there is no wall:



This time Karel's way is clear.

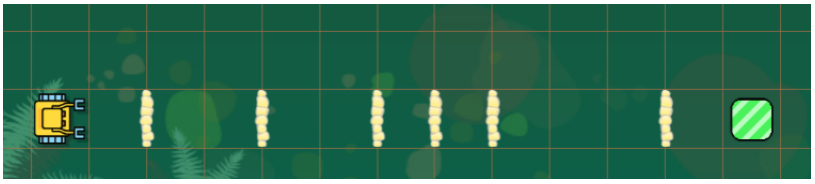
Since there is no wall in front of the robot, the condition on line 1 is not satisfied. Therefore lines 2-7 are skipped, line 9 is executed, and Karel just makes one step forward:



Karel's position after the program ends.

4.12. Hurdle race

Today Karel participates in a hurdle race! He must get to his home square, jumping over (i.e., going around) several randomly placed walls. His home square is 12 steps ahead:



Hurdle race.

The solution program is based on Program 4.5:

PROGRAM 4.6. Jump over all hurdles and enter the home square!

```
repeat 12
  if wall
    left
    go
    repeat 2
      right
      go
    left
  else
    go
```

4.13. Function `print`

As you already know, Karel shares significant functionality with Python. One such function is the function `print` which Karel can use to display text messages. For example, typing

PROGRAM 4.7. Display a greeting!

```
print("Hello world, I am Karel!")
```

will generate the following text output:

```
Hello world, I am Karel!
```

Notice:

> The text to be displayed by the `print` function
> needs to be enclosed in quotation marks.

4.14. Sensor `north`

Karel has a sensor `north` to detect whether he is facing North (top of your screen).

> The condition `if north` is only satisfied when the robot faces North.

Let's try this! We will execute the code below with Karel facing various directions on the compass:

PROGRAM 4.8. Tell if you are facing North!

```
1 if north
2   print("I am facing North!")
3 else
4   print("I am not facing North.")
```

Here are the outcomes:



"I am facing North!"



"I am not facing North."



"I am not facing North."



"I am not facing North."

4.15. Command `pass`

Another command which is the same in Karel and in Python is the command `pass`.

Command `pass` does *nothing*.

This command is often used as a placeholder for a future code. Or to really do nothing, as we will show in the next section.

4.16. Finding North

Imagine that Karel is facing a random direction - he does not know whether he faces North, South, East or West. This program will make sure that the robot turns to face North:

PROGRAM 4.9. Turn to face North!

```
1 repeat 3
2   if north
3     pass
4   else
5     left
```

But why does the program begin with `repeat 3` when there are four major directions on the compass? Well, if the robot already faces North, then he does not have to turn at all. If he faces East, then he must turn left once to face North. If he faces South, then he must turn left two times. And when he faces West, he must turn left three times. So he never needs to turn four times!

Imagine that Karel faces North, and step through the above program in your mind: The condition on line 2 is always satisfied, so the program executes `pass` three times. The `left` command is skipped every time.

Now let's say that the robot faces East. Then the condition is not satisfied in the first cycle, and therefore Karel turns left. But then he faces North, so in the remaining two cycles the condition is satisfied, and so the program executes `pass` in the second and third cycles of the loop.

Analogously, when the robot faces South, then the condition is not satisfied in the first cycle, and Karel turns left. Then he faces East. So, the condition is not satisfied in the second cycle either, and he turns left again. Then he faces North, so in the third cycle the condition is satisfied and the robot does not turn anymore.

When Karel faces West, then the condition will never be satisfied. In the first cycle he turns left to face South, in the second cycle he turns left to face East, and finally in the third cycle he turns left to face North.

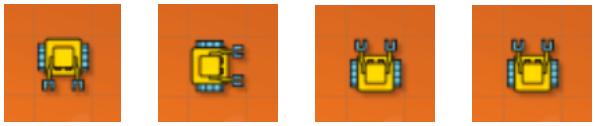
As a conclusion - no matter what direction he faces initially, Karel will always face North after the program finishes. Below is a graphical breakdown of the analysis. Let's begin with the case when Karel initially faces North. The four images in the row show where he is facing initially and then after the first, second, and third cycle of the loop:



Initial direction: North



Initial direction: East



Initial direction: South



Initial direction: West

4.17. Keyword `not`

Karel has a keyword `not` which can be used to reverse the outcome of conditions. For example, `if not wall` is satisfied when `if wall` isn't, and vice versa.

Keyword `not` reverses the outcome of conditions.

The behavior of the keyword `not` can be summarized in the following table:

Keyword not

condition	not condition
Satisfied	Not satisfied
Not satisfied	Satisfied

Let's look again at Program 4.9:

```
1 repeat 3
2   if north
3     pass
4   else
5     left
```

Using the keyword not, it can be simplified to:

PROGRAM 4.10. Shorter version of Program 4.9

```
1 repeat 3
2   if not north
3     left
```

Isn't this neat? But hold on, in Chapter 5 we will show you how this program can be simplified further to only two lines!

4.18. Sensor empty

Karel has a sensor empty to check whether his bag is empty. When in doubt whether the robot is carrying any objects, checking this sensor prior to executing the put command will prevent the program from crashing:

PROGRAM 4.11. Check the bag before removing an object!

```
1 if not empty
2   put
```

The empty sensor is useful in many other situations as well - for example, when the robot needs to remove all objects from his bag:



Karel has an unknown number of nuggets in his bag but no more than 10.

To be clear - the robot cannot know the number of objects in his bag. He does not have a sensor that would count the objects or anything like that. Here is the corresponding code:

PROGRAM 4.12. Remove all objects from the bag!

```
1 repeat 10
2   if not empty
3     put
```

Here is the outcome of the program (there were seven nuggets in the bag):



When the program finishes, seven nuggets lie on the ground.

The `empty` sensor also can be used to have Karel collect only the first of several objects:



Karel only needs to collect the first water bottle.

When Karel's bag is initially empty, the following program will do it:

PROGRAM 4.13. Only collect the first water bottle!

```
1 repeat 10
2   if empty
3     if bottle
4       get
5   go
```

When the program finishes, Karel brings home one water bottle only:



Karel arrives home with one water bottle only.

4.19. Keyword `and`

As you could see, Program 4.13 had two nested conditions: `if bottle` inside of `if empty`. As a result, the command `get` on line 4 was only executed when both these conditions were satisfied. But Karel has a keyword `and` which can do the same in a much simpler way!

> Keyword `and` makes sure that two (or more) conditions are satisfied at the same time.

The single condition `if empty and bottle` is identical to the two nested conditions. So, Program 4.13 can be simplified:

PROGRAM 4.14. Program 4.13 simplified using the keyword `and`

```
1 repeat 10
2   if empty and bottle
3     get
4   go
```

This table summarizes how the keyword `and` works:

Keyword `and`

condition1	**condition2**	**condition1** `and` **condition2**
Satisfied	Satisfied	Satisfied
Satisfied	Not satisfied	Not satisfied
Not satisfied	Satisfied	Not satisfied
Not satisfied	Not satisfied	Not satisfied

4.20. Bounty

Let's show one more example which uses the keyword `and`. This time Karel is in a mine, and he found an unknown number of gold nuggets. Now he needs to put them on mining carts (one nugget per cart). The number and positions of the carts are unknown too:



Karel is in an abandoned gold mine.

If he knew that he has enough nuggets to put one on each cart, then he could use the following program:

```
repeat 10
  if cart
    put
  go
```

But he does not know that, and the program will fail if there aren't enough nuggets in Karel's bag. Namely, at some point the robot will attempt to put a nugget on a cart when his bag is empty. The solution to this problem is very simple though - all he needs to do is also check that his bag is not empty before placing a nugget:

PROGRAM 4.15. Place gold nuggets on the carts!

```
repeat 10
  if cart and (not empty)
    put
  go
```

Notice that the expression `not empty` is enclosed in parentheses.

Using parentheses is recommended in complex conditions as it makes them easier to read.

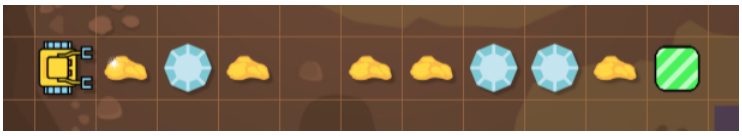
It turns out that Karel had four nuggets in his bag, so when the program finishes, only four carts are loaded:



When the program finishes, four carts are loaded.

4.21. Keyword `or`

Today Karel needs to collect nuggets and gems which are placed randomly between him and his home square.



Karel is collecting nuggets and gems.

This could be done with the following program:

```
1 repeat 10
2   if nugget
3     get
4   if gem
5     get
6   go
```

However, as you can see, there are two conditions with the same body which are checked separately, one after another. In a situation like this, the code can be simplified using the keyword `or`:

PROGRAM 4.16. Collect all nuggets and gems!

```
1 repeat 10
2   if nugget or gem
3     get
4   go
```

Importantly - Karel needs to collect *nuggets and gems*, but typing `if nugget and gem` on line 2 would be a mistake. Namely, Karel would be checking for squares which contain both a nugget and a gem. Such a condition would never be satisfied, so Karel would not collect anything. With the current line 2, using the keyword `or`, the condition will be satisfied when Karel detects any of the two objects - a nugget or a gem.

Keyword `or` makes sure that at least one of two (or more) conditions is satisfied.

The following table summarizes how the keyword `or` works (for two conditions). It can be used to combine three, four, or even more conditions as well.

Keyword `or`

condition1	condition2	condition1 `or` condition2
Satisfied	Satisfied	Satisfied
Satisfied	Not satisfied	Satisfied
Not satisfied	Satisfied	Satisfied
Not satisfied	Not satisfied	Not satisfied

4.22. Danger

This time Karel has a difficult task to find his home square in a wall of fire, water, and acid! The position of his home, as well as the positions of the fire, water, and acid are unknown:



Karel is looking for his home square.

The following program will solve the task:

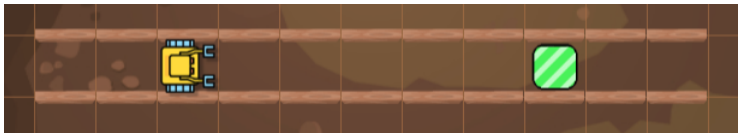
PROGRAM 4.17. Find your home square!

```
repeat 10
  if fire or water or acid
    # Move one square to the right:
    right
    go
    left
  else
    # You are in front of the home square - do nothing:
    pass
# Make one final step home:
go
```

In Chapter 5 we will show you how this program can be simplified using a different type of loop.

4.23. Sensor home

Karel has a sensor home to check whether he is in his home square. For illustration, let's write a simple program for the robot to walk straight ahead until he reaches his home square, and to stay there. He does not know how far ahead the home square is:



Karel is still in the mines.

Karel knows that the width of the maze is 15 steps, so he uses a repeat loop with 15 cycles:

PROGRAM 4.18. Find your home square in the tunnel!

```
1 repeat 15
2   if home
3     pass
4   else
5     go
```

In Chapter 5 we will show you a different type of loop which will make this task easier to solve.

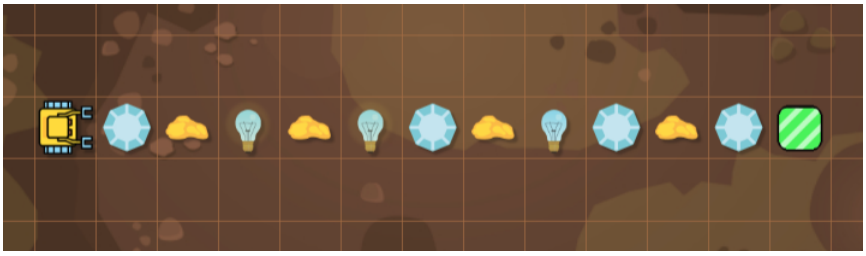
4.24. The full if-elif-else statement

The Karel language provides the full Python-style if-elif-else statement which is useful when one has more than two options to choose from. The keyword elif is an abbreviation of "else if", which makes its meaning self-explanatory. The general form of the if-elif-else statement is

```
if condition_1
  action_1
elif condition_2
  action_2
...
else
  last_action
```

In place of the ellipsis (the three dots) can be one of more additional elif branches.

Today, Karel needs to sort the treasures he collected in the mines. Located randomly between him and his home square lie on the ground light bulbs, gold nuggets, and gems. He needs to collect the bulbs, move the nuggets one square up, and the gems one square down:

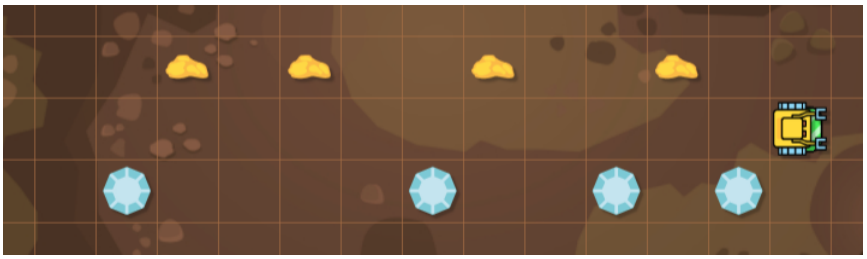


Karel is sorting his treasures.

The solution program looks as follows:

```
repeat 11
  go
  if nugget # move nuggets one square up
    get
    left
    go
    put
    right
    right
    go
    left
  elif gem # move gems one square down
    get
    right
    go
    put
    left
    left
    go
    right
  else # it's a bulb
    get
go
```

When the program finishes, Karel is at his home square, and the treasures are sorted!



Karel's treasures are sorted.

4.25. More is coming!

In this chapter we have only shown you the basic use of Karel's sensors and the `if`, `if-else` and `if-elif-else` statements. There is much more that can be done with them. In the following chapters we will introduce the Boolean values `True` and `False`, as well as Boolean expressions, variables and functions. You will learn that Karel's sensors are in fact Boolean functions. So, keep reading!

4.26. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 4.1. *Why are conditions used in computer programming?*

A *To repeat commands or sequences of commands.*
B *To check user data.*
C *To make runtime decisions.*
D *To correct syntax errors.*

QUESTION 4.2. *Why does Karel need conditions?*

A *To detect and avoid obstacles.*
B *To execute commands.*
C *To find out which way is North.*
D *To check whether his bag is empty.*

QUESTION 4.3. *When is the body of a condition executed?*

A *Always.*
B *When the condition is satisfied.*
C *When the condition is not satisfied.*
D *Never.*

QUESTION 4.4. *When can Karel detect an obstacle?*

A *When it is one step ahead.*

B When it is less than two steps ahead.
C When it is is an adjacent grid square.
D When it is right in front of the robot.

QUESTION 4.5. *When can Karel detect a collectible object?*
A When it is right in front of the robot.
B When it is in an adjacent grid square.
C When it is beneath the robot.
D When it is above the robot.

QUESTION 4.6. *Which of the following are collectible objects?*
A Scorpion
B Snake
C Plant
D Spider

QUESTION 4.7. *Which of the following are obstacles?*
A Fishing net
B Vault
C Fire
D Acid

QUESTION 4.8. *Which of the following are containers?*
A Box
B Wall
C Basket
D Tomb

QUESTION 4.9. *What is the* `else` *branch used for?*
A For code to be executed when the condition is not satisfied.
B For code to be executed after the body of the condition.
C For code to be executed before the body of the condition.
D For code to be never executed.

QUESTION 4.10. *What is the correct way to display* `Hello world!` *in Karel?*
A `print(Hello world!)`
B `print("Hello world!")`
C `print Hello world!`
D `print["Hello world!"]`

QUESTION 4.11. *When is the condition* `if north` *satisfied?*

A When the robot faces North.

B When the robot does not face North.

C When the robot is in the top (North) row of the maze.

D When the robot is in the bottom row of the maze, facing North.

QUESTION 4.12. *What does the command* `pass` *do?*

A It bypasses a condition.

B It bypasses a loop.

C It does nothing.

D It bypasses the `else` *branch.*

QUESTION 4.13. *Karel is facing South. What direction will he face when the following program finishes?*

```
1 if north
2   repeat 3
3     left
```

A North

B South

C West

D East

QUESTION 4.14. *When is the condition* `if empty` *satisfied?*

A When all collectible objects in the maze have been collected.

B When Karel's bag is empty.

C When all containers in the maze are empty.

D When at least one container in the maze is empty.

QUESTION 4.15. *What line of code is equivalent to the following nested conditions?*

```
1 if north
2   if not wall
```

A `if north and (not wall)`

B `if north or (not wall)`

C `if north (and not) wall`

D `if north (or not) wall`

QUESTION 4.16. *Condition1* `and` *Condition2 is satisfied when:*

A Condition1 is satified, Condition2 is satisfied.

B *Condition1 is not satified, Condition2 is satisfied.*

C *Condition1 is satified, Condition2 is not satisfied.*

D *Condition1 is not satified, Condition2 is not satisfied.*

QUESTION 4.17. *Condition1* `or` *Condition2 is satisfied when:*

A *Condition1 is satified, Condition2 is satisfied.*

B *Condition1 is not satified, Condition2 is satisfied.*

C *Condition1 is satified, Condition2 is not satisfied.*

D *Condition1 is not satified, Condition2 is not satisfied.*

QUESTION 4.18. *Karel stands above a nugget (there is no gem beneath him). Which of the following conditions will be satisfied?*

A `if gem and nugget`

B `if gem or nugget`

C `if gem and (not nugget)`

D `if (not gem) or (not nugget)`

QUESTION 4.19. *Karel stands above a beeper and faces a wall. Which of the following conditions will be satisfied?*

A `if beeper and wall`

B `if beeper and (not wall)`

C `if (not beeper) or wall`

D `if beeper or (not wall)`

QUESTION 4.20. *Karel stands above a spider, away from walls, looks West, and his bag is empty. What will the following code print?*

```
1 if north
2   print("I am looking West!")
3 elif wall
4   print("I don't see any walls!")
5 elif not empty
6   print("My bag is empty!")
7 else
8   print("There is a spider!")
```

A *I am looking West!*

B *I don't see any walls!*

C *My bag is empty!*

D *There is a spider!*

5. Conditional Loop

In this chapter you will learn:

- What the conditional (`while`) loop is.
- The conceptual difference between the conditional and counting loops.
- How to pass through unknown mazes using the First Maze Algorithm.

5.1. Conditional loop and the keyword `while`

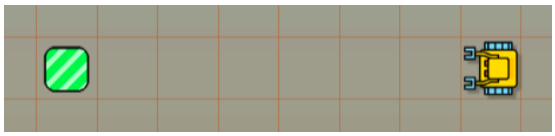
The concept of *conditional loop* is present in all procedural programming languages. Conditional loops allow us to repeat one or more commands without knowing in advance how many repetitions will be needed.

> The conditional loop is used when the number of repetitions (cycles) is **not known in advance**.

In the Karel language as well as in most other procedural programming languages, the conditional loop is defined using the keyword `while`.

5.2. The difference between the conditional and counting loops

Imagine that the robot knows that the home square lies 7 steps ahead:



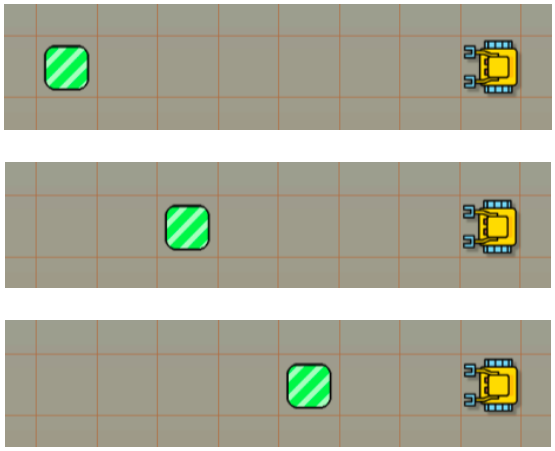
The distance between Karel and his home square is 7 steps.

Then he can use the `repeat` loop to get there:

PROGRAM 5.1. Make 7 steps to your home square!

```
1 repeat 7
2   go
```

However, what if he only knows that the home square lies ahead, but not how far?



Number of repetitions is not known in advance.

Then the conditional loop `while` combined with the condition `not home` is the right way to bring the robot safely home:

PROGRAM 5.2. Walk until you reach the home square!

```
1 while not home
2   go
```

Notice that the keyword `while` is followed by a *condition* (in this case `not home`). The body of the loop (line 2) is indented. It will be repeated while this condition is satisfied.

5.3. Step by step

Let's step through the last program to understand exactly what it does! For simplicity, assume that the initial distance between Karel and his home square is three steps (but the robot does not know that):



Karel's position before the program starts.

Initially, the robot is not at his home square. Therefore the condition `not home` is satisfied, and the body of the loop is executed for the first time. Karel makes one step forward:



Karel's position before the second cycle.

Before the second cycle, the condition is checked again. Since Karel is not home yet, `not home` is satisfied, and the robot makes a second step forward:



Karel's position before the third cycle.

Before the third cycle, the condition is checked again. Since Karel is not at his home square, `not home` is satisfied, and the robot makes a third step forward:



Karel's position before the fourth cycle.

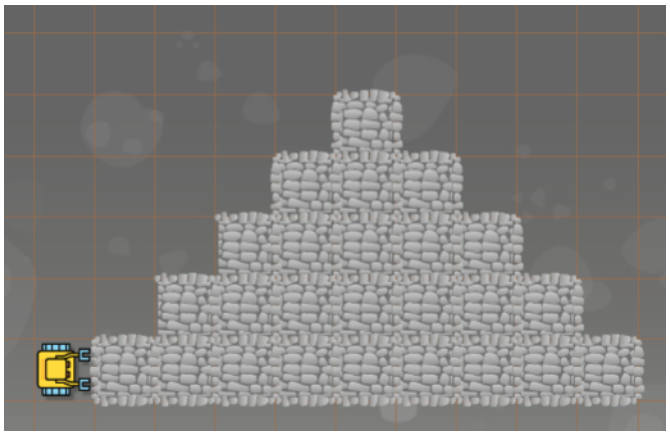
Before the fourth cycle, the `while` loop checks the `not home` condition again. This time, however, Karel is home and therefore the condition is not satisfied. Hence the fourth cycle is not done and the loop ends.

Do you know the number of repetitions? Use the `repeat` loop!
Is the number of repetitions not known? Use the `while` loop!

5.4. Climbing a pyramid

In the previous section, the `while` loop was combined with the condition `not home`. This ensured that the body of the loop was repeated as long as the robot was not home. Naturally, all other conditions that you know from Chapter 4 can be used as well. Let's begin with the condition `wall` ("is there a wall in front of you?").

Karel's next task is to climb a pyramid of unknown height:



Karel is in front of a pyramid of unknown height.

This task is perfect for the `while` loop because Karel just needs to keep climbing, one step at a time, while there is wall in front of him. The repeating pattern to climb one step and get ready in front of the next one is formed by four commands:

```
left
go
right
go
```

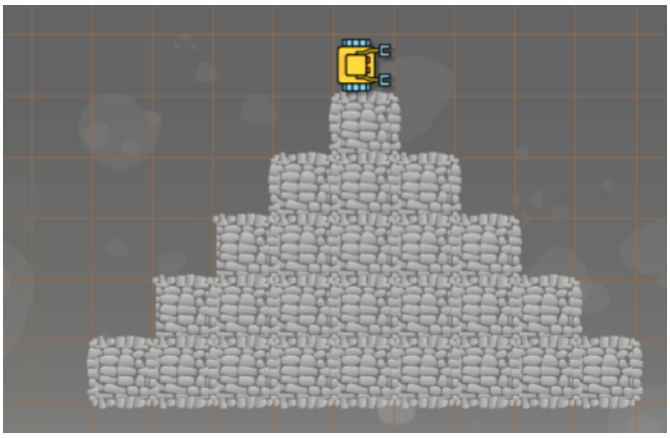
When using the `while` loop, identifying repeating patterns quickly and correctly is equally important as when using the `repeat` loop.

When these four commands are used as the body of a `while` loop, one obtains the final program:

PROGRAM 5.3. Keep climbing until you reach the top!

```
1 while wall
2   left
3   go
4   right
5   go
```

When the robot reaches the top, there is no longer a wall in front of him, therefore the condition `wall` is not satisfied, and the `while` loop ends:



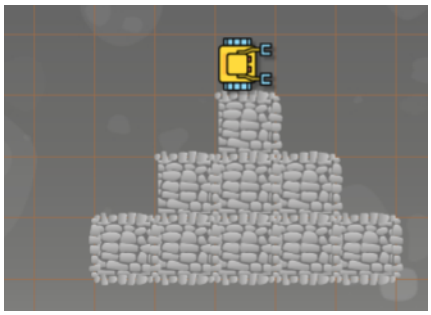
After the program finishes.

To check that our program works for a pyramid of any size, let's run it with a pyramid of height 3:



Testing the program on a pyramid of height 3.

And indeed, the robot stops on the top of the pyramid!



The program still works correctly.

5.5. Narrow escape

Let's have a look at a `while` loop combined with the condition `fire` ("is there a fire in front of you?"). The only way for Karel to get to his home square is through a narrow gap between the flames. The length of the fire wall and the position of the gap are unknown.



Karel needs to find the gap in the wall of fire.

Here, the repeating pattern is to move one square to the right. It is formed by three commands:

```
right
go
left
```

When using them as the body of a `while` loop, and adding two more `go` commands after the loop finishes, one obtains the final solution program:

PROGRAM 5.4. Find the gap and enter the home square!

```
1 while wall
2   right
3   go
4   left
5 repeat 2
6   go
```

Notice that indentation is cancelled for commands which no longer belong to the body of the `while` loop.

When the robot finds the gap in the flames, the condition `fire` is no longer satisfied, and the `while` loop ends. Then the last two lines make sure that Karel reaches the home square:



Karel needs to find the escape and enter his home square.

To check that the program works with an arbitrary length of the fire wall and an arbitrary position of the gap, let's run it with one additional maze:



Same task with different parameters.

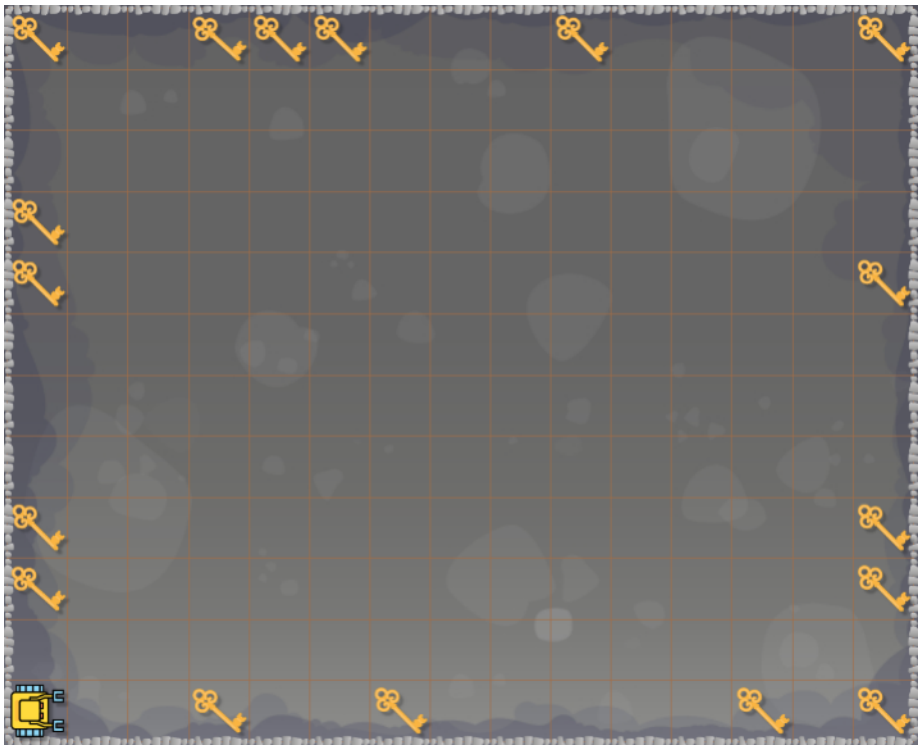
When the program finishes, the robot is home again:



The program finished successfully.

5.6. Combining the `repeat` and `while` loops

Smart combinations of the `repeat` and `while` loops can make your programs more elegant and efficient. For example, imagine that Karel's task is to run along the perimeter of the maze and collect all keys. The maze has width 15 squares and height 12 squares. These numbers are known, therefore it seems more natural to use the `repeat` loop.



Karel is collecting keys.

PROGRAM 5.5. Collect all keys!

```
1  repeat 2
2    repeat 14
3      go
4      if key
5        get
6    left
7    repeat 11
8      go
9      if key
10       get
11   left
```

This program is perfectly correct, but as you can see, a large part of the code is there twice, just because one time we need it inside a `repeat 14` loop and another time inside a `repeat 11` loop. This can be fixed by using the `while` loop which automatically takes care of the different lengths of the maze walls:

PROGRAM 5.6. Last program, improved using a `while` loop

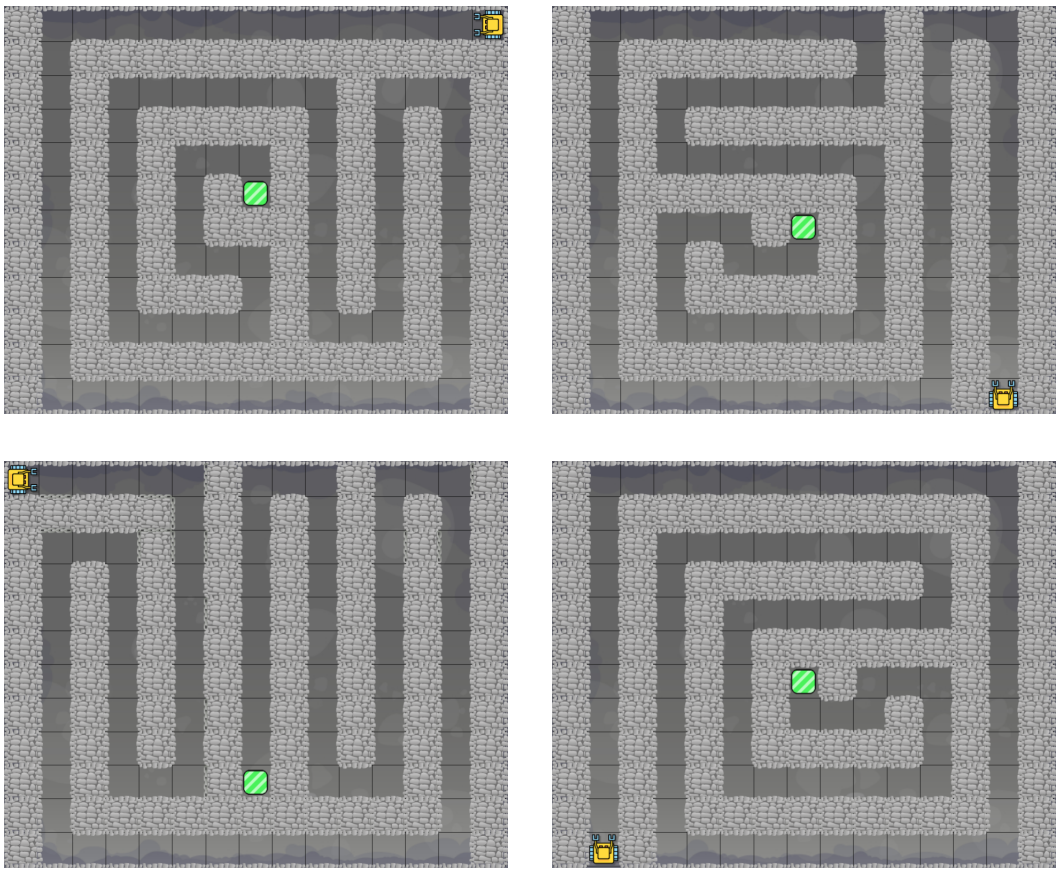
```
repeat 4
  while not wall
    go
    if key
      get
  left
```

This program has the same outcome as the previous one, but it is more elegant because it does not contain the same code twice.

> Smart combinations of the `repeat` and `while` loops
> can make your programs more elegant and efficient.

5.7. First Maze Algorithm (FMA)

Today let's teach Karel how to pass through an arbitrary maze:



This can be done using the First Maze Algorithm of robotics:

First Maze Algorithm

1: While you are not home, do the following:
2: If there is a wall in front of you, turn left.
3: If there is still a wall, turn around.
4: Move one step forward.

The algorithm allows a robot to pass through an arbitrary maze where

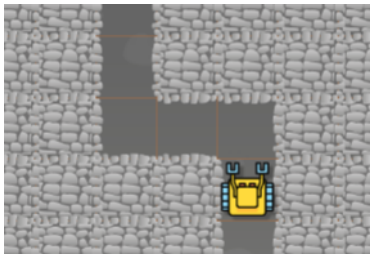
- The path is one square wide.
- At any time it goes either forward, to the left, or to the right (no forks, no loops, no dead ends).

It can be translated to the following code:

PROGRAM 5.7. First Maze Algorithm translated to Karel code

```
1 while not home
2   if wall
3     left
4     if wall
5       right
6       right
7   go
```

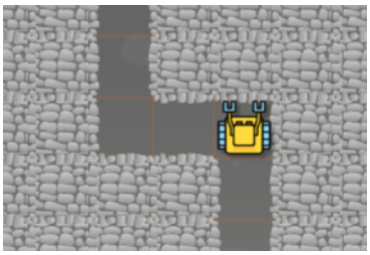
Let's see what the program does! First, what happens when Karel can just go straight?



There is no wall in front of Karel.

In this case there is no wall in front of the robot, so the condition on line 2 is not satisfied. Therefore he skips lines 2 - 6, and just makes one step forward on line 7.

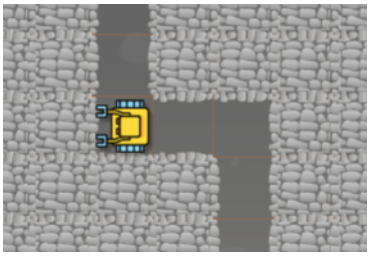
Second, what happens when the path goes to the left?



Path goes to the left.

In this case there is a wall in front of the robot. Therefore the condition on line 2 is satisfied. So, he makes a left turn. Then he checks for wall again on line 4, but there is no wall in front of him. Therefore he skips lines 5 and 6, and goes to line 7. In summary, he turns left and makes one step forward.

Third, what happens when the path goes to the right?



Path goes to the right.

Now there is a wall in front of the robot, so he condition on line 2 is satisfied and Karel turns left. On line 4 he checks for a wall again. Since there is a wall, he executes lines 5 and 6, and makes a complete 180-degree turn. Then he makes one step forward on line 7. In summary, Karel turns right and makes one step forward.

Keep in mind that the path needs to be one square wide, and
the maze should not contain forks, loops or dead ends.

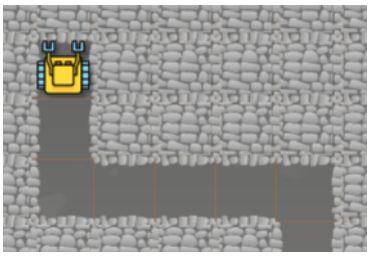
5.8. Three possible failures of the FMA

In the previous section you have learned that the FMA assumes that

- The path is one square wide.
- At any time it goes either forward, to the left, or to the right (no forks, no loops, no dead ends).

But what if some of these assumptions are not satisfied?

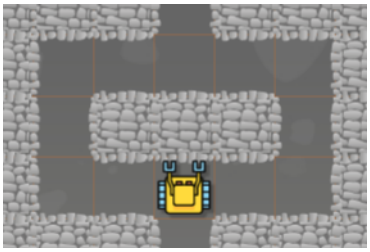
First, let's see what happens when Karel gets into a dead end:



Karel reached a dead end.

Since there is a wall in front of him, the condition on line 2 is satisfied, and he turns left. Then there is a wall in front of him again, therefore the condition on line 4 is satisfied, and he makes a 180-degree turn. But he faces a wall for a third time, and so the `go` command on line 7 will crash the program.

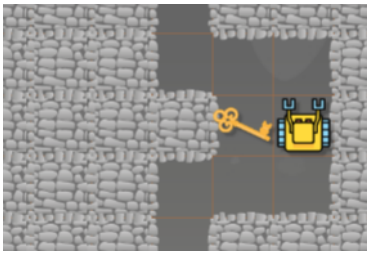
Next, why are loops in the path prohibited? Let's see:



The path forms a loop.

In this case Karel first turns left, and then he starts circling through the loop infinitely. He never gets out of this maze!

Last, the FMA requires that the path be one square wide. To explain why, let's look at a maze where the path is wider:



The path is more than one square wide.

In this case, the robot fails to collect the key because he does not visit all squares in the maze.

5.9. Right-handed version of the FMA

With Program 5.7, when the robot arrives at a T-intersection, he first turns left and chooses the left path. Therefore the algorithm is sometimes called "left-handed". Alternatively, we can first make him turn right, then check for a wall, and if there is a wall, turn around. This will be the "right-handed" version:

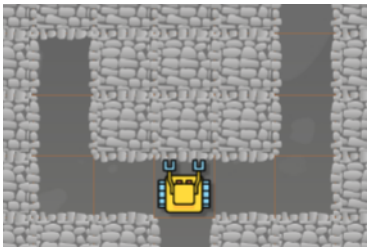
PROGRAM 5.8. Right-handed version of the FMA

```
1 while not home
2   if wall
3     right
4     if wall
5       left
6       left
7   go
```

The two versions are fully equivalent when the assumptions on the maze are satisfied:

- The path is one square wide.
- At any time it goes either forward, to the left, or to the right (no forks, no loops, no dead ends).

But when some of these assumptions are not satisfied, then the two versions may yield different results. For example, consider a maze with a dead end in it:



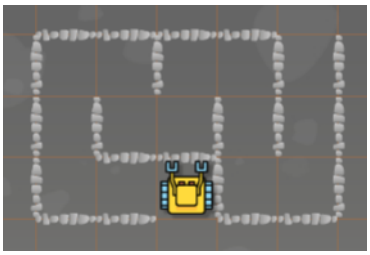
The path contains a dead end.

Here the right-handed version of FMA succeeds while the left-handed version fails because the robot gets into the dead end. Of course, if the dead end was on the right, then the outcome would be reversed. So, it would be wrong to say that one of the versions is better than the other.

It is important to know under which conditions the FMA will work correctly.

5.10. Other types of mazes the FMA can handle

Based on the images of mazes shown so far, you might have the impression that all the walls must be at least one square wide. This is not the case! The FMA can also handle mazes with thin walls such as the following one:



A maze with thin walls.

Here is the FMA again, step through it in your mind to make sure it will work!

PROGRAM 5.9. First Maze Algorithm

```
1 while not home
2   if wall
3     left
4     if wall
5       right
6       right
7   go
```

5.11. Improving Program 4.9 "Finding North"

As promised in Chapter 4, let's revisit some programs we wrote there, and show that they can be improved using the `while` loop. We can start with Program 4.9 from page 65 which turns Karel to face North no matter which direction he is facing initially:

```
1 repeat 3
2   if north
3     pass
4   else
5     left
```

Using the `while` loop, the same can be achieved using just two lines:

```
1 while not north
2   left
```

5.12. Improving Program 4.12 "Emptying bag"

Program 4.12 from page 68 makes Karel remove all objects from his bag and place them on the ground. Importantly, because it uses the `repeat` loop, it needs to make an unnatural assumption that Karel has at most 10 objects in his bag:

```
repeat 10
  if not empty
    put
```

Using the `while` loop, one can forget the limiting assumption, and just write

```
while not empty
  put
```

5.13. Improving Program 4.17 "Danger"

Program 4.17 from page 72 helps Karel locate his home square within a wall of fire, water and acid. Again, because it uses the `repeat` loop, it needs to make an assumption on the maximum length of the wall:

```
repeat 10
  if fire or water or acid
    # Move one square to the right:
    right
    go
    left
  else
    # You are in front of the home square - do nothing:
    pass
# Make one final step home:
go
```

With the `while` loop, the wall can be arbitrarily long, and the code becomes much simpler:

```
while fire or water or acid
  # Move one square to the right:
  right
  go
  left
# Make one final step home:
go
```

5.14. Improving Program 4.18 "Tunnel"

As a last example, Program 4.18 from page 73 helps Karel locate his home square in a tunnel. It as well uses a limiting assumption on the number of steps Karel can take:

```
1 repeat 15
2   if home
3     pass
4   else
5     go
```

Once more, the version based on the `while` loop is much simpler and more elegant:

```
1 while not home
2   go
```

5.15. Other types of loops: `do-while`, `until`, and `for`

Some languages provide the `do-while` or `until` loops. These are basically the same. The (big) difference vs. the `while` loop is that they execute the body *at least once* before the condition is checked at the end. However,

> Python provides neither the `do-while` nor the `until` loop,
> therefore we have not implemented them in Karel.

On the other hand, Python provides a `for` loop which is more powerful than the same type of loop in most other programming languages. Karel has the Python `for` loop, and we will get to it soon!

5.16. More is coming!

In this chapter we have only shown you the basic use of the `while` loop. There is much more to it. In the following chapters we will introduce the Boolean values `True` and `False`, as well as Boolean expressions, variables and functions. All these will expand the possibilities of what can be done with the `while` loop. Therefore, keep reading!

5.17. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 5.1. *The* `while` *loop should be used when*

A The number of repetitions is known in advance.

B The number of repetitions is not known in advance.

C The `repeat` *loop already is used elsewhere.*

D The `repeat` *loop is broken.*

QUESTION 5.2. *Which of the following codes are valid?*

A
```
while repeat not home
```

B
```
repeat while not home
```

C
```
repeat if not home
```

D
```
while not home
```

QUESTION 5.3. *At what time is the condition checked in the* `while` *loop?*

A Before every cycle.

B After every cycle.

C In the middle of every cycle.

D When the loop has finished.

QUESTION 5.4. *Does the body of the* `while` *loop need to always be indented?*

A No.

B Yes.

C Not when it contains multiple commands.

D Not when it contains a single command.

QUESTION 5.5. *What forms of the* `while` *loop are valid?*

A
```
while wall
```

B
```
while not wall
```

C
```
while not empty
```

D
```
while empty
```

QUESTION 5.6. *When using the* `while` *loop, does one need to identify repeating patterns?*

A Yes.

B No, this is only needed when using the `repeat` *loop.*

C Yes but only when the condition does not contain `not`*.*

D Yes but only when the condition contains `not`*.*

QUESTION 5.7. *What is the First Maze Algorithm (FMA) designed for?*

A To help Karel find a treasure in the maze.

B To help Karel pass through unknown mazes.

C To help Karel place collectible objects in mazes.

D To help Karel place obstacles in mazes.

QUESTION 5.8. *What are the limiting assumptions on the maze for the FMA to work correctly?*

A The path should be one square wide.

B The path should not contain forks.

C The path should not contain loops.

D The path should not contain dead ends.

QUESTION 5.9. *How can the FMA fail when the maze contains a dead end?*

A Karel can return to where he started.

B Karel can run in an infinite loop.

C Karel can crash.

D Karel can stall without moving.

QUESTION 5.10. *How can the FMA fail when the maze contains forks?*

A Karel can stall without moving.

B Karel can crash.

C Karel can return to where he started.

D Karel can run in an infinite loop.

QUESTION 5.11. *How can the FMA fail when the path is more than one square wide?*

A Karel can return to where he started.

B Karel may miss some squares.

C Karel can crash.

D Karel can stall without moving.

QUESTION 5.12. *What types of loops does Karel provide?*

A `repeat` *loop*

B `until` *loop*

C `while` *loop*

D `do-while` *loop*

6. Custom Commands

In this chapter you will learn:

- How to split complex problems into simpler ones which are easier to solve.
- How to define custom commands using the keyword `def`.
- How to use the optional `return` statement.
- The Second Maze Algorithm of robotics.
- That the shortest program may not always be the most efficient one.
- How to define new commands locally within the body of other commands.

6.1. Why are custom commands useful?

Today, Karel needs to collect 25 computer chips which form five stars:



Karel needs to collect 5 stars of chips.

The following program will solve the task, but it would not get you hired as a software engineer. Look at it first, and then we will explain what is wrong with it:

PROGRAM 6.1. Collect five stars of chips!

```
repeat 4
  go
left
go

repeat 4
  get
  go
  right
  go
  left
  left
go
get
go

right
repeat 6
  go
left

repeat 4
  get
  go
  right
  go
  left
  left
go
get
go

left
repeat 8
  go
right

repeat 4
```

```
39 ||   get
40 ||   go
41 ||   right
42 ||   go
43 ||   left
44 ||   left
45 || go
46 || get
47 || go
48 ||
49 || right
50 || repeat 10
51 ||   go
52 || left
53 ||
54 || repeat 4
55 ||   get
56 ||   go
57 ||   right
58 ||   go
59 ||   left
60 ||   left
61 || go
62 || get
63 || go
64 ||
65 || left
66 || repeat 6
67 ||   go
68 || right
69 ||
70 || repeat 4
71 ||   get
72 ||   go
73 ||   right
74 ||   go
75 ||   left
76 ||   left
77 || go
```

```
78 get
79 go
80
81 right
82 repeat 8
83   go
```

The problem with this program is that it contains a 10-line section which was copied and pasted five times:

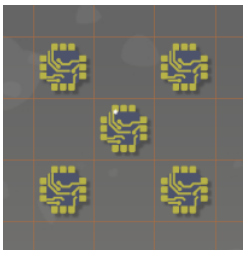
PROGRAM 6.2. Repeated code section

```
1  repeat 4
2    get
3    go
4    right
5    go
6    left
7    left
8  go
9  get
10 go
```

This is the section of the code which actually collects the pattern of five chips.

> The same code should **never** be copied and pasted at different places of your program.

Copying and pasting code makes the program not only long and cumbersome to work with, but also extremely prone to mistakes. Imagine that the pattern of the chips changes:



Another pattern.

Then the above code would have to be redone at five different places (!) Since programmers are only human, they are likely to make mistakes. And changing the same code at five different places makes the likelihood of making a mistake extremely high.

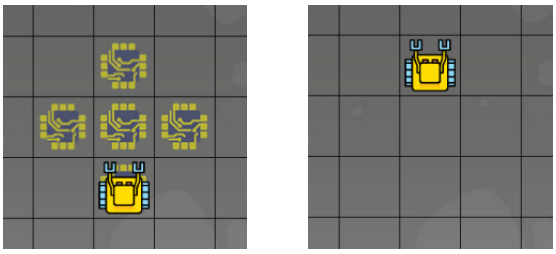
6.2. Defining a custom command `star`

The correct approach is to define a new command for the repeated functionality. Let's call it `star`:

PROGRAM 6.3. Custom command `star`

```
1  # Collect five chips:
2  def star
3    repeat 4
4      get
5      go
6      right
7      go
8      left
9      left
10   go
11   get
12   go
```

Notice that custom commands are defined using the keyword `def` which is followed by the name of the new command. Then, indented, comes the body of the command. The following image shows where Karel should be when the command `star` is executed, and then the robot's position after the command finishes:

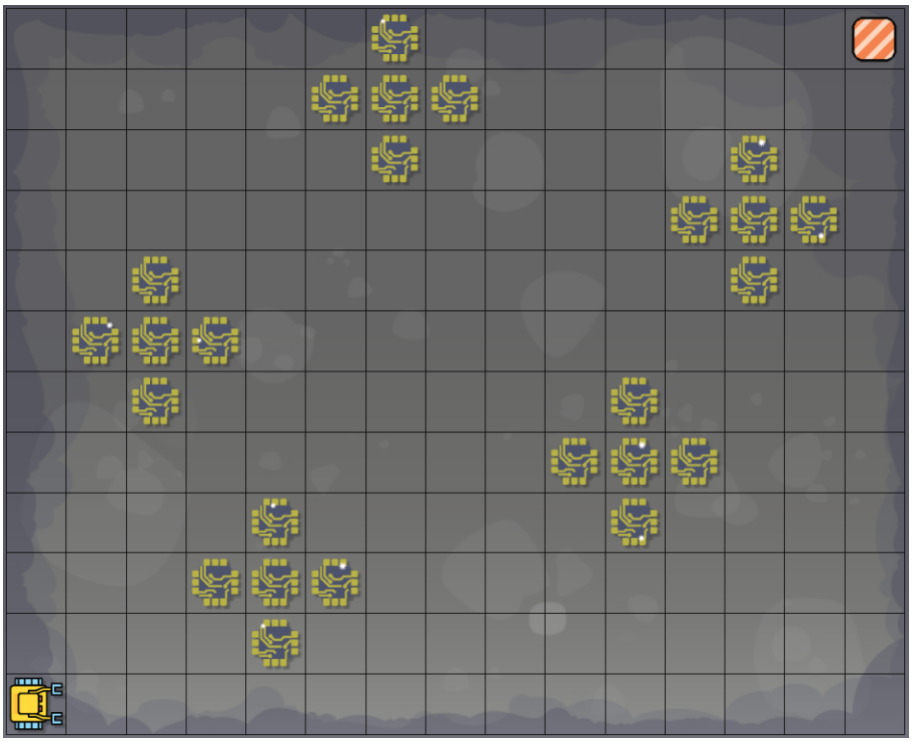


Karel's position before and after the command `star` is executed.

A custom command solves a *subtask* - a simpler problem whose solution is needed in order to solve the main task.

6.3. Assembling the solution of the main task

Recall that the main task is to collect all 25 chips:



Main task.

But how can the custom command `star` be used to solve it, when Karel is far away from the first star of chips?

The answer is - before executing the command `star`, we will program Karel to get there! In order to stand on the first chip of the first star, the robot needs to make four steps forward, turn left, and make one more step:

```
repeat 4
  go
left
go
```

After executing this initial code, Karel is ready to execute the command `star` for the first time.

Notice that one needs to identify repeating patterns quickly and correctly in order to recognize subtasks and define suitable custom commands.



Karel is ready to execute the new command `star`.



Karel has executed the command `star`.

Next, in order to get ready for collecting the second star of chips, Karel needs to turn right, make 6 steps forward, and turn left:

```
right
repeat 6
  go
left
```

After executing this code, he is ready:



Karel is ready to execute the command `star` again.

Now you should able to figure out the rest! Write down the commands Karel needs to get ready to collect the third, fourth, and fifth stars of chips. The final program is as follows:

PROGRAM 6.4. The final program

```
# Collect five chips:
def star
  repeat 4
    get
    go
    right
```

```
    go
    left
    left
  go
  get
  go

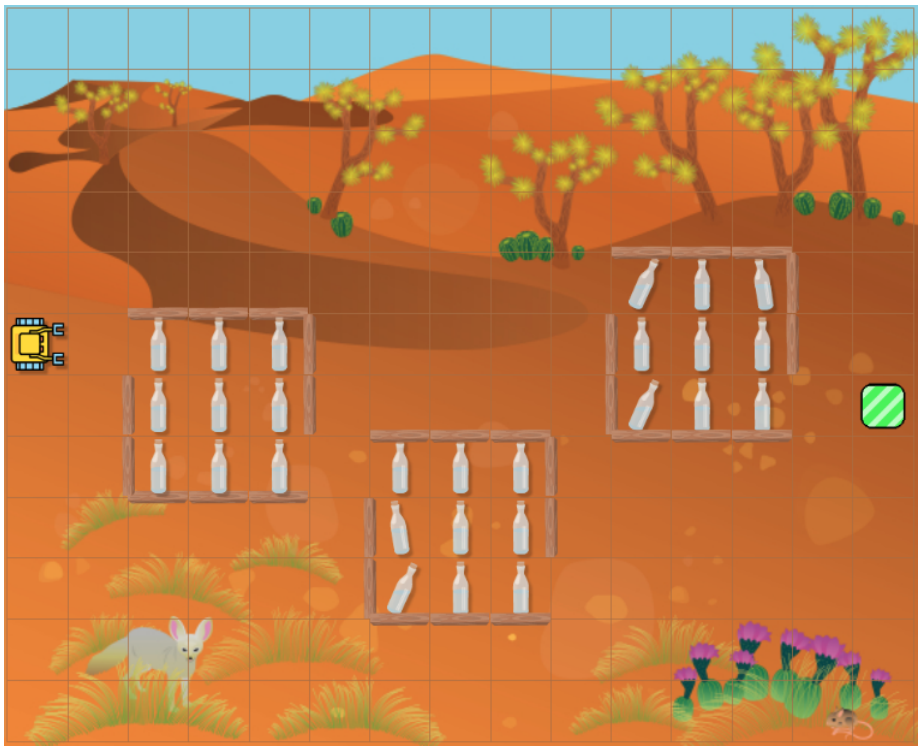
# Main program:
# Collect the 1st star:
repeat 4
  go
left
go
star
# Collect the 2nd star:
right
repeat 6
  go
left
star
# Collect the 3rd star:
left
repeat 8
  go
right
star
# Collect the 4th star:
right
repeat 10
  go
left
star
# Collect the 5th star:
left
repeat 6
  go
right
star
# Finish at the home square:
```

```
46 right
47 repeat 8
48   go
```

Decomposing complex tasks into simpler ones which are easier to solve is one of the most important skills in computer programming.

6.4. Collecting water bottles

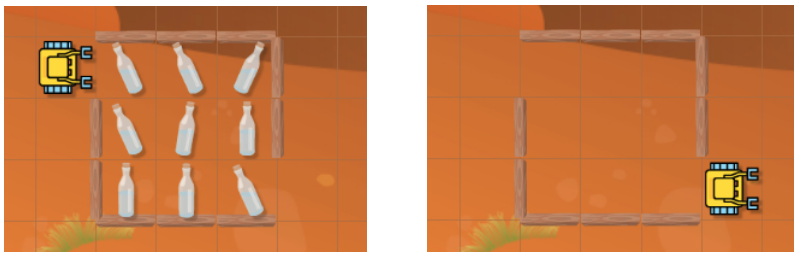
Today Karel faces another complex task - to collect 27 water bottles which are stored in three crates:



Karel needs to collect 27 water bottles from three crates.

As you already know, it would not be a good idea to go ahead and start writing code right away. Let's think about it first, and figure out a suitable subtask which is easier to solve than the main task.

Here is a good candidate for such a subtask. Let's call the new command `get 9`:



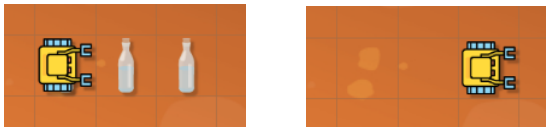
Karel's position before and after executing the command `get9`.

This time, let's work like an experienced software engineer, and solve the main task first. We know that the subtask is a routine thing to do, so we can leave it for later. The main program is below. Step through it in your mind to verify that it is correct!

PROGRAM 6.5. Main program

```
# Main program:
# Get to the 1st crate:
go
# Collect the 1st crate:
get9
# Collect the 2nd crate:
get9
# Get to the 3rd crate:
left
repeat 5
  go
right
# Collect the 3rd crate:
get9
# Make one last step home:
go
```

This was a piece of cake - which tells us that the subtask was chosen really well! It remains to write the body of the new command `get9`. But you know what? We can still simplify it by splitting it into three smaller subtasks, where one subtask will be to just collect a row of three water bottles. Let's call the new command `get3`:



Karel's position before and after executing the command `get3` (on the left, one water bottle is beneath Karel).

The following code will do it:

PROGRAM 6.6. Custom command `get3`

```
# Collect 3 water bottles:
def get3
  repeat 2
    get
    go
  get
```

And now, the new command `get9` is easy to write:

PROGRAM 6.7. Custom command `get9`

```
# Collect 9 water bottles:
def get9
  get3
  right
  go
  right
  get3
  left
  go
  left
  get3
  go
```

Finally, let's put everything together and write the solution program for the main task. The program will consist of two custom commands and a main program:

PROGRAM 6.8. Final solution program

```
# Collect 3 water bottles:
def get3
  repeat 2
    get
    go
  get

# Collect 9 water bottles:
def get9
  get3
```

```
  right
  go
  right
  get3
  left
  go
  left
  get3
  go

# Main program:
# Get to the 1st crate:
go
# Collect the 1st crate:
get9
# Collect the 2nd crate:
get9
# Get to the 3rd crate:
left
repeat 5
  go
right
# Collect the 3rd crate:
get9
# Make one last step home:
go
```

6.5. The optional `return` statement

Every custom command can (but does not have to) contain one or more `return` statements. The `return` statement terminates and exits the custom command. For example, the command `get3` from the previous section could be written as follows:

PROGRAM 6.9. Custom command `get3` (with a `return` statement at the end)

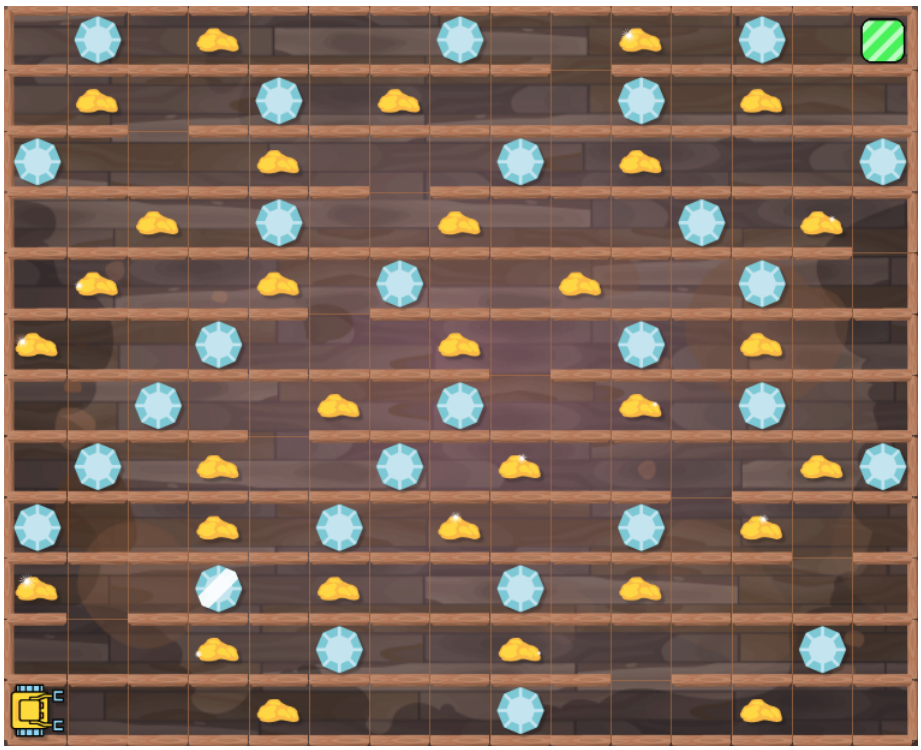
```
# Collect 3 water bottles:
def get3
  repeat 2
    get
    go
```

```
6    get
7    return
```

Although a custom command may contain multiple `return` statements, using just one at the end is considered to be a cleaner programming style.

6.6. Arcade game

Karel's next task is to pass through the entire maze, collecting all gold nuggets and gems:



Karel needs to collect all gold nuggets and gems.

This is a complex task, so why don't we solve a simpler subtask first. It will be much easier to just pass through one floor, left to right, collect all nuggets and gems, and if not at home, then find the opening to the next floor up. Let's call the new command `floor`:



Before executing the new command `floor`.



After the command `floor` finishes.

As before, first let's solve the main task assuming that we already have the command `floor`. The new command will be easy to define later:

PROGRAM 6.10. Main program

```
# Main program:
repeat 12
  # Pass through one floor:
  floor
  # If not at home, get ready at the left end of the new floor:
  if not home
    left
    while not wall
      go
    repeat 2
      right
```

And here is the new command `floor`:

PROGRAM 6.11. New command `floor`

```
# Pass through one floor and collect all nuggets and gems:
def floor
  repeat 14
    if gem or nugget
      get
    go
  if gem or nugget
    get
  # If not home, find the opening up:
  if not home
    left
    while wall
      left
      go
```

```
15        right
16      go
```

By putting the pieces together, we obtain the final solution program:

PROGRAM 6.12. Final solution program

```
# Pass through one floor and collect all nuggets and gems:
def floor
  repeat 14
    if gem or nugget
      get
    go
  if gem or nugget
    get
  # If not home, find the opening up:
  if not home
    left
    while wall
      left
      go
      right
    go

# Main program:
repeat 12
  # Pass through one floor:
  floor
  # If not at home, get ready at the left end of the new floor:
  if not home
    left
    while not wall
      go
    repeat 2
      right
```

A custom command must be defined, and then also called.
Defining a custom command without calling it does nothing.

6.7. Second Maze Algorithm (SMA)

You already know the First Maze Algorithm of robotics which allows Karel (or any other robot) to pass through unknown mazes. The Second Maze Algorithm (SMA) allows robots to follow an arbitrary winding wall or a line on the ground. Imagine that Karel is in the jungle, and he has to follow a winding line of tree branches to his home square. He stands next to the line which is on his left-hand side:



Karel is in the jungle.

The Second Maze Algorithm of robotics will guide Karel safely to his home square:

Second Maze Algorithm

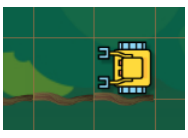
1: While you are not home, do the following:
2: Turn left.
3: While there is a wall in front of you, keep turning right.
4: Make one step forward.

It can be translated to the following code:

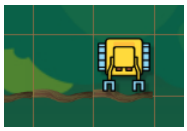
PROGRAM 6.13. Second Maze Algorithm translated to Karel code

```
1 while not home
2   left
3   while wall
4     right
5   go
```

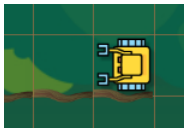
Let's have a closer look at how the algorithm works! At the beginning, the line goes straight forward:



On line 2, Karel will turn left:



The `while` loop on line 3 will do one cycle, and so Karel will turn right once:



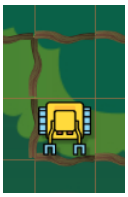
Finally, on line 5, he will make one step forward:



OK, that worked! After two more steps forward, Karel will arrive at a right turn:



On line 2, he will turn left:



Then, the `while` loop on line 3 will do two cycles:

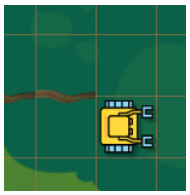




And finally, Karel will make one step forward on line 5:



After this, the robot will make another right turn and two steps forward. Then the situation becomes more interesting because there is no wall on his left. BTW, this case is why the algorithm begins with a left turn, in case you were puzzled about it:



So, on line 2 Karel will turn left:



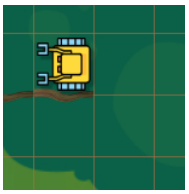
The body of the `while` loop will be skipped because there is no wall in front of the robot, and he will just make one step forward on line 5:



Now the situation repeats itself because there is no wall on Karel's left. Hence, on line 2 he turns left:



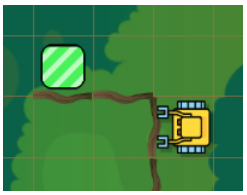
Because there is no wall in front of him, the body of the `while` loop will be skipped again, and on line 5 he will make one step forward:



Woo-hoo, almost there! After a few more steps and one more right turn, Karel arrives at a left turn:



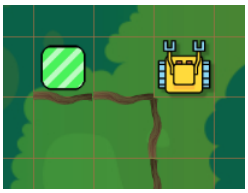
On line 2 he turns left:



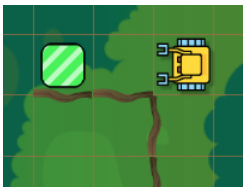
Since there is a wall in front of him, the `while` loop will make him turn right once:



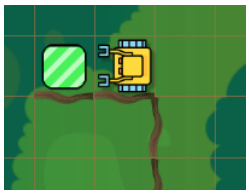
And on line 5 he makes one step forward:



You already know this situation - Karel does not have a wall on his left. On line 2 he turns left:



Then, because there is no wall in front of the robot, the body of the `while` loop is skipped, and Karel just makes one step forward:



6.8. Right-handed version of the SMA

Like the First Maze Algorithm, also the Second Maze Algorithm has a right-handed version. It is useful when Karel needs to follow a wall or line which is on his right:



Now the wall is on Karel's right.

The algorithm is symmetric to the SMA (just switch "left" to "right" and vice versa):

Second Maze Algorithm (right-handed version)

```
1: While you are not home, do the following:
2:     Turn right.
3:     While there is a wall in front of you, keep turning left.
4:     Make one step forward.
```

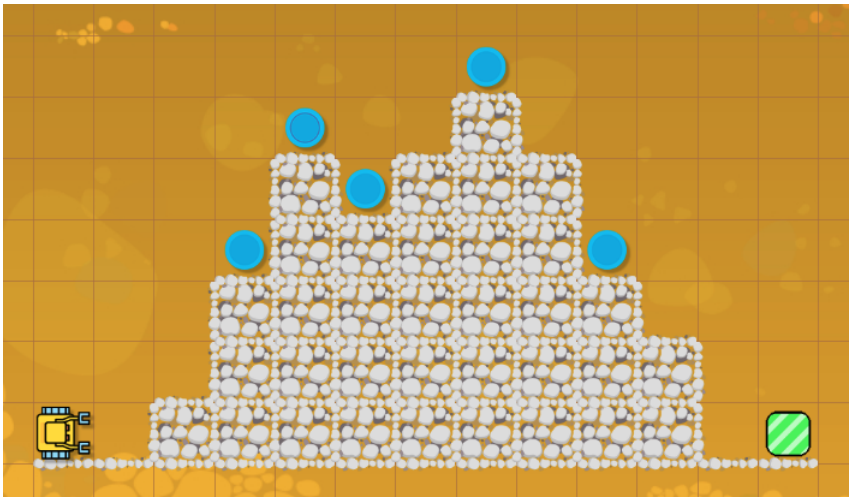
Here is the corresponding code:

PROGRAM 6.14. SMA (right-handed version)

```
1 while not home
2   right
3   while wall
4     left
5   go
```

6.9. Rock climbing

Today, Karel will climb some rocks! His task is to collect all the beepers and enter his home square. The shape of the rock is random, and so is the number and positions of the beepers on it:



Karel stands in front of a desert rock.

This task is perfect for the Second Maze Algorithm (right-handed version). Just executing Program 6.14 will make Karel follow the contour of the rock ("climb on the rock") and get to his home square. So, the only change we need to make, is to check for a beeper after each step the robot makes:

PROGRAM 6.15. Collect all beepers!

```
1 while not home
2   right
3   while wall
4     left
5   go
6   if beeper
7     get
```

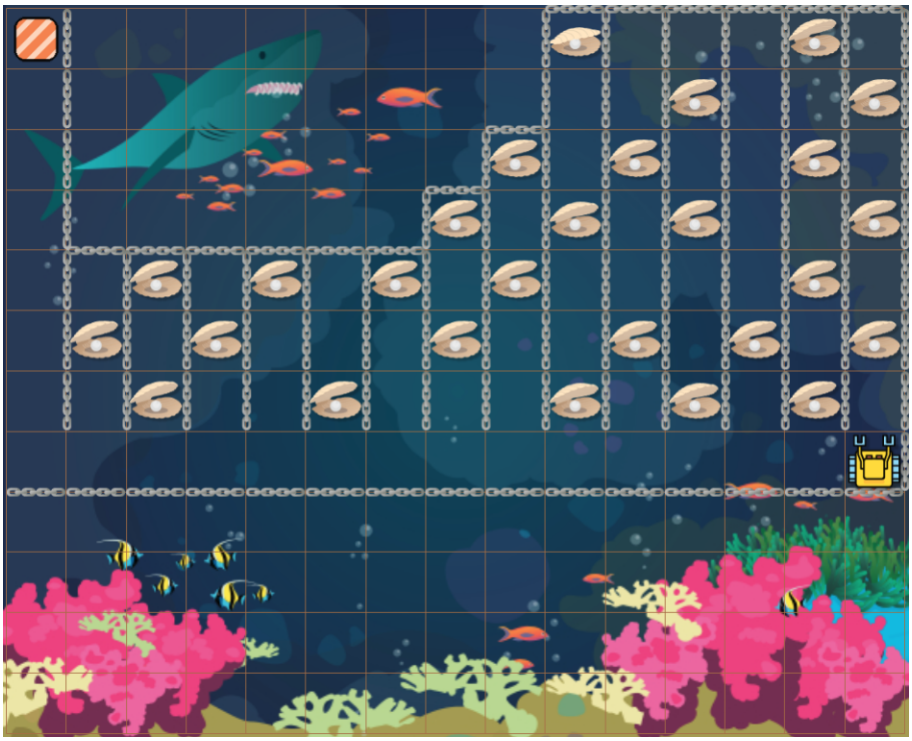
6.10. Program length vs. efficiency

You already know that a well thought-out program usually is shorter and more elegant than a program which was written hastily. However, program length is not the only thing that matters. Even more important is *efficiency*.

> A program is efficient when it uses as few operations as possible to solve the given task.

In this section we will show you that the shortest program may not always be the most efficient one.

Today, Karel is collecting pearls in the ocean:



Karel is diving and collecting pearls.

This task is perfect for the Second Maze Algorithm (right-handed version) - Karel just needs to follow the wall on his right-hand side, and it will lead him to his home square. The solution program is practically the same as Program 6.15, the only difference being that the robot is looking for pearls instead of beepers:

```
1 while not home
2   right
```

```
3     while wall
4       left
5     go
6     if pearl
7       get
```

This program needs 503 operations to collect all pearls and get Karel home. Save this number for a future comparison!

Next, let's solve the task differently. We will write a new command `column` for the robot to collect all pearls in the column above him, and get ready below the next column. Then, the main program will just loop over all columns:

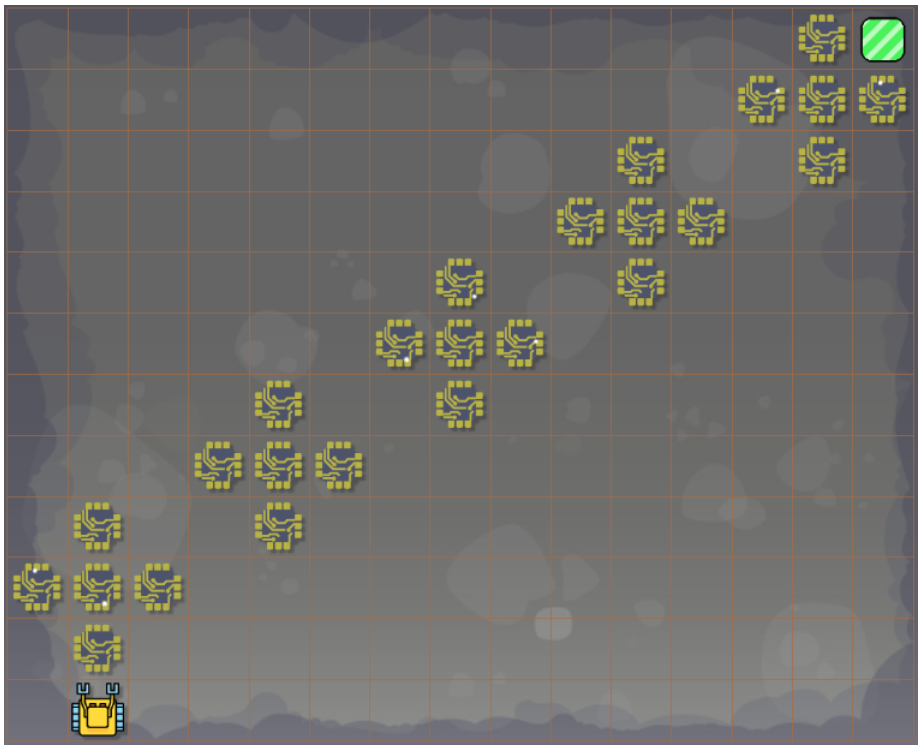
```
1  # Collect one column of pearls:
2  def column
3    while not wall
4      go
5      if pearl
6        get
7    left
8    left
9    while not wall
10     go
11   right
12   go
13   right
14
15 # Main program:
16 repeat 14
17   column
18 repeat 7
19   go
```

As you can see, this program is longer than the previous one - it has 16 lines of code as opposed to 7. However, it provides a solution which is much more efficient. It collectc all pearls and gets the robot home with only 242 operations!

The shortest program may not always be the most efficient one.

6.11. Writing monolithic code vs. using custom commands

Sometimes it is not obvious whether one will gain anything by introducing a custom command. For illustration, let's return to the five stars of chips from Section 6.1 (page 97), but now the stars will form a regular repeating pattern. The task is the same - collect all the chips!



Five stars of chips forming a repeating pattern.

The code can be written as one monolithic piece, without defining any custom commands:

PROGRAM 6.16. Collect all stars! (monolithic code)

```
1  go
2  repeat 5 # loop over stars
3    repeat 4 # collect one star
4      get
5      go
6      right
7      go
8      left
9      left
10   go
```

```
11    get
12    go
13    right # move to the next star or home square
14    repeat 3
15      if not home
16        go
17    left
```

This code is very dense, but it does not contradict any good programming practices - there is no part which would be copied and pasted at multiple different places in the program (as it was in the original task in Section 6.1). The comments make it readable, so it is a good code.

Now let's solve the same task using the custom command `star` from Section 6.1. And moreover, we will define another new command `next` for the robot to move to the next star or to the home square:

PROGRAM 6.17. Collect all stars! (with custom commands)

```
1  # Collect one star of chips:
2  def star
3    repeat 4
4      get
5      go
6      right
7      go
8      left
9      left
10   go
11   get
12   go
13
14 # Move to the next star or home square:
15 def next
16   right
17   repeat 3
18     if not home
19       go
20   left
21
22 # Main program:
```

```
23 go
24 repeat 5
25   star
26   next
```

The latter program had 21 lines (not counting empty lines and comments) while the former only had 17. This is not a huge difference. On the other hand, the latter program is so well readable that the main program does not need any comments - it is completely self-explanatory. Readability matters. If we had to pick up our favorite, then the slightly longer but much more readable program would definitely be our choice.

> Readability matters - when in doubt, define a custom command rather than writing dense, monolithic code.

6.12. Defining commands inside other commands

It is possible in Karel to define commands within the body of other commands. This feature was implemented for conceptual compatibility with Python. Its usefulness becomes clear in large, complex programs where it helps keep code better organized.

For illustration, let's return to Section 6.4 (page 106) where Karel was collecting water bottles from three crates. The solution Program 6.8 defined new commands `get3` and `get9`. Notably, the command `get3` was only used within the body of the command `get9` and nowhere else. Therefore, it can be defined locally in the body of the command `get9`:

PROGRAM 6.18. Defining command `get3` locally in the command `get9`

```
1  # Collect 9 water bottles:
2  def get9
3
4    # Collect three water bottles
5    def get3
6      repeat 2
7        get
8        go
9      get
10
11   # Main body of the command:
12   get3
13   right
```

```
14    go
15    right
16    get3
17    left
18    go
19    left
20    get3
21    go
22
23  # Main program:
24  # Get to the 1st crate:
25  go
26  # Collect the 1st crate:
27  get9
28  # Collect the 2nd crate:
29  get9
30  # Get to the 3rd crate:
31  left
32  repeat 5
33    go
34  right
35  # Collect the 3rd crate:
36  get9
37  # Make one last step home:
38  go
```

Importantly, the command `get3` is now *local* to the command `get9`, meaning that it is only valid within the body of the command `get9`.

If we tried to call the command `get3` outside the body of the command `get9`, the program would crash with an error message. In larger programs, limiting the validity of commands helps improve code structure.

6.13. Create your own Karel language!

Being able to define your own commands is powerful. If you like, you can now create your own language for Karel, adding all the commands which you were missing! For

example, the Karel language does not have a command to turn around 180 degrees, so let's add it:

PROGRAM 6.19. New command to turn around 180 degrees

```
# Turn around 180 degrees:
def turn180
  repeat 2
    left
```

With this new command in hand, many algorithms become simpler to implement, such as the First Maze Algorithm:

PROGRAM 6.20. First Maze Algorithm with the `turn180` command

```
# First Maze Algorithm:
while not home
  if wall
    left
    if wall
      turn180
  go
```

Speaking of the First Maze Algorithm, why don't we just define a new command `move` to move one step forward in the maze:

PROGRAM 6.21. New command `move`

```
# Move one step forward in the maze:
def move
  if wall
    left
    if wall
      turn180
  go
```

Then, passing through an unknown maze becomes extremely simple:

PROGRAM 6.22. Compact version of the FMA using the new command `move`

```
# Pass through an unknown maze using the FMA:
while not home
  move
```

Now let's change the subject. If your native language is different from English, you can make Karel speak your language! For example, "get" is "ukuthola" in Zulu:

PROGRAM 6.23. Translating command `get` to Zulu

```
# Command 'get' in Zulu:
def ukuthola
  get
```

6.14. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 6.1. *Is copying and pasting code a bad thing?*

A *No because the length of the program grows more quickly.*
B *Yes because the program becomes prone to mistakes.*
C *No because the program will run faster.*
D *No because this makes programming easier.*

QUESTION 6.2. *Is it a good idea to split complex tasks into simpler ones?*

A *No because it is a loss of time.*
B *No because it's more work.*
C *Yes because the simpler tasks are easier to solve.*
D *No because an experienced programmer just solves the whole task at once.*

QUESTION 6.3. *When does one need to identify repeating patterns?*

A *When working with* `repeat` *loops.*
B *When working with* `while` *loops.*
C *When working with custom commands.*
D *When debugging programs.*

QUESTION 6.4. *What keyword does Karel use to define new commands?*

A `new`
B `define`
C `var`
D `def`

QUESTION 6.5. *What keyword does Karel use to terminate and exit custom commands?*

A `end`
B `terminate`
C `exit`

D `return`

QUESTION 6.6. *Can custom commands define other custom commands in their body?*

A Yes.

B No.

C Only if the locally-defined command is a one-liner.

D Only if the locally-defined command does not contain a `return` *statement.*

QUESTION 6.7. *Can custom commands define other custom commands in their body?*

A Yes.

B No.

C Only if the locally-defined command is a one-liner.

D Only if the locally-defined command does not contain a `return` *statement.*

QUESTION 6.8. *What is the purpose of the Second Maze Algorithm (SMA)?*

A To pass through a maze without loops and dead ends.

B To pass through a maze without forsk, loops and dead ends.

C To follow an arbitrary winding wall or line on the ground.

D To find a way out of an arbitrary maze.

QUESTION 6.9. *The left-handed version of the SMA assumes that*

A Karel stands next to a wall, and the wall is on his left.

B Karel stands next to a wall, and the wall is on his right.

C Karel faces the wall.

D Karel stands with his back turned to the wall.

QUESTION 6.10. *What of the following actions represent one operation in the Karel language?*

A Karel makes one step forward.

B Karel turns around 180 degrees.

C Karel places an object on the ground.

D Karel walks from one side of the maze to the opposite one.

QUESTION 6.11. *When is a program efficient?*

A When it has as few lines as possible.

B When it is carefully thought-out.

C When it uses nested loops and conditions.

D When it solves the given task using as few operations as possible.

7. Variables

In this chapter you will learn:

- What are variables and why they are useful.
- What types of variables are provided in the Karel language.
- What are admissible and inadmissible names of variables.
- How to create and initialize variables.
- How to change their values and display them.
- About the assignment operator =.
- About arithmetic operators +, −, ⋆, / and +=, −=, ⋆=, /=.
- About comparison operators ==, !=, <, <=, > and >=.

7.1. What are variables and why are they useful?

In computer programming, variables are used to store information for later use. Karel can use *numerical variables* to remember *numbers*:

- How many objects he found.
- How many steps he took.
- Sizes of obstacles and distances between objects.
- GPS positions of objects or obstacles.
- How many objects are in his bag.
- Results of math calculations, etc.

He can also use *text string variables* to store *text*, such as:

- Names of objects that he found.
- Text that he obtained when translating Morse to English.
- Letters that he recognized when reading Braille alphabet, etc.

Karel can also use *Boolean (logical) variables* to store outcomes of conditions in the form `True` or `False`. These will be discussed in Chapter 11.

Variables are used to store useful information for later use.

7.2. Types of variables

In summary, the Karel language provides the four standard types of variables which are used in most programming languages:

- *Integer variables* to store whole numbers.
- *Floating-point variables* to store real numbers.
- *Text string variables* to store text strings.
- *Boolean variables* to store Boolean (truth) values `True` and `False`.

In addition to that, other programming languages usually provide other types of variables aimed at specific applications. For example, C/C++ which is heavily computational provides six types of integer variables (char, int, short int, long int, unsigned short int, unsigned long int) and two types of floating-point variables for single and double precision numbers. Python provides the `complex` type to work with complex numbers. Python also provides aggregate types including lists, tuples, dictionaries and sets. Of those, Karel provides lists - these can be used to do amazing things, as you will see in Chapter 13.

7.3. Choosing the names of variables

The name of a variable can be almost any combination of letters of the alphabet, integer numbers, and the underscore character `'_'`, such as `N`, `m`, `c1`, `alpha`, `Beta`, `GAMMA`, `my_text`, etc. Only remember the following important restriction:

> The name of a variable must not begin with a number.

So, names such as `1a` or `8_bits` cannot be used. This is the same in other programming languages too. Names starting with one or more underscores such as `_x` or `___my_name___` are fine.

Also, importantly, recall from Section 2.14 (page 21) that the Karel language is case-sensitive. This means that, for example, the command `left` typed as `Left` or `LEFT` will not work. The same holds for the names of variables:

> Variables whose names only differ in the case of some letters,
> such as `n` and `N`, are different variables in Karel.

The case-sensitivity of Karel comes from Python.

7.4. Creating and initializing numerical variables

The following sample code creates several numerical variables, both integer and floating-point, and initializes them with different values:

PROGRAM 7.1. Creating and initializing numerical variables

```
1  # Integer variables:
2  a = 0
3  my_age = 12
4  seconds_per_minute = 60
5  hours_per_day = 24
6  temperature = -15
7
8  # Floating-point variables:
9  half = 0.5
10 pi = 3.14159265358979323846
```

7.5. Assignment operator =

Notice that in the previous program, the symbol `'='` was used to assign values to variables. This is completely different from mathematics where the same symbol is used for comparisons and equations.

> In computer programming, the symbol `'='` has a different meaning from mathematics: `a = 0` means "assign the value `0` to the variable `a`".

The mathematical equality is expressed using the symbol ==. Comparison operators will be discussed in Section 7.10 (page 133).

7.6. Displaying the values of variables

The value of a variable can change during runtime. Therefore it is important to be able to display its value at any time. This can be done using the `print` statement. The following sample code illustrates various forms of its use:

PROGRAM 7.2. Displaying values of variables

```
1  a = 3
2  b = 5
```

```
d = 15.5
print(a)
print("Value of b is", b)
print("Distance d is", d, "meters.")
```

The output of this program is:

```
3
Value of b is 5
Distance d is 15.5 meters.
```

7.7. Basic math operations with variables

You already know how to create variables and initialize them with values. These values may come not only as raw numbers, but also from other variables and/or math expressions. Let's begin with addition (symbol `'+'`) and subtraction (symbol `'-'`):

PROGRAM 7.3. Addition and subtraction

```
item_1 = 12.5
item_2 = 7.5
amount_start = 30
total_cost = item_1 + item_2
amount_end = amount_start - total_cost
print("Initial amount:", amount_start)
print("Total cost of both items:", total_cost)
print("Remaining amount:", amount_end)
```

Output:

```
Initial amount: 30
Total cost of both items: 20
Remaining amount: 10
```

The symbols `'*'` and `'/'` represent multiplication and division, respectively:

PROGRAM 7.4. Multiplication and division

```
secs = 5400
hrs = secs / 3600
mins = hrs * 60
print(secs, "seconds =", mins, "minutes =", hrs, "hours")
```

Output:

```
5400 seconds = 90 minutes = 1.5 hours
```

7.8. Keywords `inc` and `dec`

Karel provides the keywords `inc` and `dec` which can be used to increase and decrease the value of any numerical variable by one, respectively:

PROGRAM 7.5. Increasing and decreasing the values of variables by one

```
1 m1 = 10
2 m2 = 20
3 inc(m1)
4 print("m1 =", m1)
5 dec(m2)
6 print("m2 =", m2)
```

Output:

```
m1 = 11
m2 = 19
```

Both the commands `inc` and `dec` can also be used to increase and decrease the values of numerical variables by more than one:

PROGRAM 7.6. Increasing and decreasing the values of variables by more than one

```
1 m1 = 10
2 m2 = 20
3 inc(m1, 3)
4 print("m1 =", m1)
5 dec(m2, 5)
6 print("m2 =", m2)
```

Output:

```
m1 = 13
m2 = 15
```

7.9. Arithmetic operators `+=`, `-=`, `*=` and `/=`

Karel can modify the values of numerical variables using the Python operators `+=`, `-=`, `*=` and `/=`. Here `a += 2` is exactly the same as `a = a + 2`, `b -= 3` is the same as `b = b - 2`, `c *= 4` means `c = c * 4`, and `d /= 5` is `d = d / 5`.

PROGRAM 7.7. Operators +=, −=, *= and /=

```
a = 11
b = 15
c = 5
d = 25
a += 2
b -= 3
c *= 4
d /= 5
print("a =", a)
print("b =", b)
print("c =", c)
print("d =", d)
```

Output:

```
a = 13
b = 12
c = 20
d = 5
```

7.10. Comparison operators ==, !=, <, <=, > and >=

Sometimes one needs to decide whether the value of a variable (say X) is equal to the value of another variable (say Y). As you already know, typing `if X = Y` is wrong because the assignment operator = will just assign the value of Y to the variable X. The mathematical equality should be expressed using the symbol == which means "equal to":

PROGRAM 7.8. Comparison operator ==

```
X = 99
Y = X
if X == Y
  print("X and Y are the same.")
else
  print("X and Y are different.")
```

Output:

```
X and Y are the same.
```

The comparison operator != is the opposite of ==. It means "not equal to":

PROGRAM 7.9. Comparison operator !=

```
X = 1
Y = 1.1
if X != Y
  print("X and Y are different.")
else
  print("X is the same as Y.")
```

Output:

```
X and Y are different.
```

The comparison operator < means "less than":

PROGRAM 7.10. Comparison operator <

```
X = -3
Y = X + 6
if X < Y
  print("X is less than Y.")
else
  print("X is greater than or equal to Y.")
```

Output:

```
X is less than Y.
```

The comparison operator > means "greater than":

PROGRAM 7.11. Comparison operator >

```
X = 1
Y = X
dec(Y)
if X > Y
  print("X is greater than Y.")
else
  print("X is less than or equal to Y.")
```

Output:

```
X is greater than Y.
```

The remaining two operators, <= and >= are similar to < and >, respectively, except they admit equality. Let's just show an example for <=:

PROGRAM 7.12. Comparison operator <=

```
X = 2
Y = 2 * X - 2
if X <= Y
  print("X is less than or equal to Y.")
else
  print("X is greater than Y.")
```

Output:

```
X is less than or equal to Y.
```

7.11. Counting maps

Hooray! Finally we know variables well enough to tackle first programming challenges with them. Today, Karel's task is to count maps which lie randomly between him and his home square. He does not know how many there are. The resulting variable should be named `nmaps`. He should display the value of this variable at the end.



Karel is counting maps.

The easiest way to do this is to first solve a simpler task - just go and collect all the maps. This is something you can easily do by now:

```
while not home
  if map
    get
  go
```

In a second step, we will modify this program to create a new variable `nmaps` at the beginning and initialize it with zero. Then, if Karel finds a map, instead of collecting it he will increase the value of `nmaps` by one. And, at the end he will use the `print` statement to display the value of `nmaps`:

PROGRAM 7.13. Count the maps!

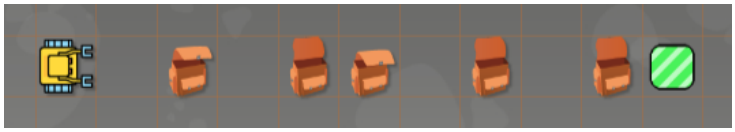
```
nmaps = 0
while not home
  if map
    inc(nmaps)
  go
print("I found", nmaps, "maps.")
```

Output:

```
I found 5 maps.
```

7.12. Water supply

Karel is carrying an unknown number of water bottles, and there is an unknown number of bags between him and his home square. He needs to put one water bottle in each one. Also, he should count how many water bottles he has at the beginning, subtract the number of water bottles which he places in the bags, and then store the number of remaining water bottles in a variable named `wb`.



Karel is putting water bottles in bags.

The only way for the robot to count the water bottles he has is to place all of them on the ground, and then collect them again:

```
wb = 0
while not empty
  put
  inc(wb)
while bottle
  get
```

Then, he needs to walk to his home square, and put a water bottle in each bag. Since Karel might have fewer water bottles than there are bags, the `put` command should only be executed if the robot's bag is not empty. Here is the complete code:

PROGRAM 7.14. Put the water bottles in the bags!

```
# Count the water bottles:
wb = 0
while not empty
  put
  inc(wb)
while bottle
  get
print("I am starting with", wb, "water bottles.")
# Walk to home square and put one water bottle in each bag:
while not home
  if bag and (not empty)
    put
    dec(wb)
    print("Found a bag - placing a water bottle.")
  go
print("Now I only have", wb, "water bottles left.")
```

Below is the text output. It reveals that Karel had 23 water bottles at the beginning:

```
I am starting with 23 water bottles.
Found a bag - placing a water bottle.
Found a bag - placing a water bottle.
Found a bag - placing a water bottle.
Found a bag - placing a water bottle.
Found a bag - placing a water bottle.
Now I only have 18 water bottles left.
```

7.13. Karel and the Fibonacci sequence

The Fibonacci sequence is one of the most famous sequences of mathematics. It begins with 1, 1 and each new number is the sum of the last two. So, the third number is 1 + 1 = 2. The fourth number is 1 + 2 = 3. The fifth number is 2 + 3 = 5 and so on. The first ten numbers in the sequence are

1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

Today, Karel wants to create this sequence by himself! He begins with two beepers on the floor, and two variables `n1 = 1` and `n2 = 1` for the first two numbers in the sequence:



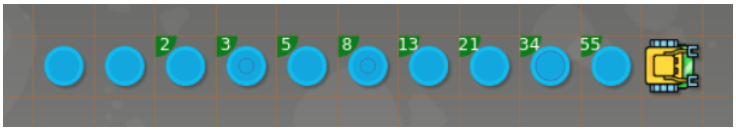
Karel is going to create the Fibonacci sequence.

In order to obtain the third number, he will create a helper variable `new_n` and assign the sum `n1 + n2` to it. Then he will place `new_n` beepers on the ground. He will assign the value of `n2` to `n1` by executing `n1 = n2`, and then also the value of `new_n` to `n2` by executing `n2 = new_n`. Then he will move to the next grid square. Hence, after this he will have the last two numbers of the sequence stored in `n1` and `n2` again. This process will be repeated eight times, because Karel wants to calculate the first 10 numbers of the Fibonacci sequence. Here is the corresponding program:

PROGRAM 7.15. Create the Fibonacci sequence!

```
1 n1 = 1
2 n2 = 1
3 repeat 8
4   new_n = n1 + n2
5   repeat new_n
6     put
7   n1 = n2
8   n2 = new_n
9   go
```

And indeed, at the end he obtains the correct result:



The first 10 numbers of the Fibonacci sequence.

7.14. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 7.1. *Why do we use variables?*

A To store useful information for later use.

B To make the program run faster.

C To make the program use less memory.

D To make the program shorter.

QUESTION 7.2. *What types of variables are provided in Karel?*

A Numerical variables.

B Text string variables.

C Boolean (logical variables).

D Complex variables.

QUESTION 7.3. *Which of the following are valid names of variables?*

A `my_name`

B `1st_name`

C `_x_`

D `NUM1`

QUESTION 7.4. *Are* `x1` *and* `X1` *the same variable in Karel?*

A Yes

B No

QUESTION 7.5. *How does one create a new variable* `A` *and assign the value 10 to it?*

A `A(10)`

B `A := 10`

C `A == 10`

D `A = 10`

QUESTION 7.6. *What is the correct way to display the value of a numerical variable* `A`*?*

A `print('A')`

B `print("A")`

C `print(A)`

D `print[A]`

QUESTION 7.7. *The initial value of a variable* `x` *is 5. What will its value be after executing* `inc(x, 3)`*?*

A 5

B 8

C 2

D An error will be thrown.

QUESTION 7.8. *The initial value of a variable* `y` *is 9. What will its value be after executing* `dec(y)`*?*

A 10

B 8

C 9

D An error will be thrown.

QUESTION 7.9. *The initial value of a variable* `n1` *is 3. What will its value be after executing* `n1 -= 4`*?*

A 1

B -1

C -3

D An error will be thrown.

QUESTION 7.10. *The initial value of a variable* `n2` *is 12. What will its value be after executing* `n2 /= 4`*?*

A 8

B 16

C 3

D An error will be thrown.

QUESTION 7.11. *The value of a variable* `m` *is 5. Will the condition* `if m != 5` *be satisfied?*

A Yes

B No

QUESTION 7.12. *The value of a variable* `M` *is 10. Will the condition* `if M <= 11` *be satisfied?*

A Yes

B No

QUESTION 7.13. *What numbers belong to the Fibonacci sequence?*

A 1

B 5

C 33

D 55

QUESTION 7.14. *The initial values of the variables* `x`, `y` *and* `z` *are 1, 2, 3, respectively. What values will they have after executing* `x += y`, `y *= x` *and* `z = y / z` *?*

A `x` *will be 2,* `y` *will be 4,* `z` *will be 6.*

B `x` *will be 3,* `y` *will be 6,* `z` *will be 2.*

C `x` *will be 2,* `y` *will be 4,* `z` *will be 3/4.*

D `x` *will be 3,* `y` *will be 2,* `z` *will be 6.*

8. Functions

In this chapter you will learn:

- How to define and use functions which return values.
- About functions which accept arguments.
- About local and global scopes.
- The difference between local and global variables.
- That using global variables may lead to problems.
- If functions can change the values of their arguments.

8.1. Defining and using functions

You already know how to define and use custom commands. A function is very similar to a command but in addition it uses the `return` statement to return a value. Functions also can accept arguments - this will be discussed in Section 8.6 (page 149). But for the time being let us stay with functions which do not need any arguments.

One of them is a simple function `distance` which will measure Karel's distance to the nearest wall. The robot will simply walk to the wall, count his steps, and at the end return their number:



Karel measures the distance to the nearest wall.

Here is the corresponding program:

PROGRAM 8.1. Karel measures the distance to the nearest wall

```
1 # Walk to the nearest wall, count steps, and return their number:
2 def distance
3   d = 0
4   while not wall
5     go
6     inc(d)
```

```
7     return d
8
9   # Call the function and display the result:
10  result = distance
11  print("The distance was", result, "steps.")
```

Notice that the function is first defined on line 2 and then called on line 10. Just defining a function, without calling it, does nothing.

On line 10, the value returned from the function was assigned to the variable `result`. Since the sole purpose of this variable was to provide the value to the `print` statement, it could be omitted. The function call could be placed into the `print` statement directly:

PROGRAM 8.2. Function call is placed directly into the `print` statement

```
1   # Walk to the nearest wall, count steps, and return their number:
2   def distance
3     d = 0
4     while not wall
5       go
6       inc(d)
7     return d
8
9   # Call the function and display the result:
10  print("The distance was", distance, "steps.")
```

And here is the outcome of the program:

```
The distance was 7 steps.
```

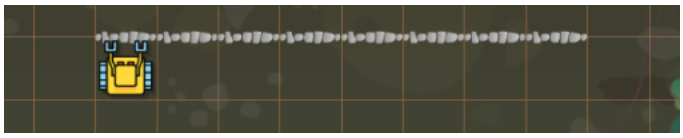
8.2. Builder

This time, Karel's task is to measure the length of a straight wall. He stands just South of it, facing North.



Karel needs to measure the length of a wall.

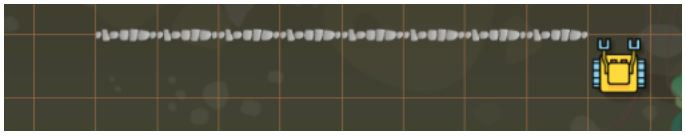
First he must locate one end of the wall - say the left one:



This can be done using the following code:

```
# Locate the left end of the wall:
while wall
  left
  go
  right
right
go
left
```

Then, he can create a new variable named `length`, and move along the wall until he reaches its other end:



After every step, he increases the value of `length` by one:

```
# Measure the length:
length = 0
while wall
  right
  go
  left
  inc(length)
```

The above two code segments can be used to define a function `measure` which will be easily reusable to measure the length of any straight wall:

PROGRAM 8.3. Measure the length of any straight wall

```
def measure
  # Locate the left end of the wall:
  while wall
    left
    go
```

```
    right
  right
  go
  left
  # Measure the length:
  length = 0
  while wall
    right
    go
    left
    inc(length)
  return length

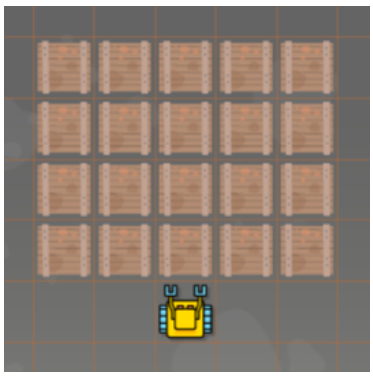
# Call the function and display the result:
print("The length of the wall is:", measure)
```

For the wall shown above, the program will display the following message:

```
The length of the wall is: 8
```

8.3. Storekeeper

Today, Karel needs to count crates which form a rectangular pattern. Its dimensions are unknown. He stands just South of it, facing North. For example, the rectangle below has 4 rows and each row has 5 crates, so altogether there are 4 * 5 = 20 crates.



Karel is counting crates.

Karel will count the crates as follows:

(1) Count the crates in the bottom row.

(2) Store the result in a variable named `t`.

(3) Create a new variable `total` and initialize it with 0.

(4) Walk North along the crates, and for each row add `t` to `total`.

So, for the example above, Karel will calculate `total = 5 + 5 + 5 + 5 = 20`. In the program below, notice that to count the crates in the first row he uses exactly the same method that he used to measure the length of the wall in Section 8.2 (page 142):

PROGRAM 8.4. Count the crates!

```
# Count crates in one row:
def measure
  # Locate left-most crate:
  while crate
    left
    go
    right
  right
  go
  left
  # Count crates:
  r = 0
  while crate
    right
    go
    left
    inc(r)
  go
  return r

# Count crates in the rectangle:
def multiply
  # Count the crates in the first row:
  t = measure
  # Turn to face the crate:
  left
  # Initialize variable 'total' with 0:
  total = 0
  # Walk north. For each row, add 't' to 'total':
  while crate
    inc(total, t)
    right
```

```
33      go
34      left
35    return total
36
37  # Main program:
38  print("I counted", multiply, "crates!")
```

And here is the corresponding output:

```
I counted 20 crates!
```

8.4. Local variables and local scope

The function `distance` from Section 8.1 (page 141),

```
1  def distance
2    d = 0
3    while not wall
4      go
5      inc(d)
6    return d
```

the function `measure` from Section 8.2 (page 142),

```
1  def measure
2    # Locate the left end of the wall:
3    while wall
4      left
5      go
6      right
7    right
8    go
9    left
10   # Measure the length:
11   length = 0
12   while wall
13     right
14     go
15     left
16     inc(length)
17   return length
```

as well as the functions `row` and `multiply` from Section 8.3 (page 144) have one thing in common: They both introduce a variable which is only used in the function and not elsewhere in the program. Such a variable is called a *local variable*. The validity of a local variable is only within the body of the function or command where it was defined.

We say that a local variable is defined in the *local scope*.
A local variable is only valid in the function or command where it was defined.

This is analogous to what you already know from Section 6.12 (page 123) - commands defined inside of other commands were *local* as well, and only valid within the command or function where they were defined.

Let's see what happens if we try to use a local variable outside of its local scope:

```
1 def distance
2   d = 0
3   while not wall
4     go
5     inc(d)
6   return d
7
8 print("The distance was", distance, "steps.")
9 print("The last value of d was:", d)
```

The program will crash with an error message:

```
Run-time error on line 9:
Unknown variable, command or function 'd'
```

8.5. Global variables and global scope

The main program is called *global scope*. A variable which is defined in the global scope is called *global*.

Global variables are valid not only everywhere in the main program,
but also in all functions and commands.

Global variables are sometimes used when they are needed in multiple commands or functions. However, their use should be kept to a minimum because they make the code prone to mistakes. For example, one may define a global variable in one program module, then forget about it, and define it again, but with a different value, in another module. Or, one can define in another module a command or function with the same name. Or one can inadvertently change the value of a global variable inside a function or command. These errors are then extremely difficult to find.

> The use of global variables should be kept to a minimum
> because they make the program prone to mistakes.

The following example is an example of bad programming. It uses a global variable to pass a value into a function. The function `back` will make Karel move back `N` steps:

```
1 def back
2   left
3   left
4   repeat N
5     go
6   right
7   right
```

Then in the main program one could write

```
1 N = 4
2 back
```

and Karel would move back 4 steps. Elsewhere one could write

```
1 N = 7
2 back
```

and the robot would move back 7 steps. However, the global variable `N` "polluted" the main scope and made the program prone to mistakes. In the following section we will show you a much better way to define such a function `back`.

8.6. Functions that accept arguments

In Karel (same as in Python and other programming languages), a function can accept one or more *arguments*. When defining such a function, one introduces one or more parameters in paretheses following the function name, such as in `def place(p)`. The names of the parameters then can be used within the function's body as local variables.

The following function `place(p)` will make Karel place `p` objects on the ground beneath him (it assumes that there are enough objects in the robot's bag):

```
1 def place(p)
2   repeat p
3     put
```

When one wants Karel to place two objects, one calls the function name with the number 2 in parentheses:

```
1 place(2)
```

And when calling

```
1 place(5)
```

the robot will place 5 objects. To give another example, let's define a function `back(n)` to move Karel back `n` steps:

```
1 def back(n)
2   left
3   left
4   repeat n
5     go
6   right
7   right
```

Then, to make Karel move back 4 steps, one calls:

```
1 back(4)
```

And when one needs to move the robot 7 steps back, one types:

```
1 back(7)
```

Functions can accept two or even more arguments. For example, the following function `rectangle(a, b)` will make Karel create a rectangle of gears with side lengths `a` and `b`:

PROGRAM 8.5. Create a rectangle of gears!

```
# Create a rectangle of gears with dimensions a x b:
def rectangle(a, b)
  repeat 2
    repeat a-1
      put
      go
    left
    repeat b-1
      put
      go
    left

# Create a 6 x 4 rectangle:
rectangle(6, 4)
```

Here is the outcome of the program:



8.7. Can a function change the values of its arguments?

This is an interesting question. Any function may attempt to modify the value of the arguments which are passed to it. For example, the following function `double(d)` executes the command `d *= 2` in its body. Now the question is - when the function is called with an argument `x` as `double(x)`, will the value of `x` be doubled when the function finishes?

```
# Double the value of the argument and return it:
def double(d)
  d *= 2
  return d
```

```
5
6  # Main program:
7  x = 2
8  print("Value of x before function call:", x)
9  xx = double(x)
10 print("Value of x after function call: ", x)
11 print("Value of xx:", xx)
```

The output of the program shows that the answer is "no". The function was not able to change the value of the argument `x` which was passed to it:

```
Value of x before function call: 2
Value of x after function call:  2
Value of xx: 4
```

We can try the same with a text string variable, and we can actually pass it to the same function `double(d)` because the operation `*=` works for text strings:

```
1  # Double the value of the argument and return it:
2  def double(d)
3    d *= 2
4    return d
5
6  # Main program:
7  x = "Karel"
8  print("Value of x before function call:", x)
9  xx = double(x)
10 print("Value of x after function call: ", x)
11 print("Value of xx:", xx)
```

The output of the program shows that the answer is "no" again:

```
Value of x before function call: Karel
Value of x after function call:  Karel
Value of xx: KarelKarel
```

The values of numerical variables, text strings, and Booleans *cannot be changed by functions*. We say that they are *immutable* types.

8.8. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 8.1. *What is the difference between a command and a function?*

A Commands cannot use the `return` *statement.*

B Commands do not return values.

C Functions return values and can accept arguments.

D There is no difference.

QUESTION 8.2. *Check all correct statements!*

A A function which is defined must also be called.

B When a function is defined, it is not automatically called.

C A function can just be called, it does not have to be defined.

D When a function is defined but not called, nothing happens.

QUESTION 8.3. *Can a function call be placed directly into the* `print` *statement?*

A Yes

B No

QUESTION 8.4. *Check all true statements about local variables!*

A They are defined in the global scope, and valid everywhere in the main program.

B They are defined in a command or function, and valid only within that command or function.

C They are defined in the global scope, and valid only in some command or function.

D They are defined in a command or function, and valid everywhere in the main program.

QUESTION 8.5. *Check all true statements about global variables!*

A They are defined in a command or function, and valid everywhere in the main program.

B They are defined in the global scope, and valid only in some command or function.

C They are defined in a command or function, and valid only within that command or function.

D They are defined in the global scope, and valid everywhere in the main program.

QUESTION 8.6. *Which types of variables in Karel can have their values changed by functions?*

A Numerical variables.

B Text string variables.

C Boolean variables.

D Complex variables.

9. Text Strings

In this chapter you will learn:

- How to work with text strings and text string variables.
- How to assign text strings to variables.
- How to display them and compare with each other.
- How to concatenate them and multiply with integers.
- How to obtain the length of a text string.
- How to use the `for` loop to parse text strings one character at a time.
- How to extract individual characters from a text string using their indices.
- How to reverse and slice text strings.
- How to display quotes and check for substrings.

At the end we will solve some very interesting tasks:

- Counting occurrences of a substring in a text string.
- Finding where a substring is located in a text string.
- Searching for substrings and replacing them with others.
- Removing substrings from text strings.
- Swapping substrings in a text string,
- and even executing a text string as Karel code!

9.1. Raw text strings and text string variables

In the Karel language, working with text strings is similar to Python, although Karel does not implement all Python's functionality.

A *text string* is a sequence of characters enclosed in either single quotes, such as `'Hello World!'` or in double quotes, such as `"I am Karel"`. It does not matter whether single or double quotes are used. A text string which contains no characters is called *empty text string*. The following screenshot shows a few examples:

```
empty_string = ""
empty_character = ' '
my_name = "Karel"
price = "$99"
novelist = "Karel Čapek"
spanish_text = "¡Te veo mañana!"
russian_text = "Как тебя зовут?"
german_text = "Grüß dich!"
greek_text = "Γεια σας, είμαι ο Κάρελ"
chinese_text = "卡雷尔是最好的！"
```

Sample text strings.

9.2. Assigning, displaying, and comparing text strings

A text string can be assigned to a variable using the standard assignment operator =:

```
1 name = 'Karel'
```

Both raw text strings and text string variables can be displayed using the `print` statement exactly as you would expect:

```
1 print(name)
2 print("My name is", name)
```

Output:

```
Karel
My name is Karel
```

Text strings (in either raw form or as text string variables) can be compared using the same comparison operator == as numbers:

```
1 str1 = 'Karel'
2 if str1 == 'Karel'
3   print("That's him, Karel!")
4 else
5     print("No, that's not him.")
```

Output:

```
That's him, Karel!
```

9.3. Concatenating text strings

One can concatenate two or more text strings using the same + symbol which is used to add numbers:

PROGRAM 9.1. Concatenating text strings

```
1 str1 = 'Tomorrow '
2 str2 = 'I will visit '
3 result = str1 + str2 + "Karel."
4 print(result)
```

As you can see, it does not matter whether one concatenates raw text strings or text string variables. The output of this program is:

```
Tomorrow I will visit Karel.
```

Notice that each of the text strings `str1` and `str2` had an empty space at the end. It is important to account for empty spaces, they are not inserted automatically. Let's see what happens when we leave them out:

PROGRAM 9.2. Pay attention to empty spaces!

```
1 str1 = 'Tomorrow'
2 str2 = 'I will visit'
3 result = str1 + str2 + "Karel."
4 print(result)
```

Output:

```
TomorrowI will visitKarel.
```

9.4. Multiplying text strings with integers

Any text string (in raw form or as a variable) can be multiplied with a (positive) integer, as shown in the following example:

PROGRAM 9.3. Multiplying text strings with integers

```
1 name = 'Karel '
2 text = name * 5 + 'everywhere!'
3 print(text)
```

Output:

```
Karel Karel Karel Karel Karel everywhere!
```

9.5. Length of a text string

Karel provides the Python function `len` to obtain the length of a text string. The following example shows various flavors of using it:

```
1 print(len('Karel'))
2 txt1 = 'I am Karel'
3 print(len(txt1))
4 txt2 = 'Karel rocks!'
5 n = len(txt2)
6 print(n)
```

Output:

```
5
10
12
```

9.6. Parsing a text string with the `for` loop

Karel provides the Python `for` loop to parse text strings one character at a time. The keyword `for` is followed by a variable for the individual characters, the keyword `in` and the name of the text string to be parsed. For example, let's parse the text string `'Karel'` and just display all the characters:

```
1 for c in 'Karel'
2   print(c)
```

Output:

```
K
a
r
e
l
```

9.7. Reversing text strings

As another example, let's use the `for` loop to reverse a text string!

```
name = 'Karel the Robot'
r = ''
for c in name
  r = c + r
print(r)
```

Output:

```
toboR eht leraK
```

9.8. Extracting individual characters by their indices

Characters in a text string are internally enumerated, starting with 0 (first character), 1 (second character), 2 (third character), etc. These numbers are called *indices* (the singular of the word is *index*). Appending the index in square brackets at the end of either a raw text string or a text string variable yields the desired character. For illustration, let's extract all five individual characters from the text string `'Robot'` using their indices 0, 1, 2, 3 and 4:

```
name = 'Karel the Robot'
print(name[0])
print(name[1])
print(name[2])
print(name[3])
print(name[4])
```

Output:

```
K
a
r
e
l
```

Keep in mind that the indices start from zero. Therefore, the 1st character has index 0, the 2nd character has index 1, the 5th character has index 4, etc.

Now let's extract individual characters from a raw text string:

```
print('Karel the Robot'[10])
print('Karel the Robot'[11])
print('Karel the Robot'[12])
print('Karel the Robot'[13])
print('Karel the Robot'[14])
```

Output:

```
R
o
b
o
t
```

9.9. Using negative indices

When one needs to extract characters from the end of a text string, it is possible to use negative indices. The last character has index -1, the one before last has index -2, etc. Let's illustrate this on a simple example:

```
name = 'Karel the Robot'
print(name[-5])
print(name[-4])
print(name[-3])
print(name[-2])
print(name[-1])
```

Output:

```
R
o
b
o
t
```

The last character in a text string has index -1,
the one before last has index -2, etc.

9.10. Slicing text strings

Slicing text strings is another useful functionality provided by Python. Sometimes one needs to extract not only one character, but an entire word or a substring. Actually, you already know how to do this: To extract a five-character word from the beginning of a text string `name`, one can type `name[0] + name[1] + name[2] + name[3] + name[4]`:

```
1 name = 'Karel the Robot'
2 first = name[0] + name[1] + name[2] + name[3] + name[4]
3 print(first)
```

Output:

```
Karel
```

But, the same can be done more easily using *slicing*. A five-character slice from the beginning of a text string `name` is `name[:5]`:

```
1 name = 'Karel the Robot'
2 first = name[:5]
3 print(first)
```

Output:

```
Karel
```

A three-character slice starting with the second letter is `name[1:4]`:

```
1 name = 'Karel the Robot'
2 word = name[1:4]
3 print(word)
```

Output:

```
are
```

A five-character slice from the end of the text string `name` is `name[-5:]`:

```
1 name = 'Karel the Robot'
2 last = name[-5:]
3 print(last)
```

Output:

```
Robot
```

A three-character slice starting with the fifth letter from the end is `name[-5:-2]`:

```
1 name = 'Karel the Robot'
2 word = name[-5:-2]
3 print(word)
```

Output:

```
Rob
```

9.11. Displaying quotes in text strings

Displaying quotes is a bit tricky, because these are the symbols which are used to define text strings. To illustrate the problem one can get into, let's try to display the text `She said: "Hello, it is nice to meet you.":`

```
1 print("She said: "Hello, it is nice to meet you."")
```

Output:

```
Unexpected name 'Hello'.
Line 1:
print("She said: "Hello, it is nice to meet you."")
```

The reason why this did not work is that the interpreter found a complete and correct text string `"She said: "` in the `print` function. Then it expected either a closing parenthesis, or a comma, or perhaps one of the `+` or `*` operators. But instead, it found `Hello` which did not make any sense. Was this a misplaced variable? The interpreter had no choice other than to throw an error.

The way to prevent this confusion is to enclose the text string that contains double quotes in single quotes:

```
1 print('She said: "Hello, it is nice to meet you."')
```

Output:

```
She said: "Hello, it is nice to meet you."
```

An analogous problem will arise when one tries to display a text string that contains single quotes by enclosing it in single quotes:

```
1 print('Hi, it's Karel!')
```

Output:

```
Unexpected name 's'.
Line 1:
print('Hi, it's Karel!')
```

The problem can be fixed by enclosing the text string in double quotes:

```
1 print("Hi, it's Karel!")
```

Output:

```
Hi, it's Karel!
```

9.12. Checking for substrings

Another Python feature provided in the Karel language is the ability to check if a text string contains a given substring. This is done using the keyword `in`:

PROGRAM 9.4. Checking for a raw substring

```
1 txt = "The robot is really good. His name is Karel."
2 if "Karel" in txt
3   print("The text string contains the word 'Karel'.")
4 else
5   print("The text string does not contain the word 'Karel'.")
```

Output:

```
The text string contains the word 'Karel'.
```

And of course this can be done using a variable as well:

PROGRAM 9.5. Checking for a text string variable

```
txt = "Karel the IV was a Roman emperor. But Karel the robot is cooler."
name = "Karel"
if name in txt
  print("The text string contains the word 'Karel'.")
else
  print("The text string does not contain the word 'Karel'.")
```

Output:

```
The text string contains the word 'Karel'.
```

9.13. Counting occurrences of substrings

Sometimes one needs to count how many times certain text string is present in another text string. Python has a built-in functionality for this which is not present in Karel. So, let's implement it! Your task is to write a function `count(sub, txt)` to count the number of occurrences of a substring `sub` in a text string `txt`. If the substring is not found, the function should return zero. Try to solve this task by yourself before looking at the solution below!

Solution: You already know from Section 9.10 (page 159) how to slice text strings. As the first thing we will introduce a counting variable (named for example `result`), initialize it with zero, and measure the length of both `sub` and `txt`. Let's denote their lengths by `l1` and `l2`, respectively. The algorithm will be very simple. We will take the first `[0:l1]` slice of `txt` and compare it with `sub`. If there is a match, we will increase `result` by one. Then we will do the same for the `[1:l1+1]` slice of `txt`, for the `[2:l1+2]` slice of `txt`, etc. There are `l2-l1+1` such slices. At the end, we will return the value of `result`. Here is the corresponding Karel code:

PROGRAM 9.6. Count the occurrences of a substring in a text string!

```
# Count the occurrences of the substring sub in the text string txt:
def count(sub, txt)
  result = 0
  l1 = len(sub)
  l2 = len(txt)
  index = 0
  repeat l2 - l1 + 1
```

```
8       slice = txt[index:index+l1]
9       if slice == sub
10        inc(result)
11      inc(index)
12    return result
13
14  # Main program:
15  text = "The Karel programming language was named after Karel Čapek."
16  word = "Karel"
17  print("The word '" + word + "' was found", count(word, text), "times.")
```

Output:

```
The word 'Karel' was found 2 times.
```

9.14. Finding the positions of substrings

Sometimes one needs to locate the position of a text string in another text string. This means, if the substring is present, to find the index of its first character in the longer text string. If the substring is not present, the result should be `-1`. Again, this is a built-in functionality in Python which is not provided in Karel. Therefore, let's implement it! Your task is to write a function `find(sub, txt)` to return the position of (the first occurrence of) a substring `sub` in a text string `txt`. If the substring is not found, return `-1`. Try to solve this task by yourself before looking at the solution below!

Solution: In Section 9.13 (page 162) we learned how to count the occurrences of a substring in a text string. We will use it as the first thing to check if the substring `sub` is part of the text string `txt`. If not, we will return `-1`. The rest will be similar to the function `count` from Section 9.13: We will run a loop and compare the `[0:l1]`, `[1:l1+1]`, `[2:l1+2]` etc. slices of `txt` to `sub`. We know at this point that the substring is there because if it was not there, we would have returned `-1` previously. So, once a match is found, we return the starting index of the slice. Here is the corresponding program:

PROGRAM 9.7. Find the position of a substring `sub` in a text string `txt`!

```
1   # Return the position of (the first occurrence of) a substring sub
2   # in a text string txt. If the substring is not found, return -1:
3   def find(sub, txt)
4     # Check for the substring:
```

```
5     if not sub in txt
6       return -1
7     # Now we know the substring is there:
8     index = 0
9     l1 = len(sub)
10    l2 = len(txt)
11    repeat l2 - l1 + 1
12      slice = txt[index:index+l1]
13      if sub == slice
14        return index
15      inc(index)
16
17  # Main program:
18  text = "The Karel programming language was named after Karel Čapek."
19  word = "Karel"
20  result = find(word, text)
21  if result == -1
22    print("The substring was not found.")
23  else
24    print("The substring was found at position", result)
```

Output:

```
The substring was found at position 4
```

9.15. Searching and replacing in text strings

Another functionality which is provided in Python but not in Karel is searching for substrings in text strings and replacing them with other substrings. Therefore, let's implement it! Your task is to write a function `replace(txt, sub1, sub2)` which in the text string `txt` finds all occurrences of the substring `sub1`, replaces them with the substring `sub2`, and returns the result as a new text string. Try to solve the task on your own before looking at the solution below!

Solution: We will begin with creating an empty text string for the result (named, for example, `result`). This is where we will add parts of the text string `txt` which do not contain `sub1`, and where we will add `sub2` instead of `sub1`. At the beginning, we will call `find(sub1, txt)` to find the position `pos` of the first occurrence of the substring

`sub1` in `txt`. While the substring is there (`pos` is not `-1`), we will add to `result` the part of `txt` which precedes `sub1` (its `[:pos]` slice) and `sub2`. Then we remove this part from `txt` by redefining `txt` by its slice `txt[pos + len(sub1):]`. This process is repeated while the substring `sub1` is present in `txt`. At the end, we add the remaining part of `txt` to `result` and return it. Here is the corresponding code:

PROGRAM 9.8. Replace all occurrences of a substring `sub1` with another substring `sub2`!

```
# Replace all occurrences of substring sub1 with substring sub2:
def replace(txt, sub1, sub2)
  result = ''
  pos = find(sub1, txt)
  while pos != -1
    result += txt[:pos] + sub2
    txt = txt[pos + len(sub1):]
    pos = find(sub1, txt)
  return result + txt

# Main program:
text = "The Carlos programming language was named after Carlos Čapek."
result = replace(text, "Carlos", "Karel")
print(result)
```

Output:

```
The Karel programming language was named after Karel Čapek.
```

9.16. Removing substrings

Your next task is to write a function `remove(sub, txt)` to remove all occurrences of a substring `sub` from a text string `txt`. Try to solve this task by yourself before checking the solution below!

Solution: This is a simple application of the function `replace()` from Section 9.15 (page 164). It is sufficient to replace all occurrences of the substring `sub` with an empty text string `""`:

PROGRAM 9.9. Remove all occurrences of a substring from a text string!

```
# Replace all occurrences of substring sub with the empty string "":
```

```
2 def remove(sub, txt)
3   return replace(txt, sub, "")
4
5 # Main program:
6 text = "The Karel the Robot programming language was named after Karel
    Čapek."
7 result = remove("the Robot ", text)
8 print(result)
```

Output:

```
The Karel programming language was named after Karel Čapek.
```

9.17. Swapping substrings

Your next task is to write a function `swap(txt, sub1, sub2)` to swap the substrings `sub1` and `sub2` in the text string `txt`. Try to come up with a solution on your own before looking at our solution below!

Solution: This is another simple application of the function `replace()` from Section 9.15 (page 164). Let's define a helper text string which for sure will not be present in the text string `txt`. For example, we can define `help = "__NCLAB_HELPER_STRING__"`. Then, one can perform the following three steps:

(1) Replace in `txt` all occurrences of the substring `sub2` with `help`.
(2) Replace in `txt` all occurrences of the substring `sub1` with `sub2`.
(3) Replace in `txt` all occurrences of the substring `help` with `sub1`.

Here is the corresponding program:

PROGRAM 9.10. Swap two substrings in a text string!

```
1 # Swap substrings sub1 and sub2 in text string txt:
2 def swap(txt, sub1, sub2)
3   help = "__NCLAB_HELPER_STRING__"
4   txt = replace(txt, sub2, help)
5   txt = replace(txt, sub1, sub2)
6   txt = replace(txt, help, sub1)
7   return txt
8
9 # Main program:
```

```
text = "The Čapek programming language was named after Čapek Karel."
result = swap(text, "Karel", "Čapek")
print(result)
```

Output:

```
The Karel programming language was named after Karel Čapek.
```

9.18. Executing text strings as code

Executing text strings as code is another built-in functionality in Python which is not present in Karel. So let's implement it! This will be really neat – imagine sending a robot an SMS from your phone and watch him act on it! Hence, your task is to write a function `eval(txt)` to execute the text string `txt` as Karel code. There will be some limitations on how this text string may look like (these limitations help you because they make your programming simpler):

- All characters in the text string `txt` should be lowercase.
- The text string should not contain empty spaces at the ebginning or at the end.
- Individual commands should be separated with exactly one empty space `' '`.
- The text string only should contain the five basic commands `go`, `left`, `right`, `get` and `put`.

An example of such a text string `txt` is `"go get left go put right go"`.

Solution: This is a straightforward application of the `if-elif-else` statement. If the first two characters of the text string `txt` are `"go"`, we will execute the command `go`, and remove this command and the following empty space from `txt` by redefining `txt = txt[3:]`. Otherwise, if the first four characters of `txt` are `"left"` then we execute the command `left` and remove this command and the following empty space from `txt` by redefining `txt = txt[5:]`. The same we do for the remaining commands `right`, `get` and `put`. The `else` branch should be used for the case when there is a problem with the text string `txt` and none of the five basic commands is recognized. All of this should be enclosed in a `while` loop which checks whether the text string `txt` is not empty.

Here is the corresponding program:

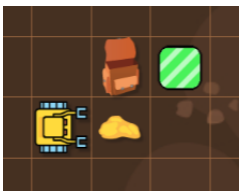
PROGRAM 9.11. Evaluate a text string as Karel code!

```
# Evaluate text string txt as Karel code.
# Only the basic commands go, left, right, get, put are accepted.
```

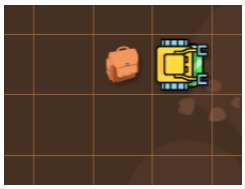
```
# All commands must be lowercase, and separated with one empty space.
# No empty spaces at the beginning or at the end!
def eval(txt)
  while txt != ""
    if txt[:2] == "go"
      go
      txt = txt[3:]
    elif txt[:4] == "left"
      left
      txt = txt[5:]
    elif txt[:5] == "right"
      right
      txt = txt[6:]
    elif txt[:3] == "get"
      get
      txt = txt[4:]
    elif txt[:3] == "put"
      put
      txt = txt[4:]
    else
      print("Unknown command, exiting.")
      return
  print("Finished!")
  return

# Main program:
txt = "go get left go put right go"
eval(txt)
```

We will test the program on a simple maze where Karel needs to make one step forward, collect a gold bugget, turn left, make another step forward, put the nugget in the bag, make a right turn, and make one more step to enter his home square. In other words, the text string will have the form `"go get left go put right go"`:



After the program finishes, the nugget is in the bag and Karel is home:



Also, the following message is displayed at the end:

```
Finished!
```

9.19. Your homework

The function `eval(txt)` from Section 9.18 (page 167) has a few technical limitations which are easy to remove:

- All characters in the text string `txt` should be lowercase.
- The text string should not contain empty spaces at the ebginning or at the end.
- Individual commands should be separated with exactly one empty space `' '`.

Remove all of them in order to make the function more robust!

Then there is one more substantial limitation:

- The text string only should contain the five basic commands `go`, `left`, `right`, `get` and `put`.

Extend the function `eval(txt)` to allow simple `repeat` loops (not nested) whose body is enclosed in two empty spaces!

9.20. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 9.1. *Which of the following are valid text strings?*

A `Hi, I am Karel!`

B `'Hi, I am Karel!'`

C `"Hi, I am Karel!"`

D `(Hi, I am Karel!)`

QUESTION 9.2. *What is the correct way to assign the text string* `"Karel"` *to a text string variable* `name`*?*

A `name := "Karel"`

B `name == "Karel"`

C `name = "Karel"`
D `"Karel" == name`

QUESTION 9.3. *What is the correct way to compare two text strings* `str1` *and* `str2`*?*
A `if str1 = str2`
B `if str1 == str2`
C `if str2 = str1`
D `if str2 == str1`

QUESTION 9.4. *How can one display the contents of a text string variable* `txt`*?*
A `print("txt")`
B `print('txt')`
C `print(txt)`
D `print[txt]`

QUESTION 9.5. *How can one concatenate two text strings* `txt1` *and* `txt2`*?*
A `txt1, txt2`
B `concat(txt1, txt2)`
C `txt1 + txt2`
D `txt1 * txt2`

QUESTION 9.6. *How can one define a new text string* `echo` *which contains 10 times the text string* `"Help!  "`*?*
A `echo = "Help!" * 10`
B `echo = "Help!  " + 10`
C `echo = "Help!  " * 10`
D `echo = "Help!  Help!  " * 5`

QUESTION 9.7. *What is the way to obtain the length of a text string* `text`*?*
A `len(text)`
B `length(text)`
C `length text`
D `text(length)`

QUESTION 9.8. *What is the correct way to parse a text string* `story` *one character at a time?*
A `for c in story`
B `for story in c`
C `in story for c`
D `repeat for c in story`

QUESTION 9.9. *What is the index of the letter* `'e'` *in Karel?*

A 2

B 3

C 4

D -2

QUESTION 9.10. *What is the slice* `txt[3:6]` *of the text string* `"Thermometer"`*?*

A `"rmo"`

B `"mom"`

C `"ome"`

D `"met"`

QUESTION 9.11. *What is the slice* `txt[-6:-3]` *of the text string* `"Thermometer"`*?*

A `"rmo"`

B `"mom"`

C `"ome"`

D `"met"`

QUESTION 9.12. *Which of the following are valid text strings in Karel?*

A `'Hi, it's me!'`

B `"Hi, it's me!"`

C `"Hi, it"s me!"`

D `'Hi, it"s me”`

QUESTION 9.13. *How can one check if text string* `txt` *contains a substring* `sub`*?*

A `if txt has sub`

B `if sub in txt`

C `if txt in sub`

D `if contains(txt, sub)`

QUESTION 9.14. *How can one swap the contents of two text strings* `str1` *and* `str2`*?*

A *Execute* `str1 = str2` *and then* `str2 = str1`

B *Create a helper variable* `help`*. Execute* `help = str2`*, then* `str2 = str1` *and then* `str2 = help`

C *Create a helper variable* `help`*. Execute* `help = str1`*, then* `str2 = str1` *and then* `str2 = help`

D *Create a helper variable* `help`*. Execute* `help = str2`*, then* `str2 = str1` *and then* `str1 = help`

10. Testing Your Programs

In this chapter you will learn:

- About the history and applications of the Morse code.
- How to write your own Morse to English translator.
- That every new command or function needs to be tested.
- How to test your new commands and functions.
- That testing your code thoroughly can save you a lot of grief.

10.1. Morse code project - Part I

The Morse code is a classical communication method which translates standard English characters to short sequences of dots and dashes. It was invented by Samuel Finley Breese Morse (1791 - 1872) with the help of his friend Alfred Vail from New York University.

In 1838, at an exhibition in New York, Morse transmitted 10 words per minute using what would forever be known as Morse code. In 1843, he received money from Congress to build a line from Baltimore to Washington DC, and, on 24 May 1844, he sent the first inter-city message: "What hath God wrought!" By 1854 there were 23,000 miles of telegraph wire in operation across the US. Morse code was later adapted to wireless radio. By the 1930s it was the preferred form of communication for aviators and seamen, and it was vital during the Second World War. BTW, here is the complete Morse code for reference:

A ·−	J ·−−−	S ···
B −···	K −·−	T −
C −·−·	L ·−··	U ··−
D −··	M −−	V ···−
E ·	N −·	W ·−−
F ··−·	O −−−	X −··−
G −−·	P ·−−·	Y −·−−
H ····	Q −−·−	Z −−··
I ··	R ·−·	

Morse code.

Today, you will teach Karel how to read the Morse code! In the maze below, each row represents one letter. The lollipops stand for dots and the candies for dashes. In other words, the message below represents `'-.-/.-/.-././.-..'` which means `'KAREL'`:



Karel's Morse code (lollipop = dot, candy = dash).

Writing a Morse code translator is a fairly complex task, therefore let's decompose it into smaller subtasks. First, Karel needs a function `row` to read one row (one Morse symbol written with lollipops and candies) and return the corresponding text string of dots and dashes. The function should assume that Karel stands in front of the symbol, facing East. The robot's final position should be the same, just in the next row down. For example, this maze contains just one letter `'K'`:



Therefore the program

```
1 print(row)
```

should bring Karel to his home square, and display the text string `'-.-'`. The corresponding program is straightforward - make sure to read all the comments!

PROGRAM 10.1. Read one Morse symbol

```
1 # Read one Morse symbol written in terms of lollipops and candies,
2 # and return it as a text string of dots and dashes:
3 def row
```

```
4     go # move on to the first object
5     morse = '' # create an empty text string
6     while lollipop or candy # scan the Morse symbol
7       if lollipop
8         morse += '.' # lollipop means dot
9       else
10        morse += '-' # candy means dash
11      go
12    repeat 2 # turn around at the end
13      right
14    n = len(morse) # figure out how many steps you made
15    repeat n+1
16      go
17    left # move one row to the South
18    go
19    left
20    return morse # return the resulting text string
21
22  # Main program:
23  print(row)
```

And indeed, when the program is executed with the above maze, Karel enters his home square and the correct symbol `'-.-'` is displayed:

```
-.-
```

Well, that's great, but it only proves that the function `row` works correctly for the letter `'K'`. What about the remaining 25 letters?

10.2. Testing the function `row`

Testing your software costs time, and sometimes it can be lengthy. But it is time very well spent. Therefore, let's stop writing new programs for a while, and prepare three mazes which will cover all 26 letters of the English alphabet:



Below is a simple testing program which will read all rows in the maze and display the corresponding Morse codes in terms of dots and dashes:

PROGRAM 10.2. Simple program to test the function `row`

```
1 while not home
2   print(row)
```

Let's run it with the three testing mazes above. And indeed, all three outputs are correct!

```
...
-
..-
...-
.--
-..-
-.--
--..
```

```
.---
-.-
.-..
--
-.
---
.--.
--.-
```

```
.-
-...
-.-.
-..
.
..-.
--.
....
..
```

Software companies, depending on their size, use software testers or entire testing departments. There is a famous quote which says *"The longer you wait to fix a problem, the more expensive and troublesome it will be to fix."* Therefore, the best option is to prevent problems from happening in the first place. And we just did that - we made sure that there won't be any problems with our function `row`. This is golden.

If your program includes custom commands or functions,
it is important that you test each of them thoroughly.

10.3. Morse code project - Part II

Awesome! Now we have a function `row` which turns Karel maze symbols into regular Morse code text strings. Next, we need a function to translate Morse text strings to English letters. This is straightforward, although lengthy because we have to account for all 26 English characters. The following program includes the function as well as a main program which should display the letter `'K'`:

PROGRAM 10.3. Translate individual Morse symbols to English

```
# Translate one Morse symbol to an English letter:
def english(symbol)
  if symbol == '.-'
    return 'A'
  elif symbol == '-...'
    return 'B'
  elif symbol == '-.-.'
    return 'C'
  elif symbol == '-..'
    return 'D'
  elif symbol == '.'
    return 'E'
  elif symbol == '..-.'
    return 'F'
  elif symbol == '--.'
    return 'G'
  elif symbol == '....'
    return 'H'
  elif symbol == '..'
    return 'I'
  elif symbol == '.---'
    return 'J'
  elif symbol == '-.-'
    return 'K'
  elif symbol == '.-..'
    return 'L'
```

```
27    elif symbol == '--'
28      return 'M'
29    elif symbol == '-.'
30      return 'N'
31    elif symbol == '---'
32      return 'O'
33    elif symbol == '.--.'
34      return 'P'
35    elif symbol == '--.-'
36      return 'Q'
37    elif symbol == '.-.'
38      return 'R'
39    elif symbol == '...'
40      return 'S'
41    elif symbol == '-'
42      return 'T'
43    elif symbol == '..-'
44      return 'U'
45    elif symbol == '...-'
46      return 'V'
47    elif symbol == '.--'
48      return 'W'
49    elif symbol == '-..-'
50      return 'X'
51    elif symbol == '-.--'
52      return 'Y'
53    elif symbol == '--..'
54      return 'Z'
55    else
56      print("Wrong Morse symbol detected in english() - returning '*'.")
57      return '*'
58
59  # Main program:
60  s = '-.-'
61  print(english(s))
```

Notice that if the function encounters an unknown symbol, it will return `'*'`. This `'*'` will then appear in the resulting English text string, so it will be easy to spot. OK, let's execute the program now!

```
K
```

This is the expected output - great! But what about the remaining 25 letters?

10.4. Testing the function `english(symbol)`

It is important to call the function with all 26 Morse symbols `'.-'`, `'-...'`, `'-.-.'`, `...`, `'--..'` and verify that indeed we obtain the corresponding English letters `'A'`, `'B'`,`'C'`,`...`,`'Z'`:

PROGRAM 10.4. Program to test the function `english()`

```
# Testing the function english():
r = ''
s = '.-'
r += english(s)
s = '-...'
r += english(s)
s = '-.-.'
r += english(s)
s = '-..'
r += english(s)
s = '.'
r += english(s)
s = '..-.'
r += english(s)
s = '--.'
r += english(s)
s = '....'
r += english(s)
s = '..'
r += english(s)
s = '.---'
r += english(s)
s = '-.-'
r += english(s)
s = '.-..'
r += english(s)
s = '--'
```

```
28 r += english(s)
29 s = '-.'
30 r += english(s)
31 s = '---'
32 r += english(s)
33 s = '.--.'
34 r += english(s)
35 s = '--.-'
36 r += english(s)
37 s = '.-.'
38 r += english(s)
39 s = '...'
40 r += english(s)
41 s = '-'
42 r += english(s)
43 s = '..-'
44 r += english(s)
45 s = '...-'
46 r += english(s)
47 s = '.--'
48 r += english(s)
49 s = '-..-'
50 r += english(s)
51 s = '-.--'
52 r += english(s)
53 s = '--..'
54 r += english(s)
55 print(r)
```

And here is the corresponding output,

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

which shows that the function works correctly.

Always make sure to test each new command and function.
Test them individually, separated from the rest of the code.

10.5. Morse code project - Part III

Super, we are ready to finish! With the heavy lifting done by the functions `row` and `english (symbol)`, the main function `morse2english` is very easy to write:

PROGRAM 10.5. Putting all parts together

```
# Read Morse symbols in Karel maze, and translate them to English text:
def morse2english
  text = ''
  while not home
    text += english(row)
  return text

# Main program:
print(morse2english)
```

The final program will comprise the functions `row`, `english(symbol)`, and the last code above. Now let's run it with the original maze,



Output:

```
KAREL
```

Indeed, this result is correct! The function `morse2english` is so simple that it does not require testing. Importantly, it works for the sample maze above, and its two essential parts, the functions `row` and `english(symbol)`, were thoroughly tested.

10.6. Your homework

Here is something for you to think about: The function `row` does not consider empty spaces between words. Would you be able to adjust it to return an empty space `' '` if it encounters an empty row in the maze?

And as a second task - implement your own English to Morse translator! It's OK to stay on the text level, you don't need to create a graphical representation of the Morse symbols in the maze.

10.7. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 10.1. *Is it mandatory that software developers test their software?*

A Yes.

B No.

QUESTION 10.2. *Is it recommended that software developers test their software?*

A Yes.

B No.

QUESTION 10.3. *What happens if a problem in the software is not fixed for a long time?*

A It disappears by itself.

B Everybody forgets about it.

C It will be very expensive and time consuming to fix.

D It may cause users to get upset and lose interest in the software.

QUESTION 10.4. *Is software testing supposed to be a quick and easy process?*

A Yes, otherwise nobody would be doing it.

B Yes, because the testing does not have to be thorough.

C Yes, because someone else will do it.

D No, testing software can take a lot of time.

QUESTION 10.5. *Should each new command or function be isolated from the rest of the code for testing?*

A No, that would just be more work.

B No, testing a new command or function within the original code is better.

C No, because the new command or function might not work when put back.

D Yes, to avoid the influence of potential bugs elsewhere in the code.

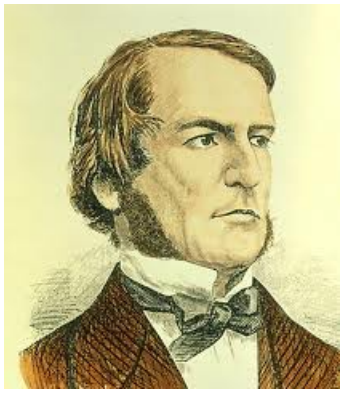
11. Boolean Values, Variables, Expressions, and Functions

In this chapter you will learn:

- Who was George Boole.
- What is the so-called *Boolean (logical) algebra*.
- About Boolean values `True`, `False` and Boolean variables.
- That the comparison operators ==, `!=`, <, >, <= and >= yield `True` or `False`.
- How to work with Karel's GPS coordinates.
- How to evaluate complex Boolean expressions.
- That Karel's sensors are Boolean functions
- That the `if` statement and the `while` loop both expect Boolean values.
- How to create an infinite loop and when such a loop can be useful.

11.1. George Boole

George Boole was an English mathematician and philosopher. During his era, logic was considered to be part of philosophy rather than mathematics. One of his greatest contributions is the mathematical formalization of logic which he introduced in his famous 1854 book The Laws of Thought. There he explained that logical processes and conclusions can be written using mathematical symbols and formulas, and he introduced his legendary *Boolean algebra* which long after his death became a fundamental pillar of modern computer science.



George Boole (1815 - 1864).

11.2. Quick introduction to logic

You certainly know the Harry Potter movies whose main stars are three young characters Harry Potter, Hermione Granger and Ron Weasley. One can state various things about them which are either true or false. For example, "Harry was born in the Suffolk village of Lavenham" is true. Actually, let's use `True` instead of just "true" to emphasize that the evaluation of the statement resulted into a logical value. Analogously, evaluating the statement "Hermione Granger is a boy" yields `False`.

To simplify all this, let's denote the statement "Harry was born in Lavenham" by `A` and the statement "Hermione Granger is a boy" by `B`. Then `A` is `True` and `B` is `False`.

From Chapter 4 you already know the logical operations `and`, `or`, `not`. The operation `not` flips `True` to `False` and vice versa. For example, saying "Harry was born in Lavenham" `and` "Hermione Granger is `not` a boy" is `True`. This can be abbreviated as

```
A and (not B) is True
```

Saying "Harry was `not` born in Lavenham" `and` "Hermione Granger is `not` a boy" is `False`. In short, this is the same as

```
(not A) and (not B) is False
```

It is also `False` to say that "Harry was born in Lavenham" `and` "Hermione Granger is a boy":

```
(not A) and B is False
```

Finally, also saying that "Harry was `not` born in Lavenham" `and` "Hermione Granger is a boy" is `False`:

```
A and B is False
```

Since `A` is `True` and `B` is `False`, the above four statements can be summarized in a table:

```
True and True is True
False and True is False
True and False is False
False and False is False
```

This table is called the the *truth table* of the `and` operator.

Now imagine that you like to eat pizza. But not all kinds - you will only eat a pizza if it has cheese or salami on it. So, when we abbreviate "has cheese" by `A` and "has salami" by `B`, you will eat a pizza only if `A` is `True` or if `B` is `True`:

```
A or B is True
```

If they bring you a pizza which only has cheese (`A` is `True`) but not salami (`B` is `False`), you will still eat it:

```
A or (not B) is True
```

And you will also eat a pizza which does not have cheese (`A` is `False`) but has salami (`B` is `True`):

```
(not A) or B is True
```

However, when a pizza has neither (both `A` and `B` are `False`), you will not eat it:

```
(not A) or (not B) is False
```

These four outcomes can be summarized in another important table which is called the *truth table* of the `or` operator:

```
True or True is True
False or True is True
True or False is True
False or False is False
```

11.3. What is *Boolean algebra?*

Looking at the truth tables above, G. Boole realized that if `True` was replaced by a nonzero number (let's just say "nonzero") and `False` by 0, then the first table (of the `and` operator) would look just like multiplication:

$$\begin{aligned} \text{nonzero} \cdot \text{nonzero} &= \text{nonzero} \\ 0 \cdot \text{nonzero} &= 0 \\ \text{nonzero} \cdot 0 &= 0 \\ 0 \cdot 0 &= 0 \end{aligned}$$

Since then, the `and` operation has been called *logical product*. And similarly, the truth table of the `or` operator looks just like addition, and therefore has been called *logical sum*:

$$\begin{aligned} \text{nonzero} + \text{nonzero} &= \text{nonzero} \\ 0 + \text{nonzero} &= \text{nonzero} \\ \text{nonzero} + 0 &= \text{nonzero} \\ 0 + 0 &= 0 \end{aligned}$$

You may have noticed that the first line has a minor imperfection, because the sum of two nonzero numbers can be zero (for example when adding -1 and 1). But this imperfection goes away when we limit ourselves from all nonzeros to only positive nonzeros.

11.4. Boolean values and variables in Karel

Karel provides the keywords `True` and `False`. For example, typing

```
1 print(True)
```

will display

```
True
```

And when typing

```
1 print(True)
```

one will obtain

```
False
```

The operations `and`, `or`, `not` can be applied to the Boolean values `True` and `False` directly. For illustration, this short program will reproduce the truth table of the `and` operator:

```
1 print(True, " and", True, " is", True and True)
2 print(False, "and", True, " is", False and True)
3 print(True, " and", False, "is", True and False)
4 print(False, "and", False, "is", False and False)
```

Output:

```
True  and True  is True
False and True  is False
True  and False is False
False and False is False
```

This program will reproduce the truth table of the `or` operator:

```
print(True, " or", True, " is", True or True)
print(False, "or", True, " is", False or True)
print(True, " or", False, "is", True or False)
print(False, "or", False, "is", False or False)
```

Output:

```
True  or True  is True
False or True  is True
True  or False is True
False or False is False
```

The `True` and `False` values can be assigned to variables. Such variables then become *Boolean variables*. For example,

```
b1 = True
print(b1)
```

will display

```
True
```

and

```
b2 = False
print(b2)
```

will yield

```
False
```

The logical operations `and`, `or` can be applied to the Boolean variables. As a last example, let's reproduce the truth table of the `and` operator again:

```
t = True
f = False
print(t and t)
print(f and t)
print(t and f)
print(f and f)
```

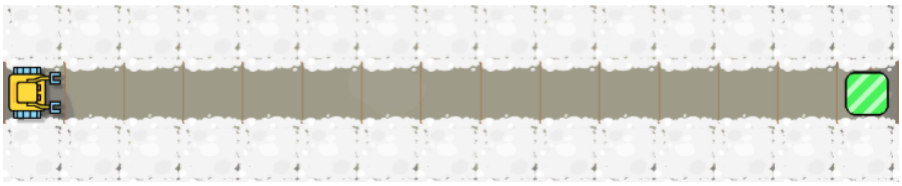
Output:

```
True  and True  is True
False and True  is False
True  and False is False
False and False is False
```

11.5. GPS sensors gpsx and gpsy

Karel has his own GPS device that reveals his GPS coordinates in the maze! The device comes in the form of two built-in functions gpsx and gpsy which return the current column and row the robot is in, respectively. Importantly, when Karel is in the left-most column, his gpsx is zero. When he is in the bottom row, his gpsy is zero.

For illustration, let's have Karel traverse the entire maze from West to East and display his gpsx coordinate in each grid square:



Here is the program the robot will use:

```
while not home
  print(gpsx)
  go
print(gpsx)
```

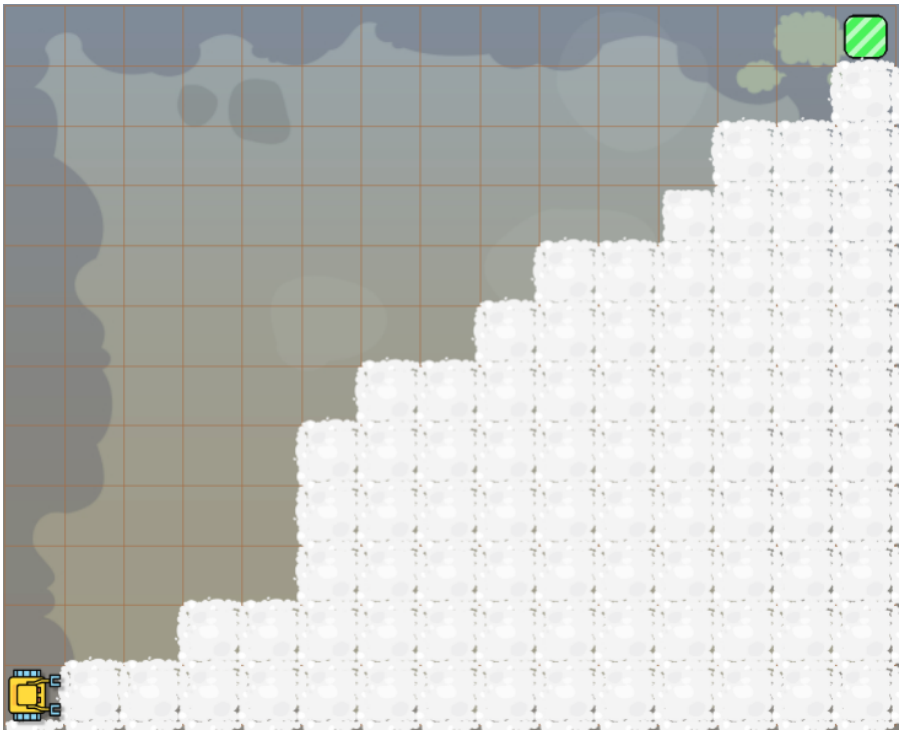
The functions gpsx and gpsy return Karel's row and column number, respectively.

Output:

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

In the output you can see that the `gpsx` coordinate of the right-most column (where the home square is) is 14.

Next, Karel will ascend a snowy mountain, reporting both his `gpsx` and `gpsy` coordinates in every grid square he visits:



This is the corresponding program:

```
while not home
  while wall
    print(gpsx, gpsy)
    left
    go
    right
  print(gpsx, gpsy)
  go
print(gpsx, gpsy)
```

Output:

```
0 0
0 1
1 1
2 1
2 2
3 2
4 2
4 3
4 4
4 5
5 5
5 6
6 6
7 6
7 7
8 7
8 8
9 8
10 8
10 9
11 9
11 10
12 10
13 10
13 11
14 11
```

In the output you can see that the gpsy coordinate of the top row of the maze is 11.

11.6. Comparison operators ==, !=, <, >, <= and >=

You first saw these operators in Chapter 7, but at that time we did not know yet that they returned Boolean values `True` or `False`. That's what they do! For illustration, look at this sample program which displays `True`:

```
1 print(1 < 2)
```

Also, look at another sample program whose output is `False`:

```
1 print(1 == 2)
```

Of course, Boolean values returned by the comparison operators can be stored in Boolean variables. This program displays `True`:

```
1 n = 5
2 b = n == 5
3 print(b)
```

Using parentheses will improve the readability of the previous program:

```
1 n = 5
2 b = (n == 5)
3 print(b)
```

And here is one last, slightly more complex example whose output is `False`:

```
1 done = False
2 n = 5
3 m = 9
4 b = (n == 5) and (m < 10) and done
5 print(b)
```

> The result of comparisons such as ==, !=, <, >, <= and >=
> is either `True` or `False`.

11.7. Boolean functions

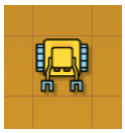
You have already seen quite a few functions which returned numbers, as well as one function which returned text strings (function `row` in the Morse translator project on page

173). Boolean values `True` and `False` can normally be returned by functions as well. A function which returns a Boolean value is called *Boolean function*.

As you know, Karel has a sensor `north` which he can use to check whether he faces North. He does not have similar sensors for South, East or West. Therefore, let's define a Boolean function `south` which returns `True` if Karel faces South and `False` otherwise:

```
1 def south
2   repeat 2
3     left
4   n = north
5   repeat 2
6     right
7   return n
```

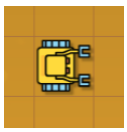
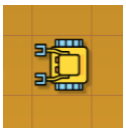
When Karel faces South,



the function returns

```
True
```

When he faces East, West or North,


it returns

```
False
```

As another example, let's write a function `border` which returns `True` if Karel stands next to the border of the maze, and `False` when he stands in the interior:

```
1 def border
2   return (gpsx == 0) or (gpsx == 14) or (gpsy == 0) or (gpsy == 11)
```

As a last example, let's write a Boolean function `edible` which returns `True` if Karel stands above an edible object, and `False` otherwise:

```
def edible
  return pumpkin or corn or apple or cherry or candy or lollipop or pie or
      orange or banana or potato or coconut or popcorn
```

In Karel, one can define Boolean functions which return `True` or `False`.

11.8. Karel's sensors are Boolean functions

All Karel's sensors for obstacles, collectible objects, and containers are Boolean functions. So are the other sensorse such as `north`, `home` and `empty`. The only exceptions are the `gpsx` and `gpsy` sensors which return integer numbers.

To illustrate this, let's put Karel one step away from a wall, have him display the value of the `wall` sensor, then have him approach the wall, and display the value of the sensor again:



Here is the corresponding program:

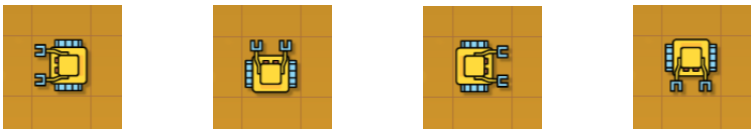
```
print(wall)
go
print(wall)
```

Karel's sensors are Boolean functions which return either `True` or `False`.

Output:

```
False
True
```

In the following example, Karel faces West. He will make four right turns, displaying the value of the `north` sensor before every turn:



The corresponding program,

```
1 repeat 4
2   print(north)
3   right
```

will output

```
False
True
False
False
```

11.9. The `if` statement revisited

You have already seen the `if` statement in many different forms. It can be used to avoid crashing into a wall in front of the robot:

```
1 if wall
2   left
```

Or it can be used to make sure that Karel can move forward safely:

```
1 if not wall
2   go
```

One can use it to check whether Karel has any objects in his bag:

```
1 if not empty
2   put
```

Or, one can use it to detect whether Karel is facing North:

```
1 if north
2   left
```

After seeing in the previous section that Karel's sensors are in fact Boolean functions, you understand that the `if` statement expects a Boolean value. That's how it decides whether

its body will be executed or not.

The Boolean values `True` and `False` can be used with the `if` statement directly:

```
if True
  print("The body of this if statement will be executed always.")
```

Although this seems to be of little practical application, it can be used to disable a condition and just let the code run. Or, the `False` value can be used to disable the body of the condition so that it never runs:

```
if False
  print("The body of this if statement will never be executed.")
```

Especially for longer codes this is less work than having to comment out many lines.

The `if` statement expects either `True` or `False`.

11.10. Using the `if` statement to display debugging information

Displaying the values of important variables while the program is running is the oldest and most universal debugging technique. One can define one Boolean variable (named for example `DEBUG`) at the beginning of the code, and then use it in `if` statements throughout the program to enable debugging output:

```
DEBUG = True # debug mode ON

...

if DEBUG
  print("Debug info X =", X)

...

if DEBUG
  print("Debug info n =", n)

...
```

When the DEBUG variable is set to False, no debugging information will be displayed:

```
DEBUG = False # debug mode OFF

...

if DEBUG
  print("Debug info X =", X)

...

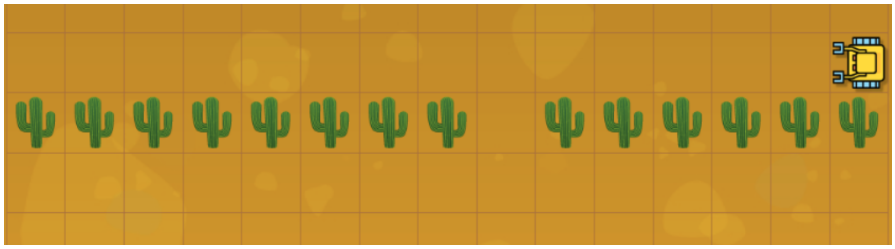
if DEBUG
  print("Debug info n =", n)

...
```

Control prints are the oldest and most universal debugging technique.

11.11. Another look at the while loop

After reading Section 11.9 (page 193) it will come as no surprise that the while statement expects a Boolean value. When True, the body of the loop is executed. When False, the loop ends. To illustrate this, let's have Karel look for a place to cross a cactus field, displaying the value of the cactus sensor on the way:

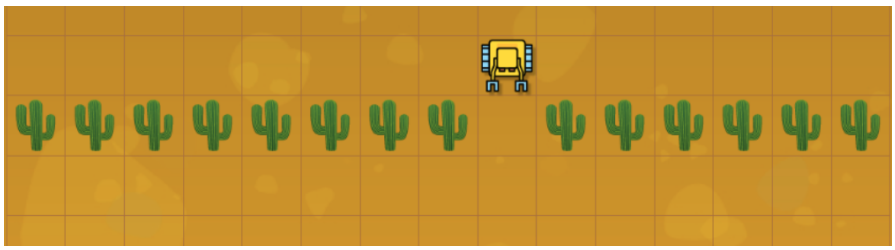


This is the corresponding program:

```
left
while cactus
  print(cactus)
  right
  go
```

```
    left
print(cactus)
```

After the program ends, Karel stands in front of the opening:



And here is the program output:

```
True
True
True
True
True
True
False
```

The `while` statement expects either `True` or `False`.

11.12. Infinite loop `while True`

By typing `while True` one can create an infinite loop. Infinite loops are used in Python and other programming languages, for instance to wait for user input or some other type of event, to simplify the code when the condition behind the keyword `while` is too complex, or to emulate the `do-while` loop.

We can illustrate the process of waiting for an event by tossing dice. Karel has a function `randint(m, n)` which returns a random integer between `m` and `n` (including `m` and `n`). So, calling `randint(1, 6)` is the same as tossing a die. Imagine that Karel is playing a game and must toss a dice until he gets a six.

This can be done using the regular `while` loop. But the code is cumbersome because the line with the `randint` function, as well as the line that displays the value, must be there twice:

```
# Karel must toss dice until he gets 6:
def game
```

```
3     n = randint(1, 6)
4     while n != 6
5       print(n)
6       n = randint(1, 6)
7     print(n)
8     return
9
10  # Main program:
11  game
```

Here is an alternative code which uses an infinite loop `while True` to emulate the `do-while` loop. It only needs to use the function `randint` and the `print` statement once:

```
1   # Karel must toss dice until he gets 6:
2   def game
3     while True
4       n = randint(1, 6)
5       print(n)
6       if n == 6 # event occurred, exit loop
7         return
8
9   # Main program:
10  game
```

Sample output:

```
1
2
4
5
3
4
5
3
2
6
```

More about randomness and probability will be said in Chapter 12. There we will even show you that randomness can be used to solve difficult tasks which cannot be solved deterministically.

11.13. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 11.1. *Who was George Boole?*

A *Australian musician who composed the song "True or False".*

B *Host of the movie show "Truth or Consequences".*

C *Russian programmer who created the video game "True or False"*

D *English mathematician who studied logic.*

QUESTION 11.2. *What are the two values used in Boolean algebra?*

A `Yes` *and* `No`

B `True` *and* `False`

C `True` *and* `Untrue`

D `Truth` *and* `Lie`

QUESTION 11.3. `A` *is* `True`, `B` *is* `False`. *What is* `A and B`*?*

A `True`

B `False`

QUESTION 11.4. `A` *is* `True`, `B` *is* `False`. *What is* `A or B`*?*

A `True`

B `False`

QUESTION 11.5. `A` *is* `True`, `B` *is* `False`. *What is* `(not A) and B`*?*

A `True`

B `False`

QUESTION 11.6. `A` *is* `True`, `B` *is* `False`. *What is* `(not A) or (not B)`*?*

A `True`

B `False`

QUESTION 11.7. `a` *is* `1`, `b` *is* `2`. *What will the code* `print(a == b)` *display?*

A `True`

B `False`

QUESTION 11.8. `a` *is* `4`, `b` *is* `0`. *What will the code* `print(a != b)` *display?*

A `True`

B `False`

QUESTION 11.9. *Karel is in the right-most column of the maze. What will be the output of the code* `print(gpsx)`*?*

A 0
B 1
C 11
D 14

QUESTION 11.10. *Karel is in the left-most column of the maze. What will be the output of the code* `print(gpsx)`*?*

A 0
B 1
C 11
D 14

QUESTION 11.11. *Karel is in the second row from the top of the maze. What will be the output of the code* `print(gpsy)`*?*

A 9
B 10
C 11
D 12

QUESTION 11.12. *Karel is in the third row from the bottom of the maze. What will be the output of the code* `print(gpsy)`*?*

A 0
B 1
C 2
D 3

QUESTION 11.13. *What will the code* `print(north)` *display?*

A The word `north`*.*
B It will generate an error message.
C It will display `True` *if Karel faces North.*
D It will display `False` *if Karel does not face North.*

QUESTION 11.14. *What will the code* `print(home)` *display?*

A The word `home`*.*
B It will generate an error message.
C It will display `True` *if Karel is at his home square.*
D It will display `False` *if Karel is away from his home square.*

12. Randomness and Probability

In this chapter you will learn:

- Why is randomness useful in computing.
- How to generate random integers using the function `randint`.
- How to use the function `randint` to simulate rolling dice.
- How to calculate the minimum and maximum of a given set of numbers.
- How to generate random Booleans using the function `rand`.
- How to use the function `rand` to simulate coin toss.
- How to calculate simple probabilities.
- How to use randomness to solve difficult tasks.

12.1. Why is randomness useful in computing

Random numbers are used for many applications starting with gambling and creating unpredictable results in computer games. But one of the major applications of randomness is in *cryptography*. Cryptography is used to secure data for safe transfer via Internet, to secure data storage, and other tasks. Creating a reliable data encryption requires numbers that nobody can guess. Think about it - if a deterministic (non-random) algorithm was used to encrypt your data, then at least the author of the algorithm would always be able to break it, read your emails, enter into your bank account, or steal your identity.

Randomness is also very important for scientific and engineering simulations. For example, imagine that an engineer computes the strength of a bridge. He or she knows that all the physical properties of the underlying materials such as the quality of the concrete and of the steel come with some amount of uncertainty. Therefore, many computations with varying parameters must be done to ensure that this uncertainty is taken into account. These methods are called Monte Carlo Methods.

Generating random numbers and using randomness
is an important part of computing.

12.2. Generating random integers

Karel can generate random integers using the function `randint` which was already mentioned in Section 11.12 (page 196). In summary, calling `randint(m, n)` returns a random integer between `m` and `n` (including `m` and `n`). Calling the function with just one argument as in `randint(n)` is the same as calling `randint(1, n)`. Let's see some examples.

First, let's use a `repeat` loop to generate and display 10 random integers between -1 and 1:

```
repeat 10
  print(randint(-1, 1))
```

Output:

```
0
0
1
1
1
-1
0
1
1
0
```

The following program generates 10 random integers between 1 and 100:

```
repeat 10
  print(randint(100))
```

Output:

```
5
14
57
90
81
33
92
27
85
35
```

12.3. Karel is rolling a die

When rolling a die, one obtains a random integer between 1 and 6. The same can be done by calling `randint(1, 6)` or just `randint(6)`. Today, Karel is curious how many attempts it will take him to throw a six. Here is the corresponding program:

```
1 n = 0
2 die = -1
3 while die != 6
4   die = randint(6)
5   print(die)
6   inc(n)
7 print("Attempts needed:", n)
```

Output:

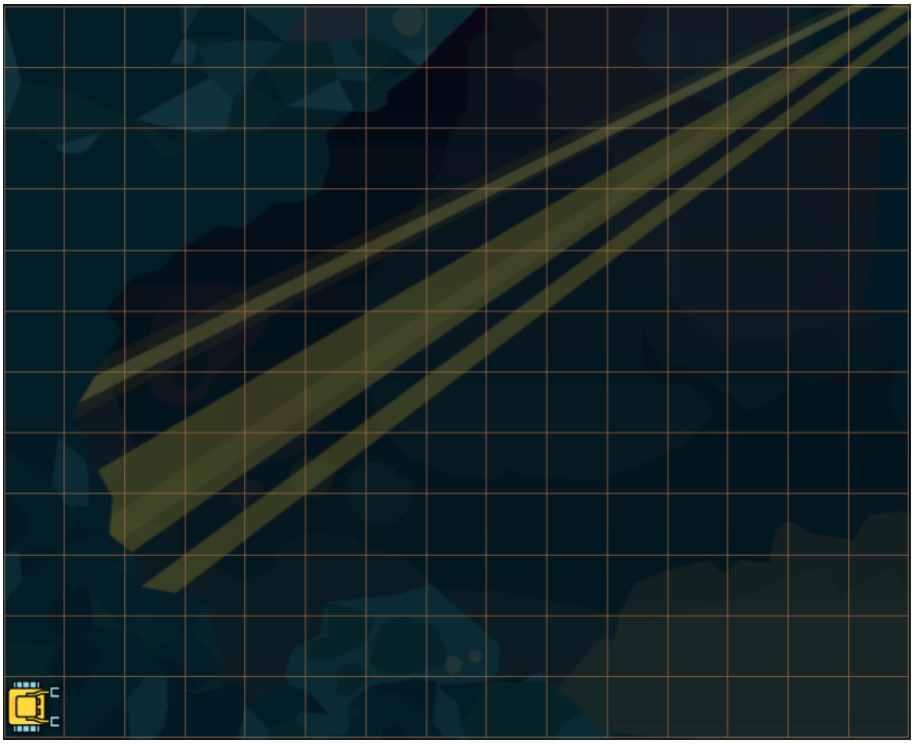
```
6
Attempts needed: 1
```

Wow, that was quick! Beginner's luck, probably. Let's try this one more time!

```
1
5
1
1
5
4
5
3
1
5
5
4
1
5
5
4
3
3
3
4
3
4
6
Attempts needed: 23
```

12.4. Placing objects at random locations

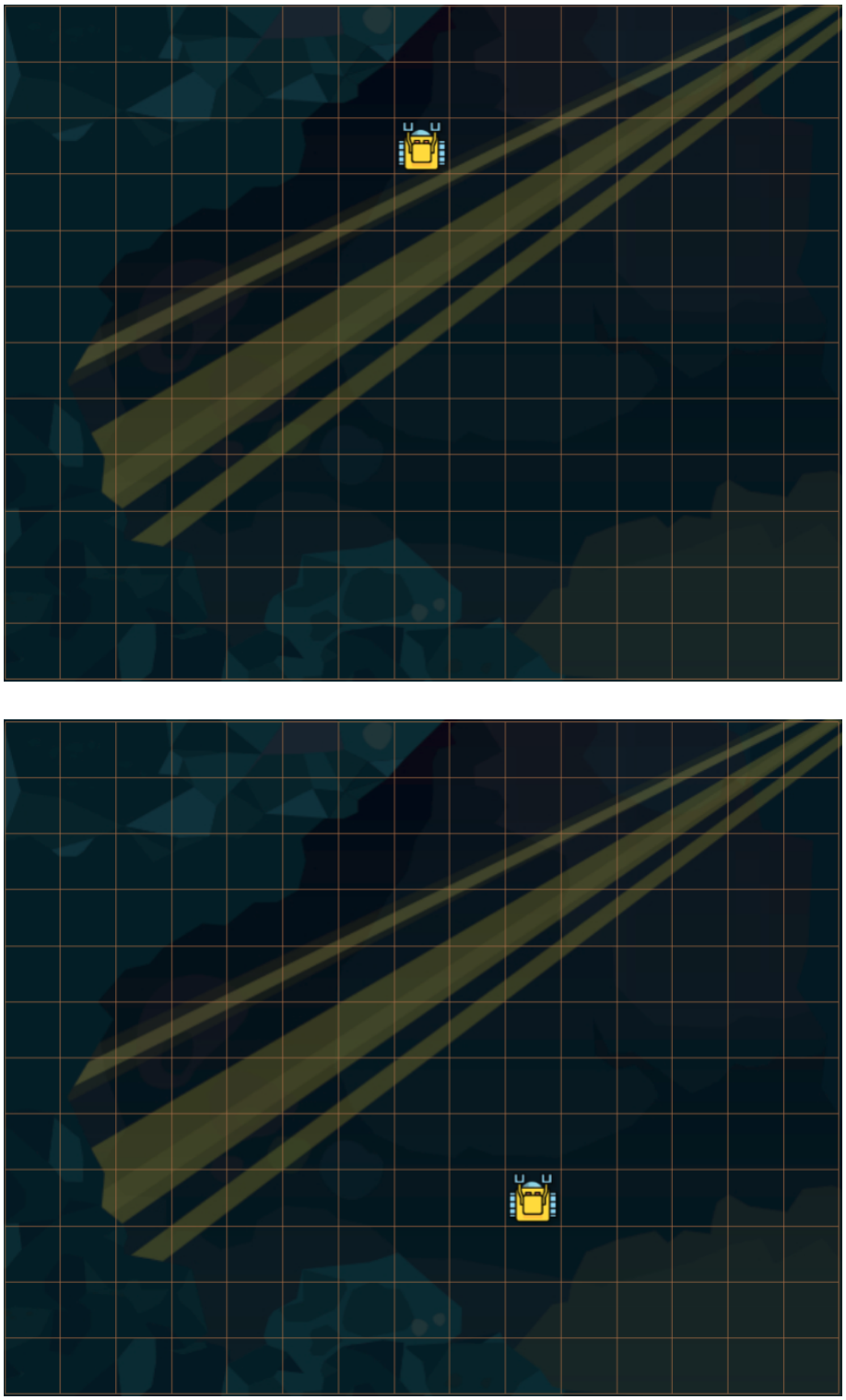
Karel stands in the Southwest corner of the maze, facing East:



He carries one light bulb in his backpack, and his task is to place it at a random location in the maze. The maze is 15 squares wide and 12 squares tall. Here is a program which will do it:

```
x = randint(0, 14) # number of steps to make in the East direction
y = randint(0, 11) # number of steps to make in the North direction
repeat x
  go
left
repeat y
  go
put
```

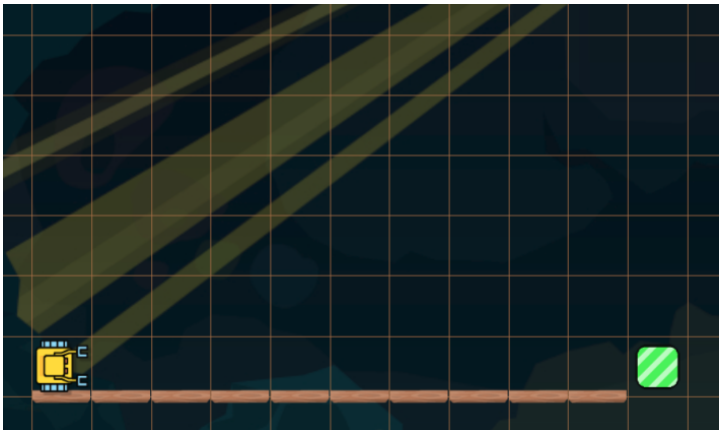
Let's execute the program two times. Keep in mind that the light bulb will not be visible because it will be beneath Karel at the end:



Karel's final positions after executing the last program two times.

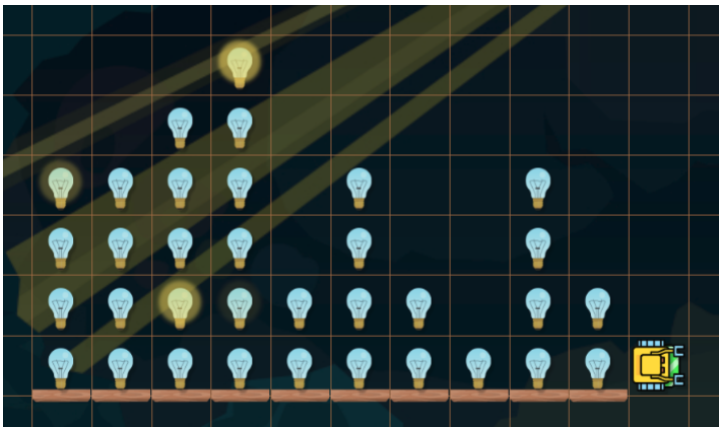
12.5. Building a random skyline

Today, Karel has 60 light bulbs in his bag, and his task is to use them to build a skyline.



Karel's new task is to build a skyline.

The skyline will consists of 10 columns of bulbs of random heights between one and six, such as this one:



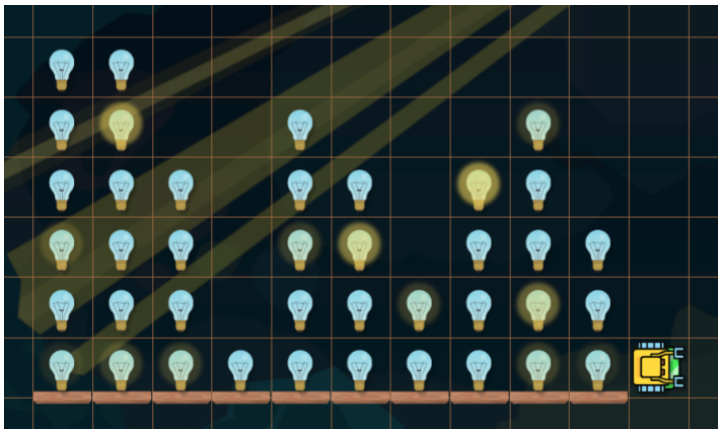
Random skyline.

This is the corresponding program:

```
repeat 10
  # Generate a random integer between 1 and 6:
  n = randint(1, 6)
  # Build one column:
  left
  repeat n
    put
```

```
8     go
9   # Turn around:
10  right
11  right
12  # Return back to the bottom:
13  while not wall
14    go
15  # Get ready for the next column:
16  left
17  go
```

Let's execute the program once more to show that the robot will create a different skyline:



Executing the program one more time.

12.6. Calculating the maximum

One important task in computer science is to calculate the maximum of a given sequence of values. Let's take a look at a sample sequence 3, 4, 2.

As the first step, create a new variable named (for example) `maximum`, and initialize it with some value which is *less than the lowest value* in the sequence:

```
1 maximum = 0
```

If you do not know how to safely choose a value which is below all values in the sequence, it is possible to take any value from the sequence. Usually it is easiest to take the first one:

```
1 maximum = 3
```

We will take, for example, the initial value `maximum = 0`. Here is the algorithm:

Step 1:

- The current value of `maximum` is 0.
- Take the next value in the sequence: `m = 3`
- Is `m` greater than `maximum`? (`m` is 3, `maximum` is 0)
- Yes, therefore `m` is the new maximum: `maximum = m`
- After this step, the value of `maximum` is at 3.

Step 2:

- The current value of `maximum` is 3.
- Take the next value in the sequence: `m = 4`
- Is `m` greater than `maximum`? (`m` is 4, `maximum` is 3)
- Yes, therefore `m` is the new maximum: `maximum = m`
- After this step, the value of `maximum` is at 4.

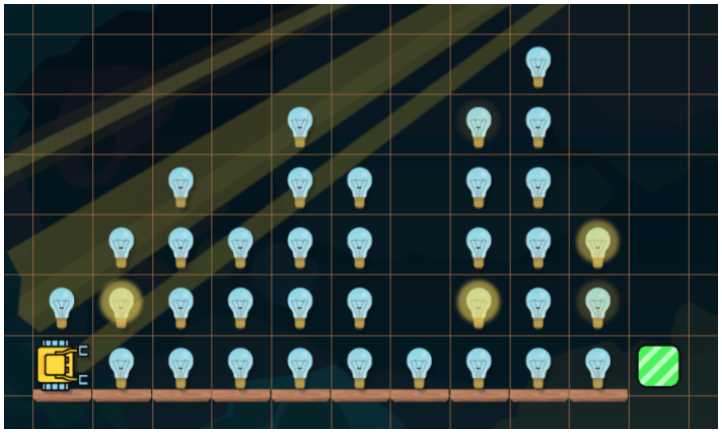
Step 3:

- The current value of `maximum` is 4.
- Take the next value in the sequence: `m = 2`
- Is `m` greater than `maximum`? (`m` is 2, `maximum` is 4)
- No, therefore `maximum` stays unchanged.
- After this step, the value of `maximum` is at 4.

There are as many steps as there are values in the sequence. Now let's apply this algorithm to measure the height of a random skyline!

12.7. Measuring the height of a skyline

Karel's next task is to measure the height of a random skyline which is formed by 10 columns of light bulbs. The height of the skyline is the height of the tallest column.



Karel is measuring the height of a random skyline.

To solve this task, the robot will employ the algorithm from the previous section. We will initialize `maximum` with 0, and measure the height `m` of each column. If `m > maximum`, he will update `maximum` to `m`. Otherwise he will leave `maximum` unchanged. Here is the corresponding program, enhanced with control prints to reveal in more detail what is happening:

```
# Initialize the maximum with 0:
maximum = 0
# Loop over all columns:
repeat 10
  # Measure the height of the next column:
  m = 0
  left
  while bulb
    inc(m)
    go
  # Turn around:
  right
  right
  # Return back to the bottom:
  while not wall
    go
  # Get ready for the next column:
  left
  go
  # If m > maximum, update maximum to m:
  if m > maximum
    print(m, ">", maximum, "therefore updating maximum to", m)
    maximum = m
  else
    print(m, "<=", maximum, "therefore maximum not updated")
# Display maximum at the end:
print("Height of the skyline is", maximum)
```

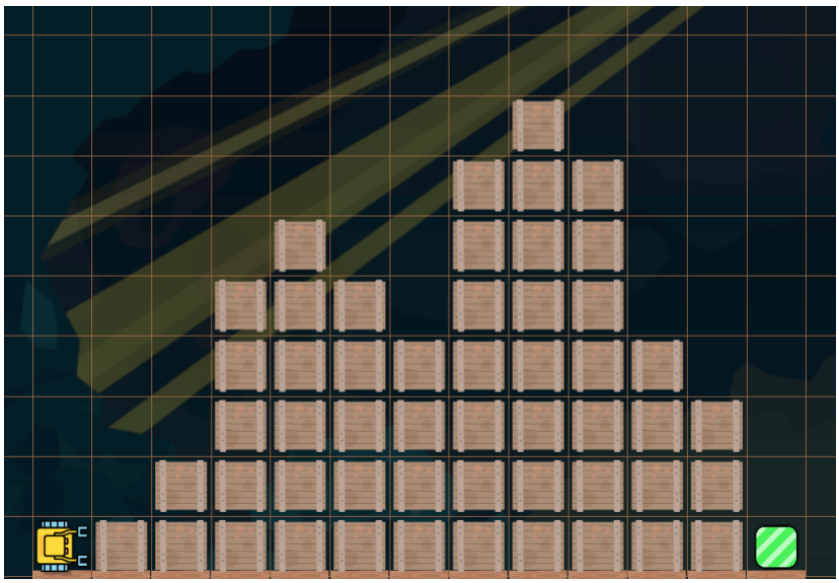
In every step compare the new value `m` with the current maximum.
If `m` is greater than the maximum, increase the maximum to `m`.

For the skyline shown above, the program will produce the following output:

```
2 > 0 therefore updating maximum to 2
3 > 2 therefore updating maximum to 3
4 > 3 therefore updating maximum to 4
3 <= 4 therefore maximum not updated
5 > 4 therefore updating maximum to 5
4 <= 5 therefore maximum not updated
1 <= 5 therefore maximum not updated
5 <= 5 therefore maximum not updated
6 > 5 therefore updating maximum to 6
3 <= 6 therefore maximum not updated
Height of the skyline is 6
```

12.8. Measuring the height of a building

Today Karel needs to measure the height of an unknown building. The building is formed by columns of crates.



Karel is measuring the height of a building.

This task is different from the previous one because Karel cannot go inside the building (crates are obstacles). Instead, he will use the Second Maze Algorithm (right-handed version) to climb on the crates. He will initialize `maximum` with 0. After each step along the crates he will measure his `gpsy` coordinate `m`. Note that `gpsy` is the height of the column below Karel. If `m` is greater than `maximum`, then he will update `maximum` with `m`. This is the corresponding program:

```
1 maximum = 0
2 # Use the Second Maze Algorithm:
```

```
while not home
  right
  while wall or crate
    left
  go
  m = gpsy
  if m > maximum
    print(m, ">", maximum, "therefore updating maximum to", m)
    maximum = m
  else
    print(m, "<=", maximum, "therefore maximum not updated")
print("Height of the building is", maximum)
```

And here is the output:

```
1 > 0 therefore updating maximum to 1
1 <= 1 therefore maximum not updated
2 > 1 therefore updating maximum to 2
2 <= 2 therefore maximum not updated
3 > 2 therefore updating maximum to 3
4 > 3 therefore updating maximum to 4
5 > 4 therefore updating maximum to 5
5 <= 5 therefore maximum not updated
6 > 5 therefore updating maximum to 6
6 <= 6 therefore maximum not updated
6 <= 6 therefore maximum not updated
5 <= 6 therefore maximum not updated
5 <= 6 therefore maximum not updated
4 <= 6 therefore maximum not updated
5 <= 6 therefore maximum not updated
6 <= 6 therefore maximum not updated
7 > 6 therefore updating maximum to 7
7 <= 7 therefore maximum not updated
8 > 7 therefore updating maximum to 8
8 <= 8 therefore maximum not updated
8 <= 8 therefore maximum not updated
7 <= 8 therefore maximum not updated
7 <= 8 therefore maximum not updated
6 <= 8 therefore maximum not updated
5 <= 8 therefore maximum not updated
4 <= 8 therefore maximum not updated
4 <= 8 therefore maximum not updated
3 <= 8 therefore maximum not updated
3 <= 8 therefore maximum not updated
2 <= 8 therefore maximum not updated
1 <= 8 therefore maximum not updated
0 <= 8 therefore maximum not updated
Height of the building is 8
```

12.9. Calculating the minimum

Calculating the minimum of a sequence of numbers is another important task in computer science. In Section 12.6 (page 206) you learned how to calculate the maximum. Calculating the minimum is almost the same, with two differences:

- The initial value of `minimum` must be greater to or equal than any value in the sequence. Alternatively, one can take the first value of the sequence.
- The value of `minimum` will be updated if the new value `m` is less than `minimum`.

Let us illustrate the algorithm on a sample sequence 7, 9, 5. We will initialize the value of `minimum` with 10:

Step 1:

- The current value of `minimum` is 10.
- Take the next value in the sequence: `m = 7`
- Is `m` less than `minimum`? (`m` is 7, `minimum` is 10)
- Yes, therefore `m` is the new minimum: `minimum = m`
- After this step, the value of `minimum` is at 7.

Step 2:

- The current value of `minimum` is 7.
- Take the next value in the sequence: `m = 9`
- Is `m` less than `minimum`? (`m` is 9, `minimum` is 7)
- No, therefore `minimum` stays unchanged.
- After this step, the value of `minimum` is at 7.

Step 3:

- The current value of `minimum` is 7.
- Take the next value in the sequence: `m = 5`
- Is `m` less than `minimum`? (`m` is 5, `minimum` is 7)
- Yes, therefore `m` is the new minimum: `minimum = m`
- After this step, the value of `minimum` is at 5.

There will be as many steps as there are values in the sequence. Now let's apply this algorithm to measure the clearance of an unknown cave!

In every step compare the new value `m` with the current minimum.
If `m` is less than the minimum, decrease the minimum to `m`.

12.10. Measuring the clearance of a cave

Today, Karel is exploring a cave. In order to know whether all his equipment will pass through it, he wants to first measure the clearance (minimum height between the floor and the ceiling).



Karel is measuring the clearance of a cave.

He will use the algorithm from the previous section. Since the maximum height of the maze is 12 grid squares, a good initial value for `minimum` is 12:

```
minimum = 12
while not home
  # Measure the height of the current column:
  m = 1
  left
  while not wall
    inc(m)
    go
  # Turn around
  right
  right
  # Get back down:
  while not wall
    go
  # Get ready for the next column:
  left
  go
  # Update the minimum if M < minimum:
```

```
19    if m < minimum
20      print(m, "<", minimum, "therefore updating minimum to", m)
21      minimum = m
22    else
23      print(m, ">=", minimum, "therefore minimum not updated")
24  print("Clearance of the cave is", minimum)
```

Output:

```
7 < 12 therefore updating minimum to 7
6 < 7 therefore updating minimum to 6
6 >= 6 therefore minimum not updated
5 < 6 therefore updating minimum to 5
7 >= 5 therefore minimum not updated
6 >= 5 therefore minimum not updated
4 < 5 therefore updating minimum to 4
5 >= 4 therefore minimum not updated
5 >= 4 therefore minimum not updated
4 >= 4 therefore minimum not updated
3 < 4 therefore updating minimum to 3
4 >= 3 therefore minimum not updated
Clearance of the cave is 3
```

12.11. Generating random Booleans

Karel has a function `rand` which returns either `True` or `False` with a 50 % probability. In other words, there is the same chance of returning `True` or `False`.

For illustration, let's call the function `rand` 10 times and display the result:

```
1  repeat 10
2    print(rand)
```

Output:

```
True
False
True
False
True
True
True
False
False
True
```

12.12. Karel is tossing a coin

Calling the function `rand` is the same as tossing a coin. Today Karel does not know whether he should go to cinema or read a book. The following program which simulates a coin toss will help him decide:

```
1 heads = rand
2 if heads
3   print("I got heads, so I will go to the cinema.")
4 else
5   print("I got tails, so I will read a book.")
```

Output:

```
I got heads, so I will go to the cinema.
```

So, Karel decided to go to the cinema today!

12.13. Calculating probabilities

This is easy. Probability is just the number of *favorable outcomes* divided by the number of *all possible outcomes*. Let's return to the coin toss for a moment.



What is the probability of getting heads? The number of favorable outcomes is 1, the number of all possible outcomes is 2. Therefore, the probability of getting heads is 1/2 which is 0.5. It is possible to report probabilities in percent, by multiplying the result by 100. Hence, the probability of getting heads is 50 %.

> Probability is the number of *favorable outcomes*
> divided by the number of *all possible outcomes*.

Now let's say that we toss the coin twice.

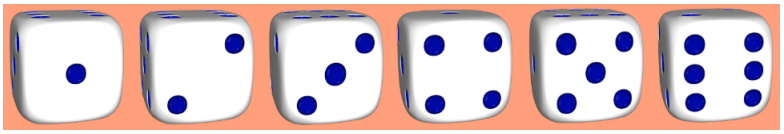


What is the probability of getting heads both times? Well, there are four possible outcomes: Getting heads in the first toss and the second, getting heads in the first toss and tails in the second, getting tails in the first toss and heads in the second, and getting tails both times. But only 1 outcome is favorable - that's the first one (heads and heads). Therefore, the probability of getting heads in both cases is 1 / 4 = 0.25, or 25 %.

Next, what is the probability of getting a 6 on a die? There is 1 favorable outcome:



The number of all possible outcomes is 6:



Therefore, the probability of getting a six is 1/6 which is approximately 0.17 or 17 %.

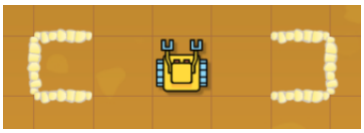
As a last example, let's roll the dice two times. What is the probability of the sum of the points being at most three? Well, that's easy. There are 36 possible outcomes: 1 in the first roll and 1 in the second, 1 in the first roll and 2 in the second, ..., 1 in the first roll and 6 in the second. That's six. Then 2 in the first roll and 1 in the second, 2 in the first roll and 2 in the second, ..., 2 in the first roll and 6 in the second. That's another six. There are four more sets of six, for getting 3, 4, 5, 6 in the first roll. Hence 6 times 6 is 36.

What is the number of favorable outcomes? Well, the sum can never be 1 because in each roll the minimum is 1. So the minimum score is 1 + 1 = 2. The score of three can only be obtained as 1 + 2 or 2 + 1. Hence, there are 3 favorable outcomes. In summary, the probablity of getting a score of at most three in a double roll is 3 / 36 = 1 / 12. This is approximately 0.08 or 8 %.

12.14. Probability of events vs. their frequency

The probability of an event only is an *approximate indicator* for the frequency of that event. For example, the probability of getting heads when tossing a coin is 50 %. But this does not mean that one gets heads exactly one time when tossing the coin twice. Let's get Karel's help to explain this in more detail.

Today, Karel has many ribbons in his bag and he wants to split them into two parts. He will be tossing a coin to do that. When he gets heads (`True`), he will place a ribbon on the left. When he gets tails (`False`), he will place a ribbon on the right.



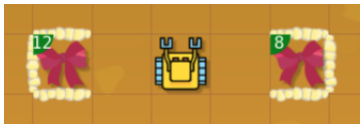
Karel wants to split the contents of his bag into two parts.

He will use the following program, starting with 20 ribbons:

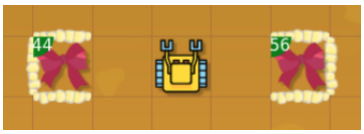
```
repeat 20
  if rand
    left
    go
    go
    put
    right
    right
    go
    go
    left
  else
    right
    go
    go
    put
    left
    left
    go
    go
    right
```

Will he get 10 on the left and 10 on the right?

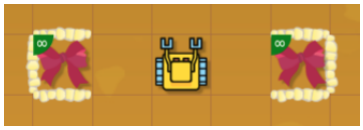
And here is the result - 12 ribbons on the left and 8 on the right!



Let's do this again, now with 100 ribbons (the number of repetitions on line 1 will be changed to 100). When the program finishes, there are 44 ribbons on the left and 56 on the right:



And one last time, let's do the same with 500 ribbons. The result is - 238 ribbons on the left and 262 on the right:



You can observe one very interesting thing here. If the exact half is the perfect outcome, then the first time the result was 2 ribbons off (12 instead of 10), the second time the result was 6 ribbons off (44 instead of 50), and the last time the result was 12 ribbons off (238 instead of 250). That number seems to be growing. But to be fair, one has to make it proportional to the number of events - which means to divide it by the number of coin tosses. Hence, in the first case we obtain 2 / 20 = 0.1, in the second case 6 / 100 = 0.06, and in the third case 12 / 500 = 0.024.

This number is the relative discrepancy between the frequency of the event and its predicted probability. As you can see, the value steadily decreases from 0.1 to 0.06 to 0.024. In other words, the relative frequency of the event is getting closer to its predicted probability. In mathematics, this is called the *Law of Large Numbers (LLN)*. If you are interested, you can find more about it on Wikipedia.

The Law of Large Numbers states that with an increasing number of events, the relative frequency will be getting closer to the predicted probability.

12.15. Karel and random walks

The Boolean function `rand` allows us to do very interesting things. Such as - we can let Karel make random walks in the maze! What does it mean? Instead of just letting him make one step forward, we can let him make the step forward with probability 50 %. The other 50 % we can split between turning left and turning right. In other words, he will turn left with probability 25 % or right with probability 25 %. This is the corresponding program (the number of steps can be changed as needed):

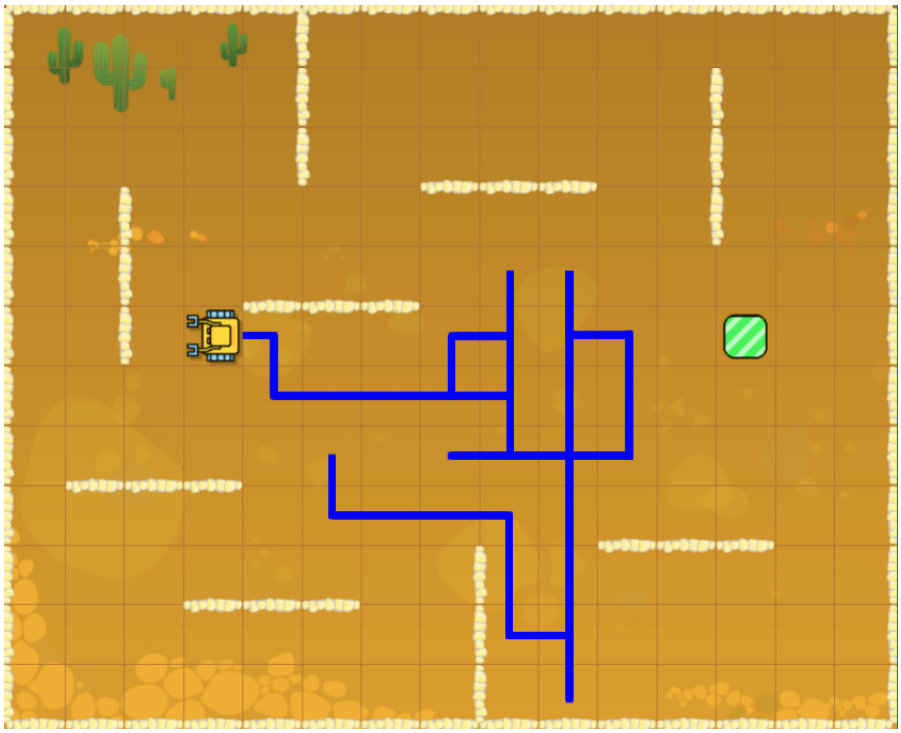
```
1 repeat 100        # make 100 steps
2   if rand         # go straight with probability 50%
3     if not wall
4       go
5   else            # the else branch is executed with probability 50%
6     if rand       # turn left with probability 25%
7       left
8     else          # turn right with probability 25%
9       right
```

As you can see, we also added a check for a wall on line 3 - without it, the robot would crash into a wall in no time. Here is the maze where we will try this program out:



Karel's initial position.

And here you can see where Karel went when the program was executed:



The trace of Karel's random walk.

Notice a few "dead ends". They occur when Karel turns twice in a row and goes back.

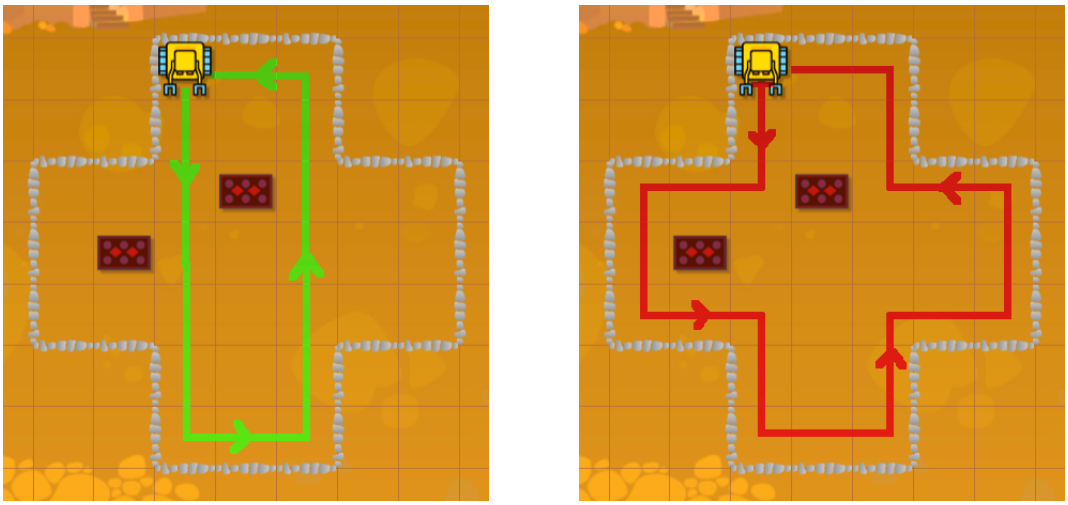
12.16. Using randomness to solve difficult tasks

Some tasks are very difficult to solve using deterministic (non-random) algorithms.



Karel needs to collect two rugs at unknown positions.

This maze is not suitable for the maze algorithms we know. The First Maze Algorithm fails to find the rugs (green line) and so does the Second Maze Algorithm (red line).



Both the First and Second Maze Algorithms fail.

Both these algorithms actually get into an infinite loop. One possible solution is to let Karel walk randomly in the maze until he finds the rugs. Well, for simplicity, let us have him stop after he finds the first one. Here is the corresponding program:

```
while empty      # repeat until you find the first rug
  if rand        # go straight with probability 50%
    if not wall
      go
  else           # the else branch is executed with probability 50%
    if rand      # turn left with probability 25%
      left
    else         # turn right with probability 25%
      right
  if rug         # if you find a rug, collect it
    get
```

This works. Use the Karel App in NCLab to re-create the maze, and try it yourself!

12.17. Review questions

QUESTION 12.1. *Which of the following areas of computing benefit from randomness?*
A logic

B cryptography
C game design
D scientific computing

QUESTION 12.2. *What function in Karel is used to generate random integers?*
A `random`
B `rint`
C `randomint`
D `randint`

QUESTION 12.3. *What is the equivalent of rolling a die?*
A `randint(1, 6)`
B `randint(0, 6)`
C `randint(1, 7)`
D `randint`

QUESTION 12.4. *What is the maximum of the numbers 3, 9, 6, 7, 2?*
A 2
B 3
C 7
D 9

QUESTION 12.5. *If* `maximum` *is the current value of the maximum, and the next value is* `m`*, what condition is used to update* `maximum`*?*
A `if m > maximum`
B `if m < maximum`

QUESTION 12.6. *What is the minimum of the numbers 3, 9, 6, 7, 2?*
A 2
B 3
C 7
D 9

QUESTION 12.7. *If* `minimum` *is the current value of the minimum, and the next value is* `m`*, what condition is used to update* `minimum`*?*
A `if m > minimum`
B `if m < minimum`

QUESTION 12.8. *When tossing a coin, what is the probability of getting heads?*
A 50 %

B 0.5
C 5.0
D 100 %

QUESTION 12.9. *When tossing a coin two times, what is the probability of getting heads both times?*

A 50 %
B 25 %
C 0.5
D 0.25

QUESTION 12.10. *When rolling a die, what is the probability of getting a 3?*

A 1/6
B 3/6
C 1/2
D 1

QUESTION 12.11. *When rolling a die, what is the probability of getting an even number?*

A 1/2
B 1/6
C 1/3
D 0

QUESTION 12.12. *When rolling a die two times, what is the probability of getting a combined score of 2?*

A 0
B 1/2
C 1/18
D 1/36

QUESTION 12.13. *A favorable event has probability 50%, and we do 10 experiments. How many times will the favorable event occur?*

A 5 times
B 4 times
C 6 times
D This is impossible to say.

13. Lists

In this chapter you will learn:

- What lists are and why they are useful.
- How to create empty and nonempty lists.
- How to append values to a list.
- How to measure the length of a list.
- How to access list items via their indices.
- How to parse lists via the `for` loop.
- How to check if a given item is in a list.
- How to remove and return ("pop") items from a list.
- How to add lists and multiply them with integers.
- How to delete items from a list.

As usual, first we will cover the necessary theory and then we will show you some cool applications of lists at the end of this chapter.

13.1. What are lists and why they are useful

You already know how to create and use numerical, text string and even Boolean variables. Variables are like containers - every variable can store one value. In contrast to this, a list is like a cargo train which can store many different values.



A list is like a cargo train.

The values in a list are ordered, so one knows which one is the first and which one is the last. They can have different types - a list may contain numerical values, text strings, Booleans, and even other lists. Items can be dynamically added and/or removed at run-time as needed.

Lists are Karel's (and Python's) most powerful data structure. The implementation of lists in Karel is compatible with Python, although Karel does not provide all the list functionality offered by Python.

13.2. Creating empty and nonempty lists

Throughout this chapter, you will see that there are strong similarities between lists and text strings. To begin with, an empty text string `txt` is created via `txt = ''` or `txt = ""`. It is also possible to create a nonempty text string such as `msg = 'Hi there!'`.

An empty list is created using a pair of empty square brackets:

```
1 L = []
```

Do not use parentheses () or curly braces {} as they have a different meaning. You can also create a non-empty list:

```
1 S = [2, 3, 5]
```

A list can contain text strings,

```
1 W = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Boolean values,

```
1 B = [True, False, False, True, False, True]
```

and the types of values in a list can even be mixed:

```
1 M = [1, 2, 'Emily', 'Jocob', 'True', 'False']
```

Empty and nonempty lists are created using square brackets.

13.3. Appending items to a list

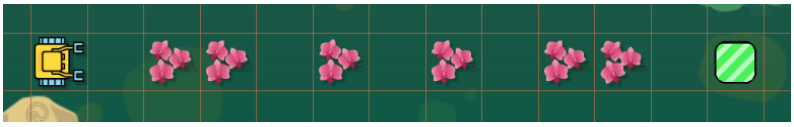
Items can be appended to the end of a (empty or nonempty) list using the list method `append`:

```
1 L = []
2 print("L =", L)
3 L.append(5)
4 print("L =", L)
5 L.append(10)
6 print("L =", L)
```

Output:

```
L = []
L = [5]
L = [5, 10]
```

Lists are perfect for storing several values at once. Like today, when Karel needs to remember the positions of all the orchids:



He can do it as follows:

```
1 X = []
2 while not home
3   if orchid
4     X.append(gpsx)
5   go
6 print("Orchids are in columns:", X)
```

Output:

```
Orchids are in columns: [3, 4, 6, 8, 10, 11]
```

Items can be appended to the end of a list using the list method `append`.

13.4. Measuring the length of a list

From Section 9.5 (page 156) you know that `len(txt)` returns the length of the text string `txt`. Similarly, when working with lists, calling `len(L)` will return the length of the list `L`. The length of a list means the number of its items. For illustration, let's change the last line of the previous program, and display the length of the list `X`:

```
X = []
while not home
  if orchid
    X.append(gpsx)
  go
print("I found", len(X), "orchids.")
```

Output:

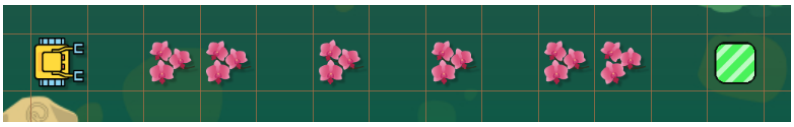
```
I found 6 orchids.
```

13.5. Accessing list items via their indices

You already know from Section 9.8 (page 157) how to access individual characters in a text string using their indices. For example, `txt[0]` is the first character in the text string `txt`, `txt[1]` is the second character, etc. Also, `txt[-1]` is the last character of the text string, `txt[-2]` is the second from the end, and so on.

When working with lists, it is possible to access individual list items exactly in the same way. For example, `X[0]` is the first item of the list `X`, `X[1]` is its second item, `X[-1]` is its last item, etc.

Today, Karel is still counting orchids:



But now he wants to know the position of the first orchid, and the position of the last one before his home square:

```
X = []
while not home
  if orchid
    X.append(gpsx)
```

```
  go
print("The first orchid was found in column:", X[0])
print("The last orchid was found in column: ", X[-1])
```

Output:

```
The first orchid was found in column: 3
The last orchid was found in column:  11
```

Items in a list can be accessed using indices
in the same way as characters in a text string.

13.6. Creating a list of lists

Karel is still in the jungle counting orchids. But now, there can be multiple of them per grid square:



He needs to not only remember their positions, but also their numbers. This naturally leads to a list of pairs of the form "position, number". In fact, each of these pairs can be a two-item list, and these lists can be stored in a list. Hence one obtains a list of lists. Here is the code from Section 13.3 (page 225), updated for the current situation:

```
# Count orchids in grid square:
def count
  n = 0
  while orchid # collect and count them
    get
    inc(n)
  while not empty # put them back
    put
  return n

# Main program:
X = []
while not home
```

```
14    if orchid
15      o = count
16      X.append([gpsx, o])
17    go
18  print("List of [position, number] pairs:")
19  print(X)
```

Output:

```
List of [position, number] pairs:
[[3, 2], [5, 1], [6, 3], [7, 2], [9, 5], [11, 4]]
```

Lists can contain other lists.

13.7. Parsing lists with the `for` loop

Let's stay with the previous example for another moment:



The text output was too compact, and it would be good to make it nicer. It should say something like "I found 2 orchids in column 3, 1 orchid in column 5, ..." etc.

You already know from Section 9.6 (page 156) how to parse text strings one character at a time. Lists can be parsed one item at a time exactly in the same way. For example, with the list `X` from the previous example, the code

```
1  for x in X
2    print(x)
```

will display

```
[3, 2]
[5, 1]
[6, 3]
[7, 2]
[9, 5]
[11, 4]
```

This is still not very nice, but we are getting closer. Each item `x` in the list `X` is a two-item list containing a position-number pair. Therefore we know that `x[0]` is the position and

`x[1]` the corresponding number. Therefore, a nicer output can be arranged as follows:

```
1 print("I found:")
2 for x in X
3   print(x[1], "orchids in column", x[0])
```

This will display

```
I found:
2 orchids in column 3
1 orchids in column 5
3 orchids in column 6
2 orchids in column 7
5 orchids in column 9
4 orchids in column 11
```

Although there is a minor English imperfection, this is way better than before. If we wanted to improve line 3, we would have to include one extra condition:

```
1 print("I found:")
2 for x in X
3   if x[1] == 1
4     print(x[1], "orchid in column", x[0])
5   else
6     print(x[1], "orchids in column", x[0])
```

Final output:

```
I found:
2 orchids in column 3
1 orchid  in column 5
3 orchids in column 6
2 orchids in column 7
5 orchids in column 9
4 orchids in column 11
```

13.8. Checking if an item is in a list

You already know from Section 9.12 (page 161) that Karel can check for the presence of a substring in a text string using the keyword `in`. The same keyword can be used to check whether an item is present in a list. For illustration, the list `X` we created in Section 13.3 (page 225) provides information about the column positions of orchids:

```
Orchids are in columns: [3, 4, 6, 8, 10, 11]
```

Karel can now use the list to ask whether there is an orchid in column number 5:

```
col = 5
if col in X
  print("There is an orchid in column number", col)
else
  print("There is no orchid in column number", col)
```

Output:

```
There is no orchid in column number 5
```

Or, he can ask whether there is an orchid in column number 10:

```
col = 10
if col in X
  print("There is an orchid in column number", col)
else
  print("There is no orchid in column number", col)
```

Output:

```
There is an orchid in column number 10
```

The list of two-item lists `X` that we created in Section 13.6 (page 227) provides information about positions of orchids and their numbers:

```
List of [position, number] pairs:
[[3, 2], [5, 1], [6, 3], [7, 2], [9, 5], [11, 4]]
```

Karel can use it to ask whether there are two orchids in column number 7:

```
col = 7
n = 2
if [col, n] in X
  print("There are", n, "orchids in column number", col)
else
  print("There aren't", n, "orchids in column number", col)
```

Output:

```
There are 2 orchids in column number 7
```

13.9. Removing and returning ("popping") items from a list

At the beginning of this chapter we said that lists can easily be modified at runtime, but so far we only showed you how to append new items at the end. Karel (and Python) provide the list method `pop` which will remove and return items from a list. Let's look at a sample list `names = ['Ann', 'Brett', 'Charles', 'Dave', 'Emily', 'Frank']`.

Calling `names.pop()` will remove and return the last item, `'Frank'`:

```
1 names = ['Ann', 'Brett', 'Charles', 'Dave', 'Emily', 'Frank']
2 print("Before:", names)
3 print("Popping", names.pop())
4 print("After:", names)
```

Output:

```
Before: ['Ann', 'Brett', 'Charles', 'Dave', 'Emily', 'Frank']
Popping Frank
After: ['Ann', 'Brett', 'Charles', 'Dave', 'Emily']
```

The method `pop` can also be called with the index of a specific item we want to remove and return. For example, calling `names.pop(0)` will remove and return the item `'Ann'` from the list:

```
1 names = ['Ann', 'Brett', 'Charles', 'Dave', 'Emily']
2 print("Before:", names)
3 print("Popping", names.pop(0))
4 print("After:", names)
```

Output:

```
Before: ['Ann', 'Brett', 'Charles', 'Dave', 'Emily']
Popping Ann
After: ['Brett', 'Charles', 'Dave', 'Emily']
```

And as a last example, calling `names.pop(-2)` will remove and return the second item from the end which is `'Dave'`:

```
1 names = ['Brett', 'Charles', 'Dave', 'Emily']
2 print("Before:", names)
3 print("Popping", names.pop(-2))
4 print("After:", names)
```

Output:

```
Before: ['Brett', 'Charles', 'Dave', 'Emily']
Popping Dave
After: ['Brett', 'Charles', 'Emily']
```

List items can be removed and returned using the method `pop`.

13.10. Adding lists

From Section 9.3 (page 155) you know that text strings can be added just as numbers. Lists can be added in the same way:

```
1 names = ['Brett', 'Charles', 'Dave', 'Emily']
2 new_names = ['Fred', 'Gillian', 'Harry']
3 print(names + new_names)
```

Output:

```
['Brett', 'Charles', 'Dave', 'Emily', 'Fred', 'Gillian', 'Harry']
```

And one can even use the += operator to extend a list with another one:

```
1 names = ['Brett', 'Charles', 'Dave', 'Emily']
2 new_names = ['Fred', 'Gillian', 'Harry']
3 names += new_names
4 print(names)
```

Output:

```
['Brett', 'Charles', 'Dave', 'Emily', 'Fred', 'Gillian', 'Harry']
```

13.11. Multiplying lists with integers

The analogy between lists and text strings goes further. In Section 9.4 (page 155) you have seen that a text string can be multiplied with a positive integer `N`, which will copy and paste its contents `N` times. The same can be done with lists:

```
1 morse = ['....', '..']
2 print(morse*4)
```

Output:

```
['....', '..', '....', '..', '....', '..', '....', '..']
```

And as you would expect, the *= operator works as well:

```
1 morse = ['....', '..']
2 morse *= 4
3 print(morse)
```

Output:

```
['....', '..', '....', '..', '....', '..', '....', '..']
```

Lists can be added and multiplied with positive integers analogously to text strings.

13.12. Deleting items from a list

Sometimes one just needs to delete an item from a list (and destroy it) because there is no use for it. This can be done using the keyword `del`. Typing `del L[i]` will delete and destroy the item with index `i` from the list `L`. Let's illustrate this on another Morse example:

```
1 morse = ['.--', '---', '.-.', '.-..', '-..']
2 print("Before:", morse)
3 print("Removing letter L.")
4 del morse[3]
5 print("After:", morse)
```

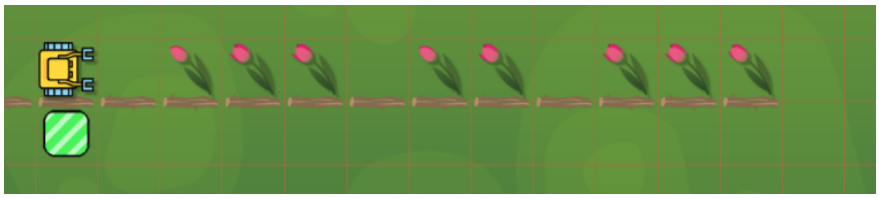
Output:

```
Before: ['.--', '---', '.-.', '.-..', '-..']
Removing letter L.
After: ['.--', '---', '.-.', '-..']
```

Typing `del L[i]` will delete and destroy the item with index `i` from the list `L`.

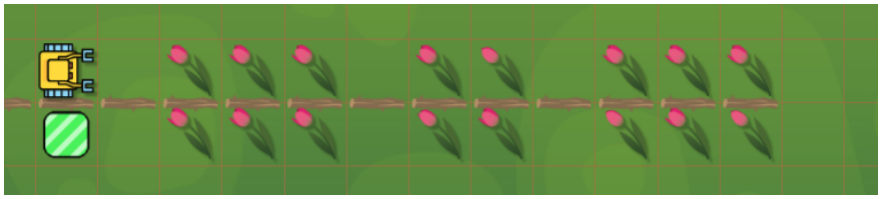
13.13. Gardener

Let's show a task which would be difficult to solve without using a list. Karel is working in his garden. In front of him is a flower bed with tulips:



Karel is gardening.

The robot does not know the length of the flower bed, or the number or positions of the tulips. His task is to create an identical flower bed on the other side of the wall:



He has enough tulips in his bag to do it.

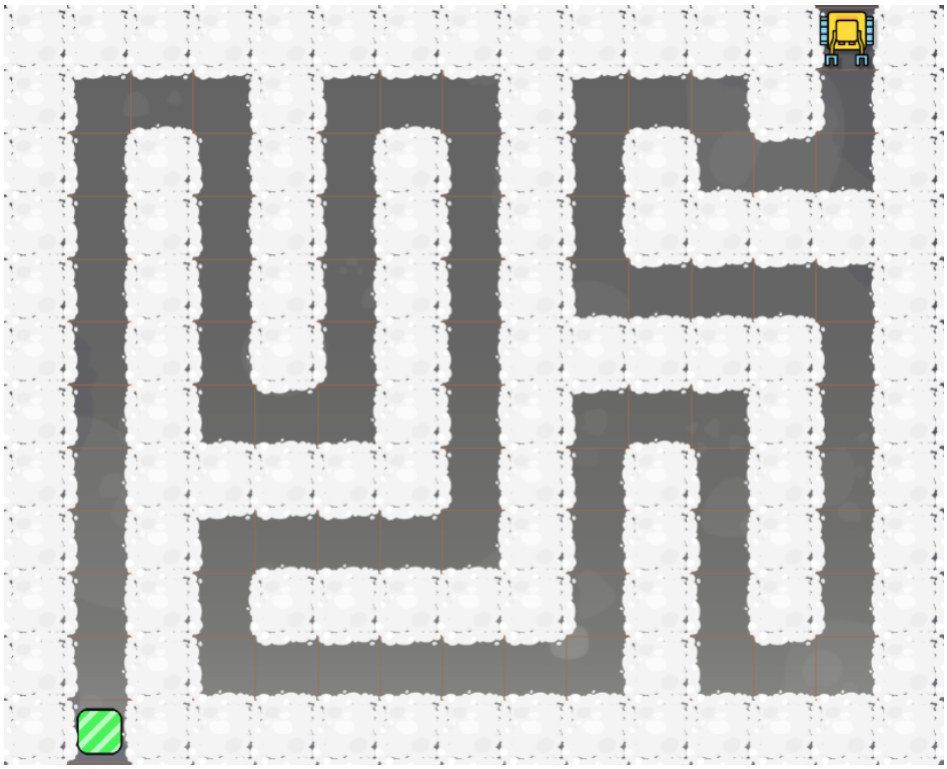
The best way to solve this task is to follow the wall until its end, and store the column positions of all the tulips in a list `T`. Then, on the other side of the wall, Karel will walk towards his home square. In each grid square he will check whether his `gpsx` coordinate is in the list `T`. If the value is found, he will place a tulip. Here is the corresponding code:

```
T = [] # to store the column positions of the tupils
right
while wall # follow the wall to its end
  left
  go
  right
  if tulip # if there's a tulip, add gpsx to the list T
    T.append(gpsx)
go # go over to the other side of the wall
right
while not home # walk home
  go
  if gpsx in T # if the gpsx value is in T, place an orchid
    put
```

13.14. Expedition Antarctica

The following task is interesting not only because it would be very difficult to solve without using a list, but also because it involves a discussion of different ways to store data in the list.

Karel is part of an expedition to Antarctica, and his task is to find a way through ice and snow. He should record his path in such a way that later he can draw a map.



Karel is in Antarctica.

Clearly, the robot can use the First Maze Algorithm to pass through the maze. He will need to store the information about his path in a list (say `P`). But this can be done in many different ways.

Option #1: For every step he makes, Karel can add `0` to the list `P`. When he needs to turn right, he can add `True`. When he needs to turn left, he can add `False`.

Option #2: At every turn Karel can add to the list `P` a triplet `[gpsx, gpsy, R]` where `R` is `True` if the path goes to the right and `False` otherwise.

Option #3: Karel can count his steps and store their number in a variable `N`. At every turn he can add to the list `P` a pair `[N, R]` where `R` is `True` if the path goes to the right and `False` otherwise. He would reset `N` back to `0` after every turn.

Let's evaluate these three options: The sample path shown above involves 77 steps and 20 turns. So, option #1 would produce a list of length $77 + 20 = 97$. Option #2 would add three values at every turn, hence the resulting list would have length $3 \cdot 20 = 60$. And finally, option #3 would add two values at every turn, yielding a list of length $2 \cdot 20 = 40$. Since option #3 is most memory-efficient, we will implement it.

The following program will do it. Make sure to read the comments in the code:

```
P = [] # to store path information
N = 0  # counter of steps
while not home
  go
  inc(N)  # increase counter of steps
  if wall # First Maze Algorithm
    right
    if wall
      left
      left
      P.append([N, False]) # path goes left
    else
      P.append([N, True])  # path goes right
    N = 0 # reset counter of steps
print(P)
```

Output:

```
[[2, True], [2, True], [1, False], [2, False], [3, False], [4, True], [6,
True], [2, True], [4, False], [2, False], [4, True], [6, True], [2, True],
[4, False], [7, False], [2, False], [5, True], [2, True], [5, False], [2,
False], [10, True]]
```

Awesome - first part done! The second part of the task is to use this list to draw a map of the path. The program is very simple. Karel will draw the path by placing beepers he has in his bag:

```
# Path information:
P = [[2, True], [2, True], [1, False], [2, False], [3, False], [4, True],
    [6, True], [2, True], [4, False], [2, False], [4, True], [6, True], [2,
    True], [4, False], [7, False], [2, False], [5, True], [2, True], [5,
    False], [2, False], [10, True]]

```

```
# Draw the path by placing beepers!
for p in P          # pass through all pairs in P
  repeat p[0]       # make p[0] steps forward
    put             # place a beeper before each step
    go
  if p[1]           # if p[1] is True, turn right
    right
  else              # otherwise turn left
    left
```

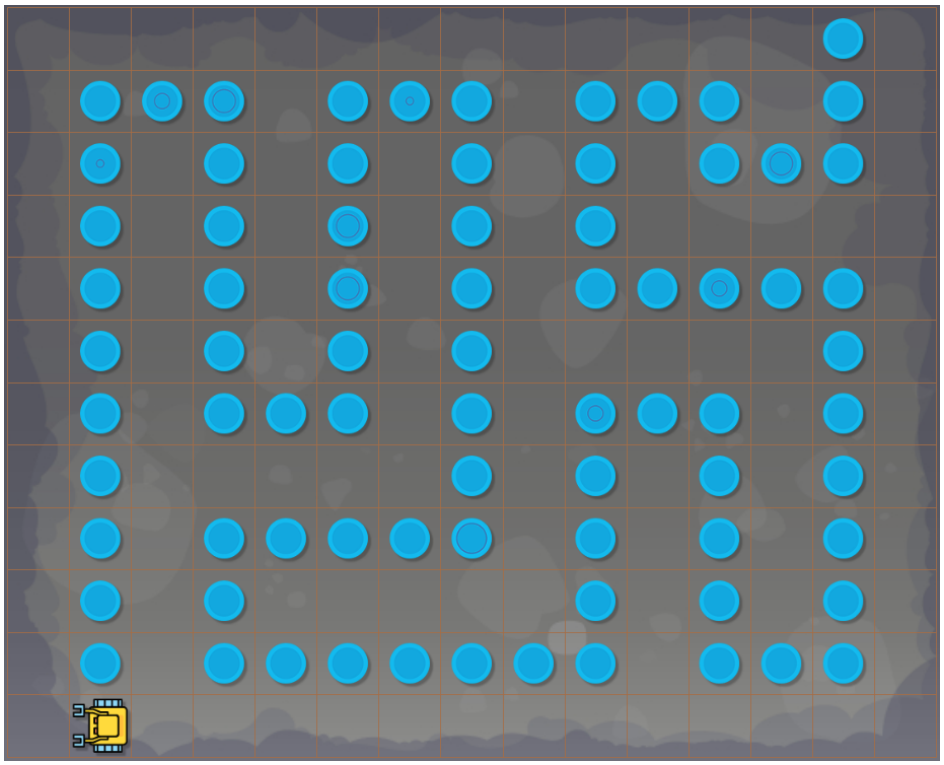
Let's start from a clean sheet:



Karel is ready to draw the path.

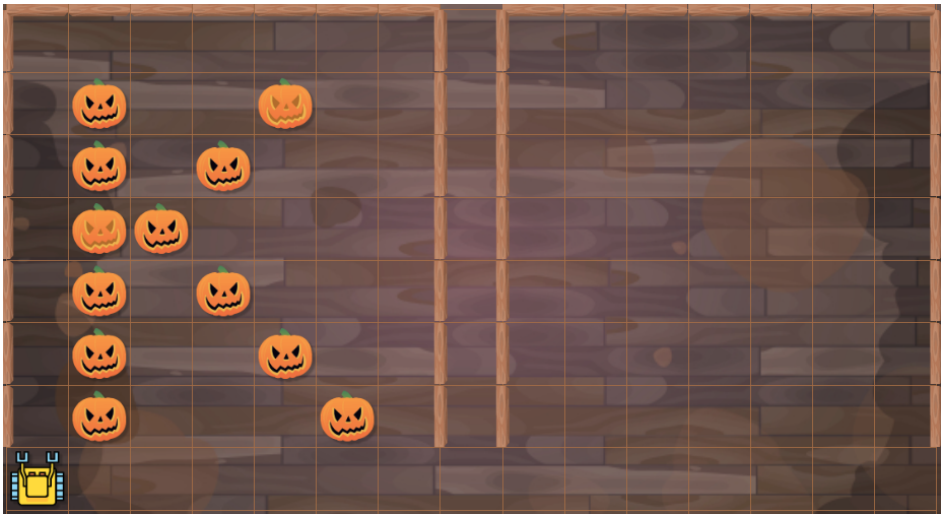
Always try to find different ways to solve the given task.
Then compare them, and choose the best one.

And here is Karel's drawing after the program finishes:



13.15. Copycat

Karel's next task is to copy a random pattern from the left box and paste it in the box on the right:



Hence the robot needs to visit all grid squares in the left box, learn where the pumpkins are, then visit all grid squares in the box on the right again, and place pumpkins at the corresponding positions.

Rather than writing a complicated code for the robot to visit all grid squares in the first box, let's use the method from the previous section. Here is a path through the first box, stored in a list named `H`:

```
# Path to visit all grid squares in the box:
H = [[7, True], [6, True], [1, True], [5, False], [1, False], [5, True],
    [1, True], [5, False], [1, False], [5, True], [1, True], [5, False], [1,
     False], [5, True], [1, False]]
```

We will use this list and the previous program to visit all grid squares in the left box, and then also in the box on the right. This time we will use a Boolean list `B` to store the pattern. Initially, this list will be empty. After each step, Karel will append to it `True` if he found a pumpkin, and `False` if he did not. In the box on the right, the list `B` will be used to place pumpkins at the corresponding positions. Here is the complete code:

```
# Path to visit all grid squares in the box:
H = [[7, True], [6, True], [1, True], [5, False], [1, False], [5, True],
    [1, True], [5, False], [1, False], [5, True], [1, True], [5, False], [1,
     False], [5, True], [1, False]]

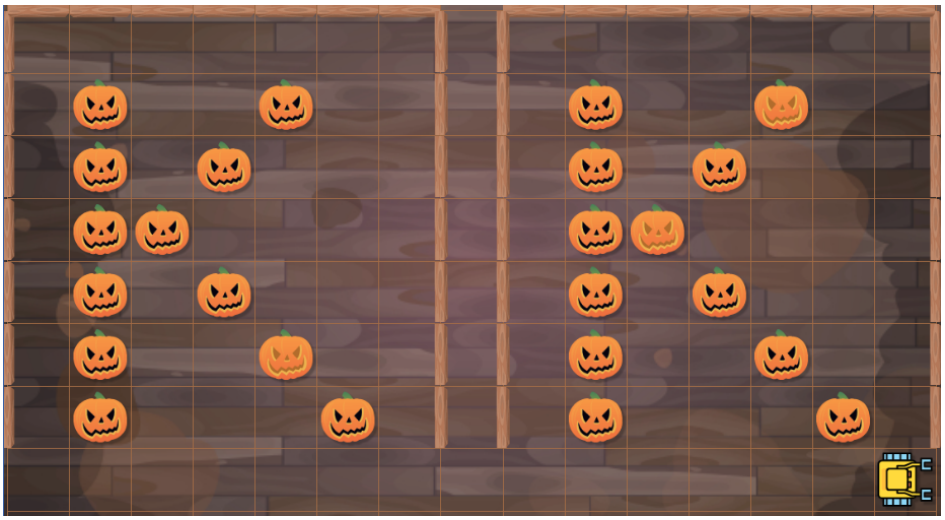
# Go through the box on the left:
B = []           # create an empty Boolean list
for p in H       # pass through all pairs in the list H
  repeat p[0]    # make p[0] steps forward
    go
    B.append(pumpkin) # if pumpkin, append True, else append False
  if p[1]        # if p[1] is True, turn right
    right
  else           # otherwise turn left
    left

# Move to the second box:
go
go
left

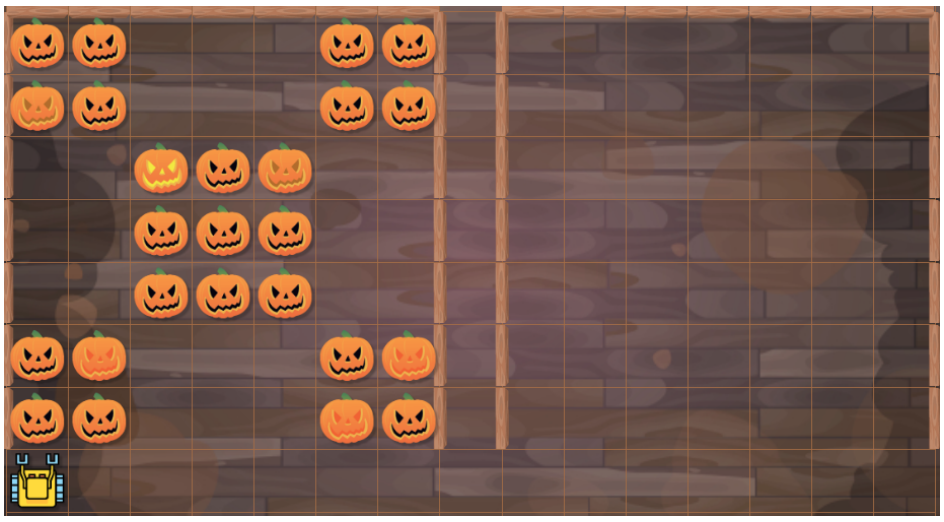
```

```
# Go through the box on the right:
for p in H       # pass through all pairs in the list H
  repeat p[0]    # make p[0] steps forward
    go
    if B.pop(0) # get the first item from the list B
      put
  if p[1]        # if p[1] is True, turn right
    right
  else           # otherwise turn left
    left
```

When the program finishes, Karel has reproduced the pattern from the left box:



To make sure that the program works for other patterns as well, let's try at least one more:



And here is the corresponding output:



13.16. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 13.1. *Check all true statements about variables and lists!*

A *Variables can store numerical values, lists cannot.*

B *Lists can store Boolean values, variables cannot.*

C *A variable can only store a single value.*

D *A list can store multiple values.*

QUESTION 13.2. *How can one create a list named* `L` *containing the letters* `'A'`, `'B'` *and* `'C'`*?*

A
```
L = ('A', 'B', 'C')
```

B
```
L = ['A', 'B', 'C']
```

C
```
L = {'A', 'B', 'C'}
```

D
```
L = "'A' 'B' 'C'"
```

QUESTION 13.3. *How can the value of a variable* `v` *be appended to a list* `V` *?*

A
```
V.append(v)
```

B
```
V.add(v)
```

C
```
V.pop(v)
```

D
```
V += v
```

QUESTION 13.4. *How can one measure the length of a list* `X` *?*

A `length(X)`
B `length X`
C `len(X)`
D `len X`

QUESTION 13.5. *In the list* `[6, 4, 8, 9, 2, 3]`*, what is the index of the value 2 ?*
A 4
B 5
C -1
D -2

QUESTION 13.6. *In the list* `[[1, 2], [3, 4], [5, 6]]`*, what is the index of the value 5 ?*
A `[3][1]`
B `[3][0]`
C `[2][0]`
D `[2][1]`

QUESTION 13.7. *There is a list* `L = [2, 3, 5, 7, 11, 13, 17, 19]`*. What will the list become after executing* `n = L.pop(5)` *?*
A `[2, 3, 7, 11, 13, 17, 19]`
B `[2, 3, 5, 7, 11, 17, 19]`
C `[2, 3, 5, 7, 11, 13, 19]`
D `[13, 17, 19]`

QUESTION 13.8. *There are lists* `L1 = [1, 1, 1]` *and* `L2 = [2, 2, 2]`*. What will be the result of* `L1 + L2` *?*
A `[3, 3, 3]`
B `[1, 1, 1, [2, 2, 2]]`
C `[1, 1, 1, 2, 2, 2]`
D An error message

QUESTION 13.9. *There is a list* `C = ['a', 'b', 'c', 'd']`*. What will be the result of* `C*2` *?*
A `['aa', 'bb', 'cc', 'dd']`
B `['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']`
C `['a', 'a', 'b', 'b', 'c', 'c', 'd', 'd']`
D An error message

14. Recursion

In this chapter you will learn:

- What recursion is and why it is useful.
- How to recognize whether or not a task is suitable for recursion.
- How to implement recursion correctly.
- What can happen if recursion is not done right.
- How to use variables, lists, and functions in recursion.
- About mutually recursive commands and functions.

At the end, we will show you examples of tasks which can easily be solved with recursion, but which would be extremely difficult to tackle without it.

14.1. What is recursion and why is it useful

Recursion is a fundamental concept of computational thinking.

"To iterate is human, to recurse divine." (L. Peter Deutsch)

This classical programming quote is 100% true - recursion is a beautiful and powerful tool. On the other hand, it is important to know that as any other tool, it is suitable for some tasks but not for all.

When you try to learn about recursion, you will come across statements like this:

"Programming technique where a function can call itself..."

or

"The process in which a function calls itself..."

Well, this is not exactly true, as we will see in Section 14.8 (page 252). But more importantly, recursion is much more. It can be found in many areas ranging from art and linguistics to mathematics and computer science.

Recursion is a repetitive process which takes an object,
event or activity and makes it "the same but smaller".

Let's see some examples, starting with the famous Russian dolls:



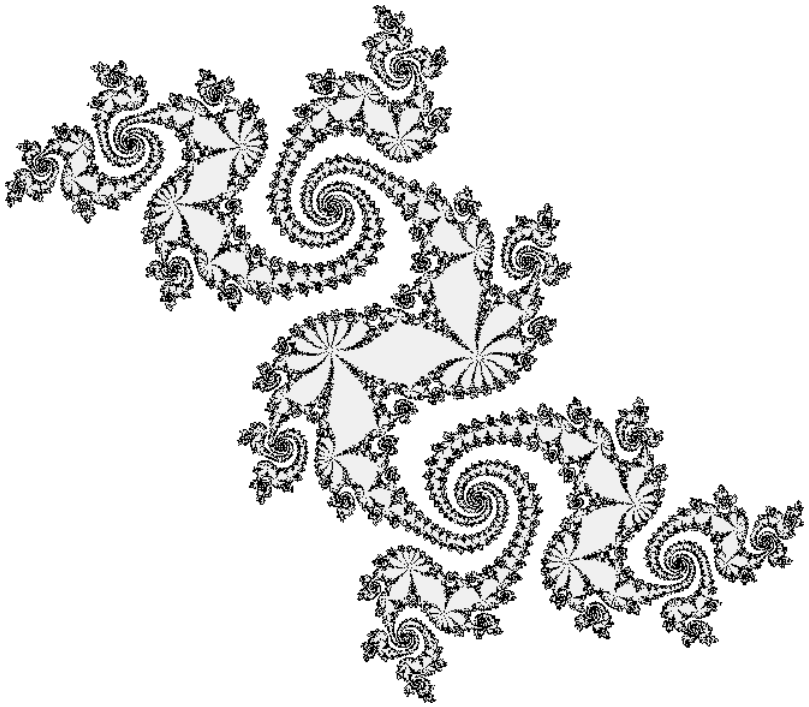
Russian dolls - an example of recursion in art.

Snowflakes are an example of recursion in nature:



A snowflake.

Fractals are mathematical objects which exhibit recursion. The following is an example of the Julia fractal:



Julia fractal.

In the following section we will give examples of tasks which are suitable for recursion.

14.2. Which tasks are suitable for recursion?

All recursive tasks have one thing in common: Part of the task can be solved by some method. The rest is then a smaller copy of the same task. So, one can solve part of it using the same method as before. The rest will then be a smaller copy of the same task. So, one can solve part of it using the same method... etc.

Example 1: Eating your lunch is a recursive task



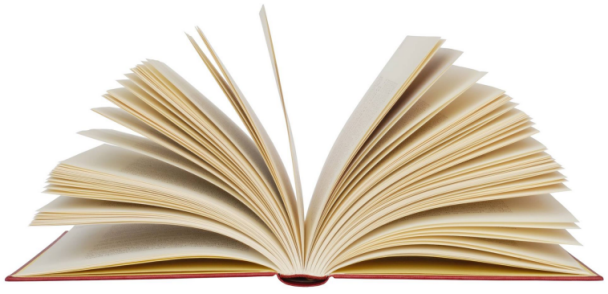
Imagine that your "task" is to eat your food. If you don't like chili, picture your favorite food now. How do you eat it? Well, first you check: Is there any more left? If so, you eat a bite. Then there is a bit less food to eat. So you check: Is there any more left? If so, you eat a bite. Then there is a bit less food to eat...

Example 2: Cleaning up is a recursive task



Your next task is to put all the Lego pieces back in the box. First you check the floor: Are there some pieces still lying around? If so, collect a few and put them in the box. Then there are fewer loose pieces on the floor. So you check: Are there some pieces still lying around? If so, collect a few and put them in the box. Then there are fewer loose pieces on the floor...

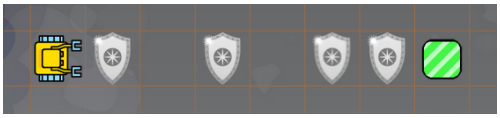
Example 3: Reading a book is a recursive task



Our third and last example is about reading a book. Your progress will be measured by the number of unread pages. First you check: Are there any more pages left to read? If so, read a few pages from where you left off. After that, fewer pages remain, so your task got smaller. So you check: Are there any more pages left to read? If so, read a few pages from where you left off. After that, fewer pages remain, so your task got smaller...

14.3. Collecting shields

Today, Karel's task is to collect all shields and enter the home square:



Collect all shields!

The recursive algorithm for Karel to do this is as follows:

- If you are not home:
  - If there is a shield beneath you, collect it.
  - Make one step forward.
- After this, the task is the same, just smaller: Collect all shields!

Notice the condition "if you are not home" at the beginning. This condition is very important in recursion. It is called the *stopping condition*. When one forgets it, recursion instantly turns into an infinite loop.



After one step, Karel's task is the same: Collect all shields!

And here is the corresponding code. Notice that after collecting one shield and moving one step forward, the command `walk` calls itself on line 9. Also, notice that the recursive call on line 9 is made from the inside of a stopping condition `if not home`. This means that when Karel arrives at home, the command `walk` is not called anymore and the recursion ends:

```
1 # Recursive command:
2 def walk
3   if shield
4     get
5   go
6   # Stopping condition:
7   if not home
8     print("Recursive call...")
```

```
9       walk
10      print("Recursive call ended.")
11    return
12
13  # Main program:
14  print("Calling 'walk' from the main program...")
15  walk
16  print("Call from the main program ended.")
```

The recursive call always must be made from the body of a stopping condition. If one forgets this, the recursion will instantly turn into an infinite loop.

14.4. Under the hood

The last program contained some control outputs to help us understand the program flow better. Let's have a look at them:

```
Calling 'walk' from the main program...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call ended.
Recursive call ended.
Recursive call ended.
Recursive call ended.
Recursive call ended.
Recursive call ended.
Call from the main program ended.
```

The first line corresponds to calling `walk` on line 15 in the main program. The second line corresponds to executing `walk` on line 9 of its own body. At this point, a new copy of the command `walk` is created in the computer memory and called. The original command `walk` which was called on line 15 is put on hold.

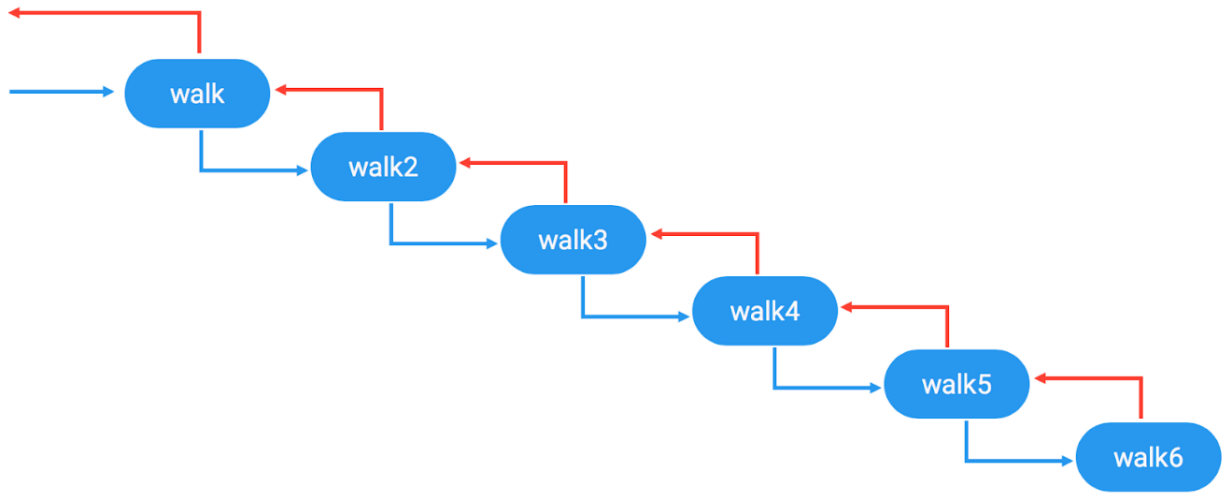
To make the explanation simpler, let's name the new copy `walk2` (although in reality the names do not change like this). The command `walk2` executes `walk` on line 9 again. This creates a new copy of the command `walk` which we can name `walk3`. The new copy `walk3` is called and the copy `walk2` is put on hold.

This process continues and other three copies `walk4`, `walk5` and `walk6` are created and called. Finally, in `walk6`, Karel enters the home square and therefore the stopping

condition on line 7 is not satisfied. Therefore, the program goes directly to line 11 and `walk6` ends.

At this moment, the control is returned to `walk5` where the program continues on line 10 and displays for the first time the message `Recursive call ended.` The program then goes to line 11 and `walk5` ends. Then the control is returned to `walk4` where the program continues on line 10, and displays for the second time the message `Recursive call ended.`

This process continues until `walk2` ends and the control is returned to the original command `walk`. When this command ends, control is returned to the main program, and the last message is displayed: `Call from the main program ended.` Here is the sequence of the recursive calls in graphic form:



The sequence of recursive calls.

A recursive call creates and calls a new copy of the command (or function). The original command is then put on hold until the call to the copy ends.

14.5. Forgetting the stopping condition

Forgetting to place the recursive call in the body of a stopping condition leads to an infinite loop - the recursion never stops. Let's illustrate this by removing the stopping condition from the previous program:

```
# Recursive command:
def walk
  if shield
    get
  go
  print("Recursive call...")
  walk
  print("Recursive call ended.")
  return

# Main program:
print("Calling 'walk' from the main program...")
walk
print("Call from the main program ended.")
```

Here is the corresponding output:

```
Calling 'walk' from the main program...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
Recursive call...
```

The infinite recursion was actually interrupted when Karel stepped out of the maze and the program crashed:

```
Run-time error on line 5:
Can't go there!
```

Infinite recursion may be interrupted with
an error which causes the program to crash.

14.6. Moving shields to boxes

Today Karel needs to use recursion to put the shields in the boxes, instead of just collecting them:



Move the shields to the boxes!

The program is similar to the previous one. We only removed the control outputs and added a new condition on lines 5 and 6:

```
1  # Recursive command:
2  def walk
3    if shield
4      get
5    if box
6      put
7    go
8    # Stopping condition:
9    if not home
10     walk
11   return
12
13 # Main program:
14 walk
```

Again, notice that the recursive call on line 10 is made from the body of a stopping condition `if not home` which guarantees that the recursion stops when the robot gets to his home square.

14.7. Museum heist

Someone stole from the museum historical items of immense value, and left some shields lying scattered on the floor. Karel needs to sweep the floor for any traces, collect the shields, and bring them to the home square:



Sweep the floor and collect all shields!

Here is the corresponding program. You will recognize the First Maze Algorithm on lines 5 - 10:

```
1  # Recursive command:
2  def sweep
3    if shield
4      get
5    if wall
6      left
7      if wall
8        right
9        right
10   go
11   # Stopping condition:
12   if not home
13     sweep
14   return
15
16 # Main program:
17 sweep
```

14.8. Mutually recursive commands

In recursion, a command or function does not always have to call itself. Recursion also occurs when command A calls command B, then command B calls command A, then

command A calls command B, etc. Let's illustrate this on the previous example, where we decompose the recursive command `sweep` into a pair or mutually recursive commands `move` and `collect`:

```
# Recursive command:
def move
  if wall
    left
    if wall
      right
      right
  go
  # Stopping condition:
  if not home
    collect
  return

# Recursive command:
def collect
  if shield
    get
  # Stopping condition:
  if not home
    move
  return


# Main program:
move
```

Importantly, notice that in both commands, the recursive call occurs in a stopping condition.

14.9. Using variables and functions in recursion

So far we have discussed recursive commands which did not involve variables. Now we will show you that variables and functions can be used in recursion as well.

For illustration, let's write a recursive function `sum(n)` to add the numbers `n`, `n-1`, `n-2`, `...`, `1` and return the result. The idea is that if `n > 1` then the function will call `s = sum(n-1)` and return `s + n`. If `n == 1` then the function will return 1:

```
1  # Recursive function to add numbers n, n-1, ..., 1:
2  def sum(n)
3    # Stopping condition:
4    if n > 1
5      # Calculate the rest of the sum:
6      print("Calling 'sum' from 'sum' with argument", n-1)
7      s = sum(n-1)
8      # Add n to the rest of the sum:
9      print("Adding", n)
10     inc(s, n)
11     # Return the sum:
12     return s
13   # Else branch of the stopping condition:
14   else
15     # Return 1:
16     print("Recursion ended, returning 1")
17     return 1
18
19 # Print result for a sample value 10:
20 print("Calling 'sum' from main program with argument 10")
21 num = sum(10)
22 print("Final result =", num)
```

The recursive call on line 7 evaluates the sum of the numbers `n-1`, `n-2`, `...`, `1` to which is then added `n` on line 10, and the result is then returned on line 12.

Notice that the stopping condition has the form `if n > 1` which ensures that the recursion stops when `n` reaches 1. The function contains some control prints to help us better understand the program flow. This is the output which shows that everything went as expected:

```
Calling 'sum' from main program with argument 10
Calling 'sum' from 'sum' with argument 9
Calling 'sum' from 'sum' with argument 8
Calling 'sum' from 'sum' with argument 7
Calling 'sum' from 'sum' with argument 6
Calling 'sum' from 'sum' with argument 5
Calling 'sum' from 'sum' with argument 4
Calling 'sum' from 'sum' with argument 3
Calling 'sum' from 'sum' with argument 2
Calling 'sum' from 'sum' with argument 1
Recursion ended, returning 1
Adding 2
Adding 3
Adding 4
Adding 5
Adding 6
Adding 7
Adding 8
Adding 9
Adding 10
Final result = 55
```

14.10. Parsing lists using recursion

Today, Karel is given a list of text strings. His task is to parse the list without using a loop, and display all its items. This is very easy indeed:

```
1  # Sample list:
2  days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
3
4  # Recursive function to parse a list:
5  def parselist(L)
6    if len(L) != 0
7      print(L[0])
8      parselist(L[1:])
9    return
10
11 # Main program:
12 parselist(days)
```

Output:

```
Mon
Tue
Wed
Thu
Fri
Sat
Sun
```

That was too simple, so here is one more task: Write a recursive function `add(L)` to add all items in a list `L`! But you already understand recursion really well, so this can't throw you off balance:

```
1  # Sample list:
2  nums = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
3
4  # Recursive function to add all items in a list:
5  def add(L)
6    if len(L) > 0
7      s = add(L[1:])
8      return s + L[0]
9    return 0
10
11 # Main program:
12 result = add(nums)
13 print("The result is:", result)
```

The result is 110. Add the numbers yourself to verify it's true!

14.11. Recursion at its best

All the examples that we have presented so far were easily solvable without recursion as well. So, you might get an impression that *all tasks* which can be solved with recursion can easily be solved without it. But that would not be true. The problem with demonstrating the real advantages of recursion is that one needs more difficult tasks for that.

Therefore, we prepared for you three such tasks. All of them can be solved easily with recursion but they would be much more difficult to tackle without it. You will find them in Section 15.1 (page 258), Section 15.2 (page 260) and Section 15.3 (page 262).

14.12. Review questions

Friendly reminder - for every question either none, one, or several answers may be correct.

QUESTION 14.1. *In which areas can one encounter recursion?*

A Art
B Nature
C Mathematics
D Computer programming

QUESTION 14.2. *Which of the following activities can be viewed as recursion?*

A Eating your lunch.
B Cleaning your room.
C Reading a book.
D Counting down from 10 to 1.

QUESTION 14.3. *What is characteristic for tasks which are suitable for recursion?*

A They are repetitive.
B They require variables.
C They require lists.
D They can be reduced to the same task, but smaller in size.

QUESTION 14.4. *Where should the recursive call be made?*

A At the beginning of the command or function.
B At the end of the command or function.
C From the body of a stopping condition.
D Before a stopping condition.

QUESTION 14.5. *What happens if one forgets the stopping condition?*

A The program will run forever or crash.
B Recursion will turn into an infinite loop.
C The recursive call will not be executed.
D The recursive call will be only made once.

QUESTION 14.6. *What does it mean that two commands A and B and mutually recursive?*

A A is recursive and B is not.
B B is recursive and A is not.
C A calls B and B calls A.
D A calls itself and B calls itself.

15. Advanced Applications

The objective of this section is to present a few advanced applications of various techniques and algorithms which you have learned so far. We will begin with recursion.

15.1. Flood (recursion)

Karel's first task is to "flood" the maze with beepers. This means - place a beeper in every grid square he can access. The layout of the maze is random, but there are no other obstacles besides walls, no containers, and no collectible objects. Notice that the robot is not able to access all grid squares because some areas in the maze are surrounded with walls:



Karel is going to put beepers in all accessible squares.

This task is perfect for recursion, and in a moment we are going to show you why. But first a technicality - let's define a new command `back` for Karel to back one step:

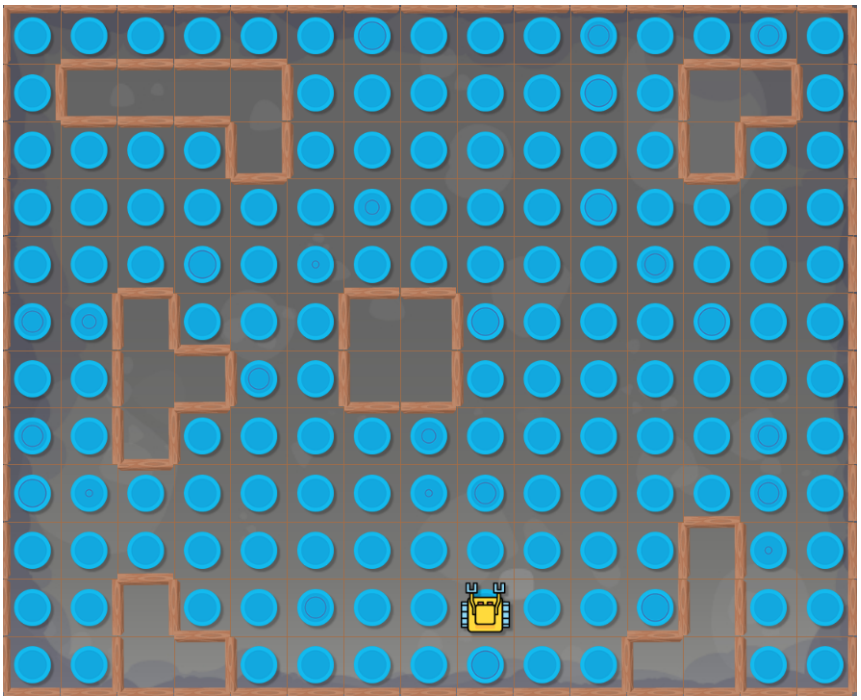
```
# Back one step:
def back
  left
  left
  go
  right
  right
```

Then we can write the recursive command `fill`. The idea is as follows: If you stand on a beeper, the grid square was already "flooded". So do nothing and just return (this is when the recursion will stop). Otherwise (there is no beeper where you stand), place a beeper. Then, you will step one by one in the squares on your right, in front of you, and on your left, and always start a new "flood" from there. Of course you always need to check for a wall before making a step forward. Amazingly, this is it! Below is the code of the command `fill`:

```
# Put a beeper in every accessible square:
def fill
  if beeper
    return
  else
    put
  right
  repeat 3
    if not wall
      go
      fill
      back
    left
  right
  right

# Main program:
fill
```

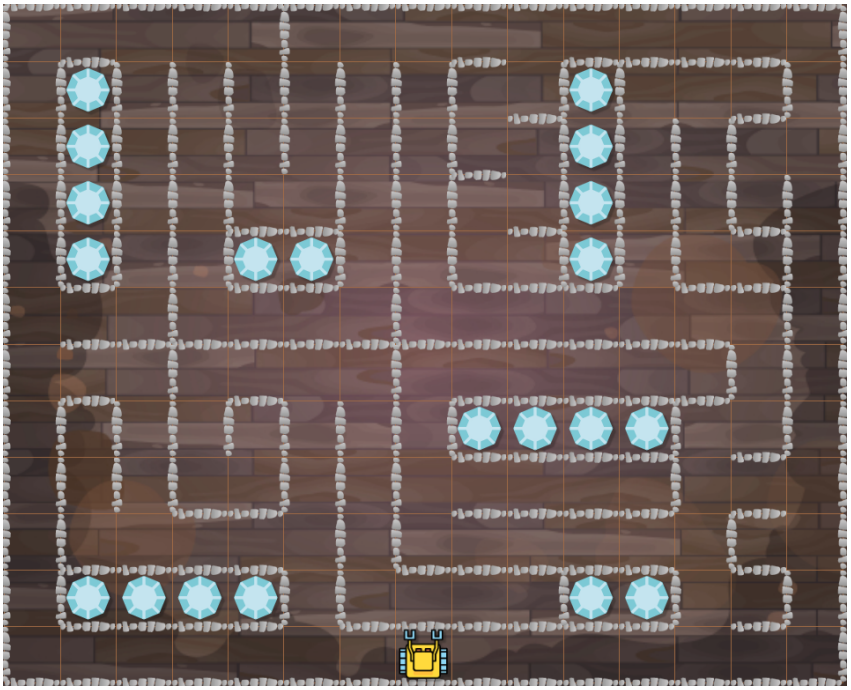
Before executing the program, one has to ensure that the robot has enough beepers in his bag. Since the maze has 15 columns and 12 rows, the best is to give him $12 \cdot 15 = 180$ beepers. After the program ends, the maze will look as follows:



At the end, the maze is "flooded".

15.2. Contraband (recursion)

An ancient story has it that robbers hid gems in this maze.



Sample maze with an unknown number of hidden gems.

If there were any, they would be in unreachable squares. Hence, your task is to calculate the number of unreachable squares!

One way to do this, is to "flood" the maze as we did in the previous section. When the robot begins with $12 \cdot 15 = 180$ beepers in his bag, and finishes with `B` beepers, then there are `180 - B` unreachable squares. Here is the corresponding program:

```
# Return one step back:
def back
  left
  left
  go
  right
  right

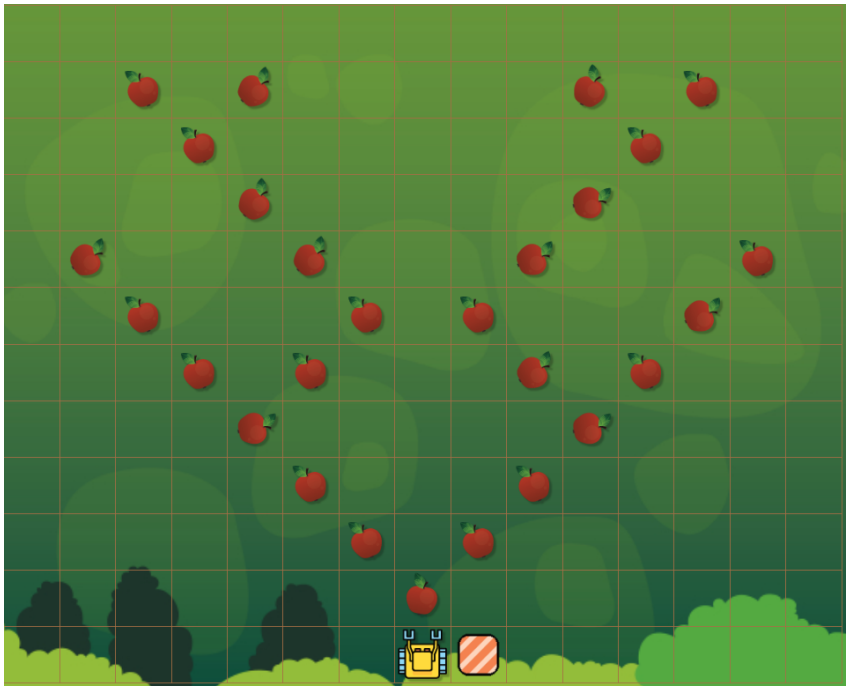
# Put a beeper in every accessible square:
def fill
  if beeper
    return
  else
    put
  right
  repeat 3
    if not wall
      go
      fill
      back
    left
  right
  right

# Main program:
N = 180
# Place beepers into all accessible squares:
fill
# How many beepers were you not able to place?
while not empty
  put
U = -1
while beeper
```

```
34      get
35      inc(U)
36  print("There are", U, "hidden gems in this maze!")
```

15.3. Traversing binary trees (recursion)

Binary trees are an important data structure in computer science, and traversing them is one of the fundamental tasks. In Karel, binary trees can be represented by trees of apples:



Sample binary tree.

Why do we use the word "binary"? The reason is that at any position in the tree, there can be at most two branches - to the left and/or to the right.

The idea of the recursive algorithm is that at each position in the tree there can be at most two branches, which themselves are binary trees. Hence, Karel first needs to check for an apple beneath him. If there is no apple, the recursion stops. Otherwise (there is an apple beneath him), he needs to check whether a tree starts in the Northwest square. If so, he needs to traverse it and return to his initial position. Then he also needs to check if a tree starts in the Northeast square. If so, he needs to traverse it and return to his initial position. Finally, Karel needs to collect the apple beneath him.

The corresponding code can be found below:

```
# Check both branches:
def collect
  go
  left
  go
  right
  if apple    # check Northwest square
    collect
  else
    right
    right
  left
  go
  go
  left
  if apple    # check Northeast square
    collect
  else
    left
    left
  right
  go
  left
  go
  if apple    # check for apple at the initial position
    get

# Main program:
go
collect
go
left
go
```

A short video of Karel performing this program is available at `https://youtu.be/Z659mYXHWuc`.

15.4. Bubble sort

Sorting algorithms constitute another important chapter of computer science. Bubble sort

is not the most sophisticated one, but it is great as a stepping stone before learning QuickSort and other more advanced sorting algorithms.

In Karel, a sequence of numbers could be represented by a list. But that would not be fun. So, let's represent the numbers visually as piles of beepers. Of course the sequence is random. Let's say that it contains 6 numbers:



Random initial sequence.

The idea of the bubble sort algorithm is simple: Karel passes through the sequence from left to right, and whenever he sees that the next number is less than the current one, he switches them.

Of course, it is not sufficient to only do this once, because the sequence `2, 1, 1, 1` would just become `1, 2, 1, 1` which is not sorted yet. When the sequence has `N` numbers, Karel needs to make `N-1` passes. Here is the corresponding program:

PROGRAM 15.1. Bubble sort

```
# Number of piles:
n = 6

# Place p objects:
def place(p)
  repeat p
    put

# Move back n steps:
def back(s)
  left
  left
  repeat n
    go
  right
  right

# Collect all beepers in the pile and return their number:
def getall
```

```
20    c = 0
21    while beeper
22      get
23      inc(c)
24    return c
25
26  # Main program:
27  done = False
28  dec(n)
29  while not done
30    go
31    a = getall
32    done = True
33    repeat n
34      go
35      b = getall
36      back(1)
37      if a > b
38        done = False
39        place(b)
40      else
41        place(a)
42        a = b
43      go
44    while not empty
45      put
46    back(n+1)
47    dec(n)
48  print("Done!")
```

In the 1st cycle, Karel switches (5, 2) → (2, 5), (6, 3) → (3, 6), (6, 4) → (4, 6) and (6, 1) → (1, 6):



After the 1st cycle.

In the 2nd cycle, he switches (5, 3) → (3, 5), (5, 4) → (4, 5), (5, 4) → (4, 5) and (5, 1) → (1, 5):



After the 2nd cycle.

In the 3rd cycle, he only switches (4, 1) $\rightarrow$ (1, 4):



After the 3rd cycle.

In the 4th cycle, he only switches (3, 1) $\rightarrow$ (1, 3):



After the 4th cycle.

In the 5th cycle, he only switches (2, 1) $\rightarrow$ (1, 2), and he is done:

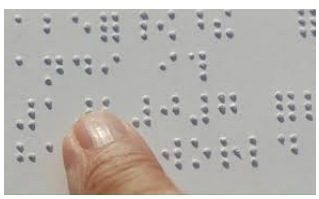


After the 5th cycle, the sequence is sorted.

A short video of Karel performing this program is available at `https://youtu.be/b2_YzJscFuo`.

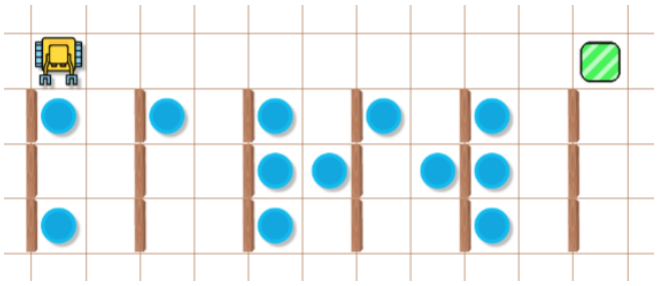
15.5. Reading Braille

Braille is a system of raised dots that can be read with the fingers by people who are blind or who have low vision:



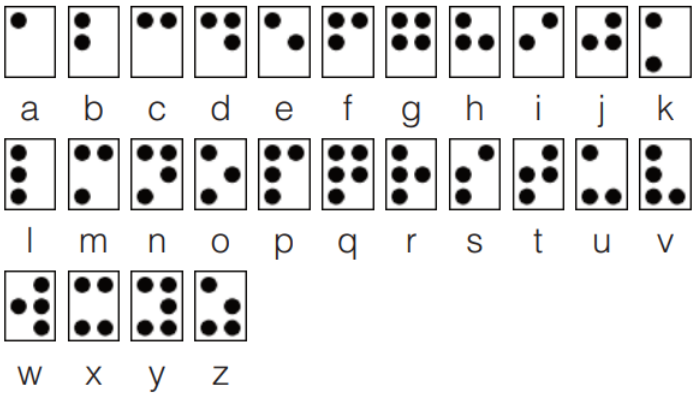
Reading Braille text.

Teachers, parents, and others who are not visually impaired ordinarily read Braille with their eyes. Braille is not a language. Rather, it is a code analogous to the Morse code. Each letter is formed by 1 - 6 dots in a box of width 2 and height 3. For illustration, here is the word "Karel" in Braille:



Karel's own name in Braille.

For reference, this is the complete English alphabet in Braille:



All 26 English letters in Braille.

The algorithm is straightforward: First we write a function `braillenum` to translate the Braille symbols to numerical codes. To do this, we enumerate the six positions in the 2×3 box by 1, 2, 3 (left column, top to bottom) and by 4, 5, 6 (right column, top to bottom). Then for each Braille letter, we will use these numbers to create a text string. For each dot, the text string will contain the corresponding number. For example, `'a'` is `'1'`, `'b'` is `'12'`, `'c'` is `'14'` etc.

PROGRAM 15.2. Reading Braille text - part 1

```
1 # Translate one letter into numbers:
2 def braillenum
```

```
  s = ''
  go
  if beeper
    s += '1'
  go
  if beeper
    s += '2'
  go
  if beeper
    s += '3'
  left
  go
  left
  if beeper
    s6 = 1
  else
    s6 = 0
  go
  if beeper
    s5 = 1
  else
    s5 = 0
  go
  if beeper
    s4 = 1
  else
    s4 = 0
  if s4 == 1
    s += '4'
  if s5 == 1
    s += '5'
  if s6 == 1
    s += '6'
  repeat 2
    go
    right
  return s
```

Then we write the resulting function `braille` to translate the numerical codes to English letters. Actually this is just one long `if-elif-else` statement:

PROGRAM 15.3. Reading Braille text - part 2

```
# Read one letter as a string and analyze it:
def braille
  b = braillenum
  if b == '1'
    return 'a'
  elif b == '12'
    return 'b'
  elif b == '14'
    return 'c'
  elif b == '145'
    return 'd'
  elif b == '15'
    return 'e'
  elif b == '124'
    return 'f'
  elif b == '1245'
    return 'g'
  elif b == '125'
    return 'h'
  elif b == '24'
    return 'i'
  elif b == '245'
    return 'j'
  elif b == '13'
    return 'k'
  elif b == '123'
    return 'l'
  elif b == '134'
    return 'm'
  elif b == '1345'
    return 'n'
  elif b == '135'
    return 'o'
  elif b == '1234'
    return 'p'
```

```
  elif b == '12345'
    return 'q'
  elif b == '1235'
    return 'r'
  elif b == '234'
    return 's'
  elif b == '2345'
    return 't'
  elif b == '136'
    return 'u'
  elif b == '1236'
    return 'v'
  elif b == '2456'
    return 'w'
  elif b == '1346'
    return 'x'
  elif b == '13456'
    return 'y'
  elif b == '1356'
    return 'z'
  else
    return '?'

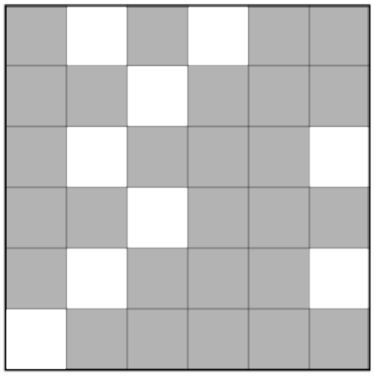
# Main program:
word = ''
while not home
  word += braille
print(word)
```

15.6. Cardan grille

The Cardan grille is an old encryption technique which assumes that the text is written in a square table. The size of the table corresponds to the number of letters. For example, a 5×5 table is enough for a 25-letter text. A table of the size 8×8 can be used for a text of up to 64 letters.

The grille itself is a square cloth (or paper) whose size matches the size of the table of letters. The cloth contains holes so that when laid over the text, certain letters appear. The following example shows a 6×6 table of letters, and a grille which when laid over the text reveals "JULESVERN".

E	J	E	U	S	T
Y	G	L	É	Á	N
Á	E	Y	S	É	S
C	N	V	D	Í	B
O	E	R	M	M	R
N	E	Ű	N	R	Á



Sample Cardan grille.

The sender and the addressee each have a copy of the grille, and the grille never travels along with the encrypted text message.

After the addressee reads the first group of letters, s/he rotates the grille by 90 degrees and reads part 2. Then s/he rotates the grille by another 90 degrees and reads part 3. And finally, s/he rotates the grille one last time and reads part 4.

If the grille is designed well, then rotating it each time reveals a different group of letters. However, if the grille is flawed, it can happen that rotating the grille reveals a position which was already revealed before. This problem is present in the following grille (where the gold coins represent the holes). Can you find it?



Flawed Cardan grille.

On the other hand, the following grille is designed correctly and the above problem does not occur:



Well-designed Cardan grille.

Karel's task is to analyse a given 6×6 Cardan grille and count the flaws. A zero result means that the grille is well designed.

The algorithm is as follows: As a first step, we will write a function `findholes` to locate all holes and store their GPS coordinates (relative to the bottom-left corner of the grille) in a helper list `H`.

PROGRAM 15.4. Checking a Cardan grille - part 1

```
1  # Create a list of the GPS coordinates of all holes. Lower left corner of
       grille is [0, 0]:
2  def findholes
3    holes = []
4    repeat 6
5      repeat 6
6        go
7        if coin
8          holes.append([gpsx - 5, gpsy - 3])
9      left
10     left
11     repeat 6
12       go
13     right
14     go
15     right
16   return holes
```

Next, we will write a Boolean function `check(i, j, L)` to check whether the list `L` contains the position `[i, j]` rotated by 90, 180 or 270 degrees.

PROGRAM 15.5. Checking a Cardan grille - part 2

```
# Return True if position [i, j], rotated by 0, 90, 180 and 270 degrees, is
# (a) not found or (b) is found more than once in the list L.
# Otherwise return 0:
def check(i, j, L)
  count = 0
  if [i, j] in L
    inc(count)
  if [5-j, i] in L
    inc(count)
  if [5-i, 5-j] in L
    inc(count)
  if [j, 5-i] in L
    inc(count)
  if count == 0 or count > 1
    return True
  return False
```

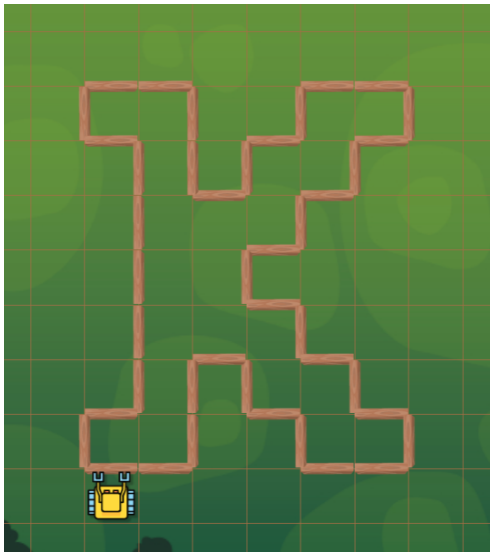
And in the main program we will check all nine positions in the lower-left 3×3 quadrant of the 6×6 grille:

PROGRAM 15.6. Checking a Cardan grille - part 3

```
# Main program:
H = findholes
res = 0
l = [0, 1, 2]
for ipos in l
  for jpos in l
    f = check(ipos, jpos, H)
    if f == 1
      print("Problem found at", ipos, jpos)
    inc(res, f)
print("Flaws found:", res)
```

15.7. Land surveyor

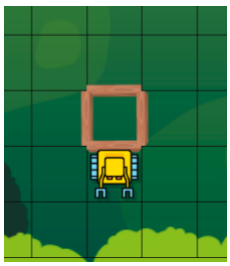
Karel works as a land surveyor. Today he needs to measure the area of a fenced garden.



Sample fenced area.

But he cannot enter because there is a big dog inside, and Karel is afraid of dogs. Fortunately, he knows how this can be done without entering the garden!

Let's demonstrate his method on a small one-square garden shown below. Karel creates a new variable `A = 0` for the area and walks along the fence. After each step, he positions himself to face away from the fence. Depending whether he is facing South, East, North or West, he does the following:



If you face South, decrease `A` by `gpsy+1`.

Since his `gpsy` is 1 and `A` was 0, after this operation `A` will be `-2`.

Here Karel's `gpsy` is 3. Adding 3 to `A` will change the value of `A` to 1:



If you face North, increase `A` by `gpsy`.

If Karel faces East or West, the value of `A` will not change:





If you face East or West, do not change `A`.

Hence Karel correctly calculated that the area `A` of this mini-garden is 1.

Below is the corresponding program which moreover uses the Second Maze Algorithm to walk around the garden. Karel stores his initial GPS coordinates so that he knows where to stop:

```
# Are you facing South?
def south
  left
  left
  s = north
  right
  right
  return s

# Remember GPS coordinates:
x = gpsx
y = gpsy
# Variable A for the area:
A = 0
```

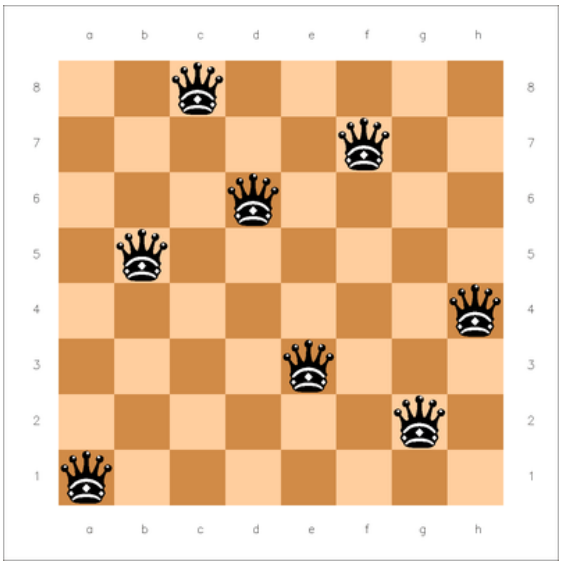
```
# Make the first step:
while wall
  if north
    dec(A, gpsy)
  if south
    inc(A, gpsy)
    dec(A)
  right
go
# Walk around fence:
while (x != gpsx) or (y != gpsy)
  left
  while wall
    if north
      dec(A, gpsy)
    if south
      inc(A, gpsy)
      dec(A)
    right
  go
# Print the result:
print("Area =", A)
```

For the K-shaped garden shown at the beginning of this section, Karel obtains:

```
Area = 22
```

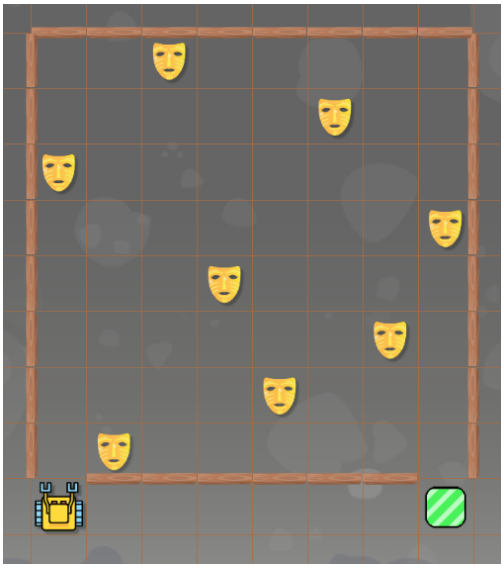
15.8. The Eight Queens puzzle

This is a classical puzzle whose objective is to place eight queens on the chess board so that they do not threaten each other. Recall that the queen is the most powerful figure on the chess board - she dominates the column, the row, and both diagonals from where she stands. In other words, no two queens can stand in the same row, in the same column, or on the same diagonal. Below is a sample solution:



One of 92 existing solutions.

The queens will be represented by masks. Karel's task is to analyse the positions of the eight queens and determine whether or not some of them threaten each other.



Queens are represented by masks.

The task will be solved in two steps. First, the function `sweep` will be used to go through the entire chess board, locate all queens, and store their GPS coordinates in a list:

```
# Add the GPS positions of all queens to a list:
def sweep
  q = []
  while gpsx != 10 or gpsy != 2
    go
    if mask
      q.append([gpsx, gpsy])
    if wall
      if north
        right
        go
        if mask
          q.append([gpsx, gpsy])
        right
      else
        left
        go
        if mask
          q.append([gpsx, gpsy])
        left
  return q
```

In the next step, this list will be analysed. If Karel finds two pairs of coordinates which have the same X component and a different Y component, then he knows he found two queens in the same column. If he finds two pairs of coordinates which have the same Y component and a different X component, then he knows that he found two queens in the same row.

How can he check the diagonals? Notice that positions on the same diagonal, such as [1, 6], [2, 5], [3, 4], 4, 3, [5, 2], [6, 1] satisfy that the sum of the X and Y coordinates is constant (in this case 7). This is true for all diagonals which go from from Northwest to Southeast, just the constants differ. For the remaining diagonals which go from from Southwest to Northeast, such as [4, 1], [5, 2], [6, 3], [7, 4] etc. it holds that the difference Y - X is constant (in this case -3). Again the constant differs from one diagonal to another.

In the function `analyze` below, Karel considers all 8 rows, all 8 columns, and all NW-SE and SW-NE diagonals. In each case he is looking into the list `L` to see whether two (or more) queens are there, threatening each other:

```
# Analyze the list:
def analyze(L)
  # We will go through the list once, counting queens in every column, row,
      and diagonal:
  cols = [0, 0, 0, 0, 0, 0, 0, 0]
  rows = [0, 0, 0, 0, 0, 0, 0, 0]
  dia1 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  dia2 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  for pos in L
    posx = pos[0] - 3
    posy = pos[1] - 2
    cols[posx] += 1
    rows[posy] += 1
    dia1[posx + posy] += 1
    dia2[posy - posx + 7] += 1
  print(cols)
  print(rows)
  print(dia1)
  print(dia2)
  # Check if there is more than one queen in some column, row, or diagonal:
  success = True
  for x in cols
    if x == 0
      print("No queen found in column", x)
      success = False
    if x > 1
      print("More than one queen found in column", x)
      success = False
  for y in rows
    if y == 0
      print("No queen found in row", y)
      success = False
    if y > 1
      print("More than one queen found in row", y)
      success = False
  for n in dia1
    if n > 1
      print("More than one queen found on a NW-SE diagonal.")
      success = False
```

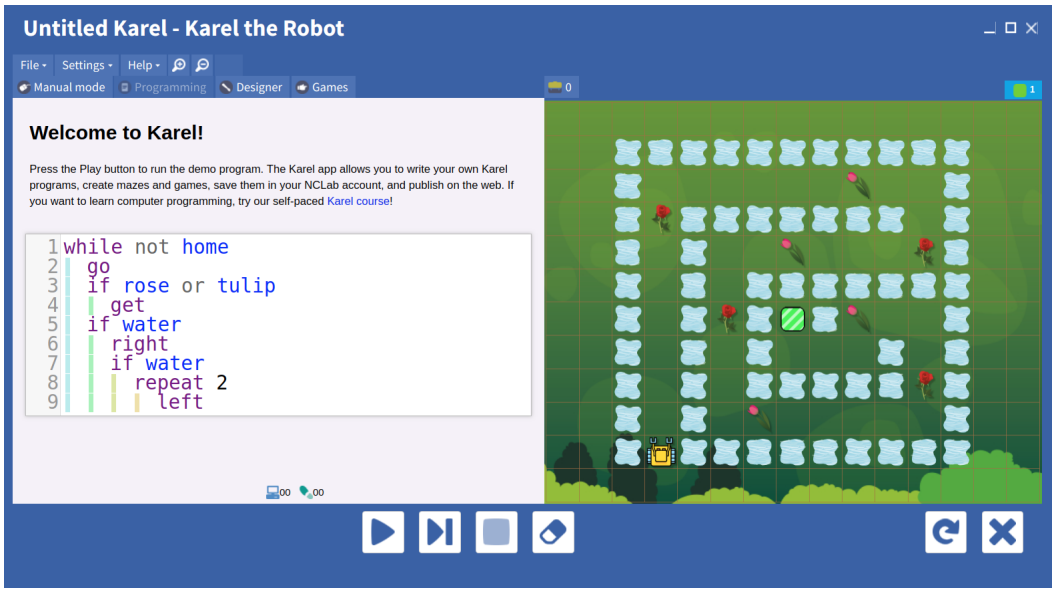
```
39    for n in dia2
40      if n > 1
41        print("More than one queen found on a SW-NE diagonal.")
42        success = False
43    return success
44
45  # Main program:
46  Q = sweep
47  ok = analyze(Q)
48  if ok
49    print("No threats found.")
50  else
51    print("Threats found!")
52  go
```

A. Karel App in NCLab

The Karel app is located in the Programming section of the Creative Suite on NCLab's Desktop. It allows you to create your own mazes and games, write your own programs, save them in your NCLab user account, and publish them online and share with others. In this chapter, we will show you how to launch the Karel application and how to use it.

A.1. Launching the Karel app

The Karel app opens with a demo program:



The Karel app in programming mode, with a demo program.

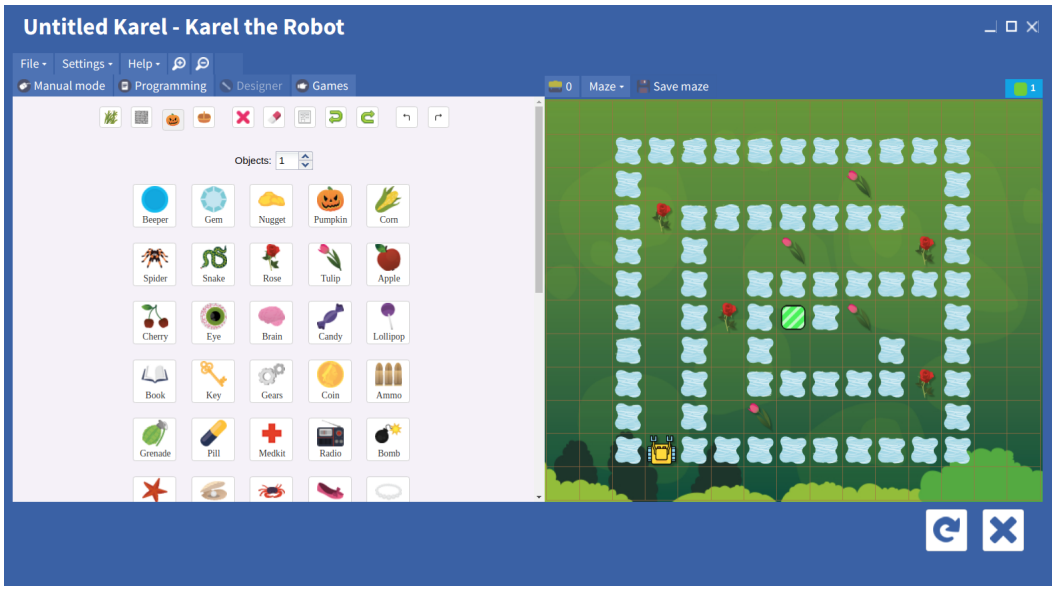
The top menu bar includes three items: File, Settings, and Help along with two zoom icons. The File menu allows you to open a new instance of the application, open an existing file, save the maze to a file, etc. The Settings menu allows you to change the robot's speed, show/hide the status bar, highlight the active line of the code, etc. The status bar is located on the bottom of the window and it displays important information about the status of the robot.

The Karel app has 4 modes: Manual, Programming, Designer, and Games. The Manual mode allows the user to guide the robot using the mouse or keyboard. For more information about the Manual mode visit Section 2.1 on page 8.

The Programming mode makes it possible to write, execute, and debug programs for Karel. For more details about the Programming mode visit Section 2.5 on page 11.

A.2. Building mazes

The Designer mode allows you to create your own mazes, save them in your NCLab user account, and to share them with others online. This is how the Karel app looks in Designer mode:



The Karel app in Designer mode.

The Designer has its own menu on top:

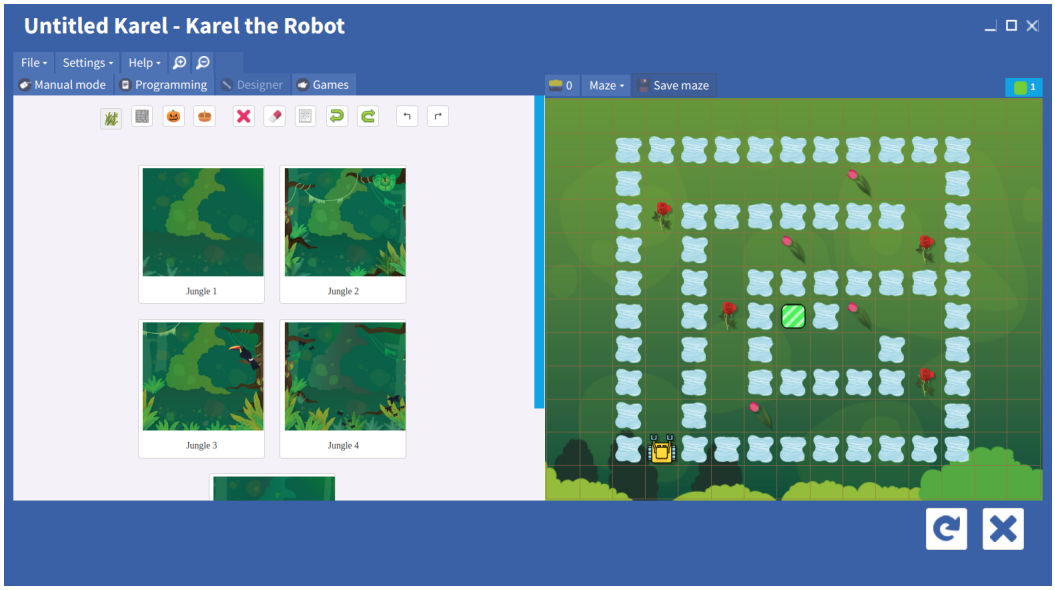


Designer menu.

From left to right, the functions of the menu buttons are as follows:

(1) *Themes:* The Designer offers over 20 different themes including Zoo, Garden, Mines, Desert, Jungle, Factory, Mountains, etc. In each theme, there are several

wallpapers to choose from. For example, five wallpapers are available in the Jungle theme. Clicking on a wallpaper will install it to the maze:



Jungle theme wallpapers.

(2) *Obstacles:* The Obstacles button is the second one from the left in the Designer menu, and it has a grey wall on it. There are more than 40 different obstacles including wall, fire, water, stone, acid, etc. To install an obstacle into the maze, click on it and then click into the maze. There are two types of obstacles: thin (one-dimensional) and thick (two-dimensional). The thin ones occupy an edge of a square. The thick ones take an entire grid square. There are 21 different types of walls.

(3) *Collectible objects:* Over 60 different types of collectible objects are available under the pumpkin button (third one from the left).

(4) *Containers:* Containers (the fourth button from the left with a basket on it) are objects which can store other objects in them. When placing a container, one can specify its capacity as well as the types of objects that it admits. Karel has 14 types of containers: mark, target (target is invisible to the user, mark is visible), box, bag, basket, mining cart, etc.

(5) *Erase mode:* The button with the red mark "X" on it is used to remove individual objects, containers, or obstacles from the maze. If this button is toggled, one can move the mouse over the maze and start removing objects. One can remove everything except for the robot and his home square. The home square can be hidden in the Maze menu which is located above the maze.

(6) *Erase all:* Pressing the button with the eraser mark on it will instantly remove all objects, containers and obstacles from the maze. This comes handy when one is starting to design a completely new maze.
(7) *Random mode:* When this button is toggled, obstacles, collectible objects, and containers will be added to random positions in the maze as one clicks on them in the left panel.
(8) *Undo:* This is the fourth button from the right. The Designer remembers all your actions (since the app was last launched), so all of them can be undone using this button.
(9) *Redo:* If you have used the Undo button, then this button will undo the undo.
(10) *Left:* The left arrow button will turn Karel left 90 degrees.
(11) *Right:* The right arrow button will turn Karel right 90 degrees.

Besides this, one can use the mouse to drag and drop the robot, the home square, as well as all the obstacles, collectible objects, and containers.

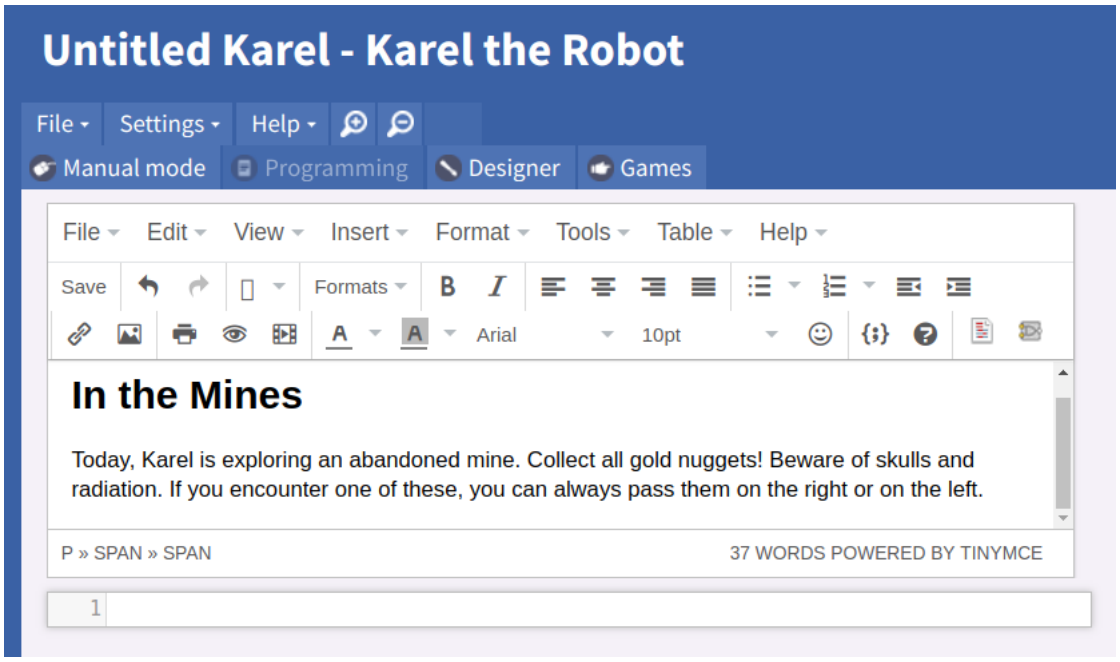
When the maze has unsaved changes, the button "Save maze" will start flashing, reminding you to click on it and save maze changes. If the maze hasn't been saved yet, then the File Manager will launch, asking you to choose a folder and filename. Otherwise, the maze will just be saved to the existing file and the button will stop flashing.

A.3. Create your own game!

Let's create a game! First of all, you need to figure out a story and a task for Karel. Today, Karel will be exploring an abandoned gold mine and collecting nuggets. However, he must beware of skulls and radiation! If comes across them, he must pass them either on the right or on the left.

The first step in creating a game is to launch the Karel app and write the story. For this, click on the welcome message of the Karel app. The text will open in a WYSIWYG text editor. The editor allows you to insert and remove text, change text size, font and color, insert images, code snippets, and even videos, and do many other things.

Although sometimes one is tempted to write a long and intricate story, keep in mind that the story is not the main objective of the game. Make it short and sweet. And most importantly - *precise*. It should be perfectly clear to the users what the game goal is.



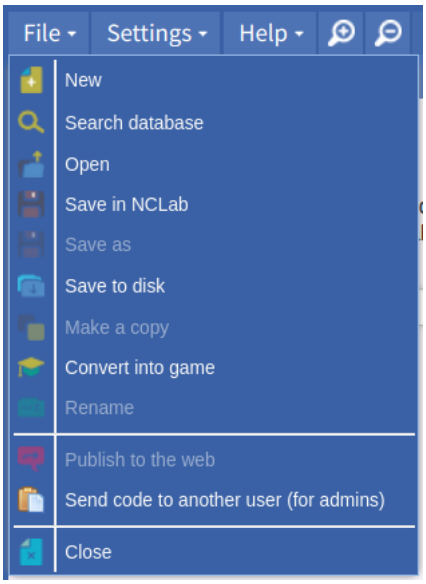
Writing the story in the WYSIWYG text editor.

When you are satisfied with the story, hit "Save" in the upper left part of the text editor. This will save the text and exit the editor.



After exiting the editor, the story is ready.

At this point you can still lose all your changes (for example, if your computer gets disconnected from the Internet). So, let's save the file in your NCLab account. This can be done by clicking on "Save in NCLab" in the File menu:



Saving the file in your NCLab account.

This will launch the File Manager, and you will be able to enter the file name as well as select the folder where your game should be saved. We saved the file under the name "In the Mines":



When the file is saved, the name is displayed in the header.

The next step is to build a maze to match the story. You already know from Section A.2 how to do that, so we will leave this step to you. Try to come up with your own maze! The maze we created is shown below:



The Designer is used to build the maze for the story.

When your maze is finished, do not forget to click on the blinking button "Save maze" before leaving the Designer.

A.4. Check how difficult your game is

It is very easy to build a maze which is extremely difficult or impossible to solve. Therefore, at this point, it is important that you try and solve the maze by yourself. When you plan for your game to be in manual mode, switch to Manual mode and solve it. When your game will be in programming mode, then switch to Programming mode and write the corresponding program. Here is our program which is based on the First Maze Algorithm, enhanced with additional tests for skulls and radiation:

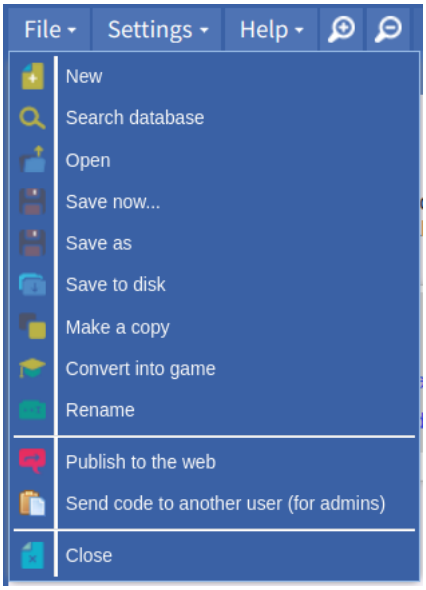
```
1 while not home
2   go
3   if nugget
4     get
5   if wall or skull or radiation
6     left
```

```
7     if wall or skull or radiation
8       right
9       right
```

That was not too difficult, was it? Nine lines is a short code. But to achieve this most elegant solution, the user needs to realize that the skulls and radiation should be treated just as additional types of walls. Not everybody will do that - most frequently the users will try to handle the walls and the remaining obstacles separately. Then their code will be longer, and they should get fewer points for that. This brings us to the next step - defining game goals.

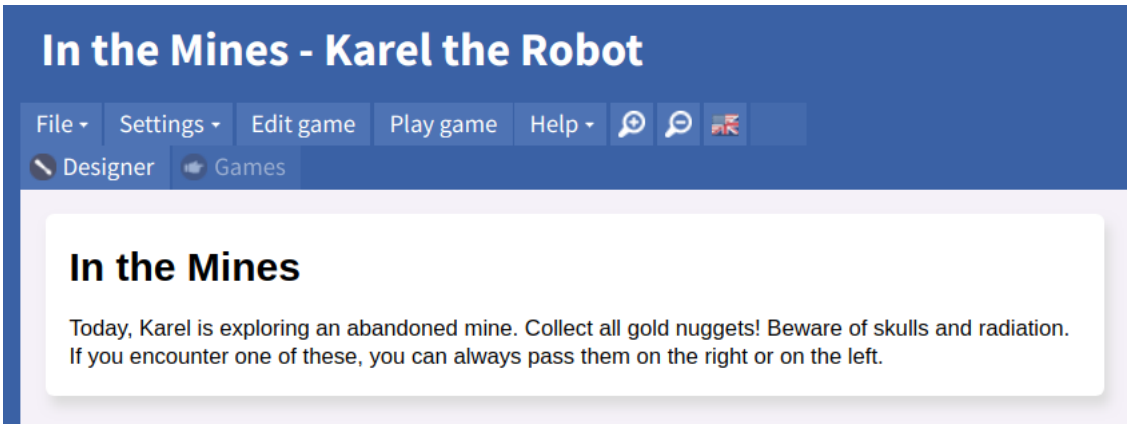
A.5. Convert worksheet into a game

In order to define game goals, we need to convert the worksheet into a game first. This is done by clicking on "Convert into game" in the File menu:



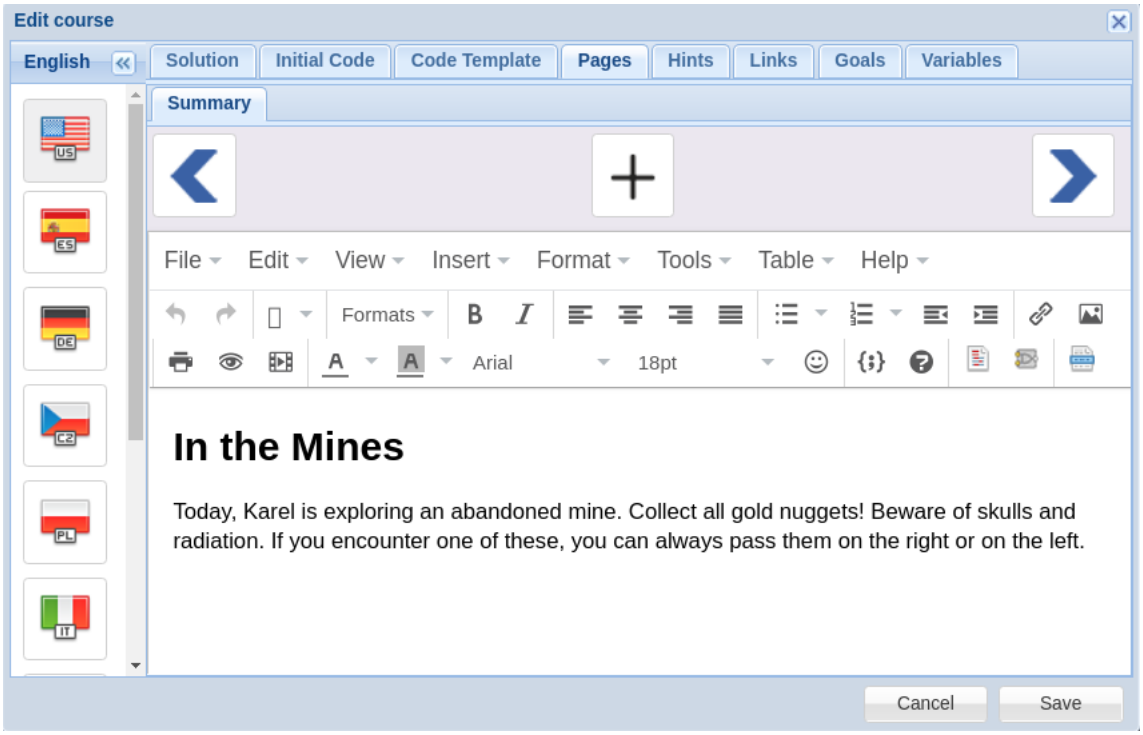
Converting the worksheet into a game.

This is an irreversible change, so be sure that you want to do this. The menu in the upper part of the window will change:



After converting into a game, the upper menu changes.

The button "Play game" can be used to play the game as if you were the user. But we are not there yet. First, we need to define game goals. For this, click on "Edit game". Then the following menu will appear:



Edit game menu.

In the column on the left you will see many different flags. Each flag represents one language - in case you want to have your game translated into several languages. The top

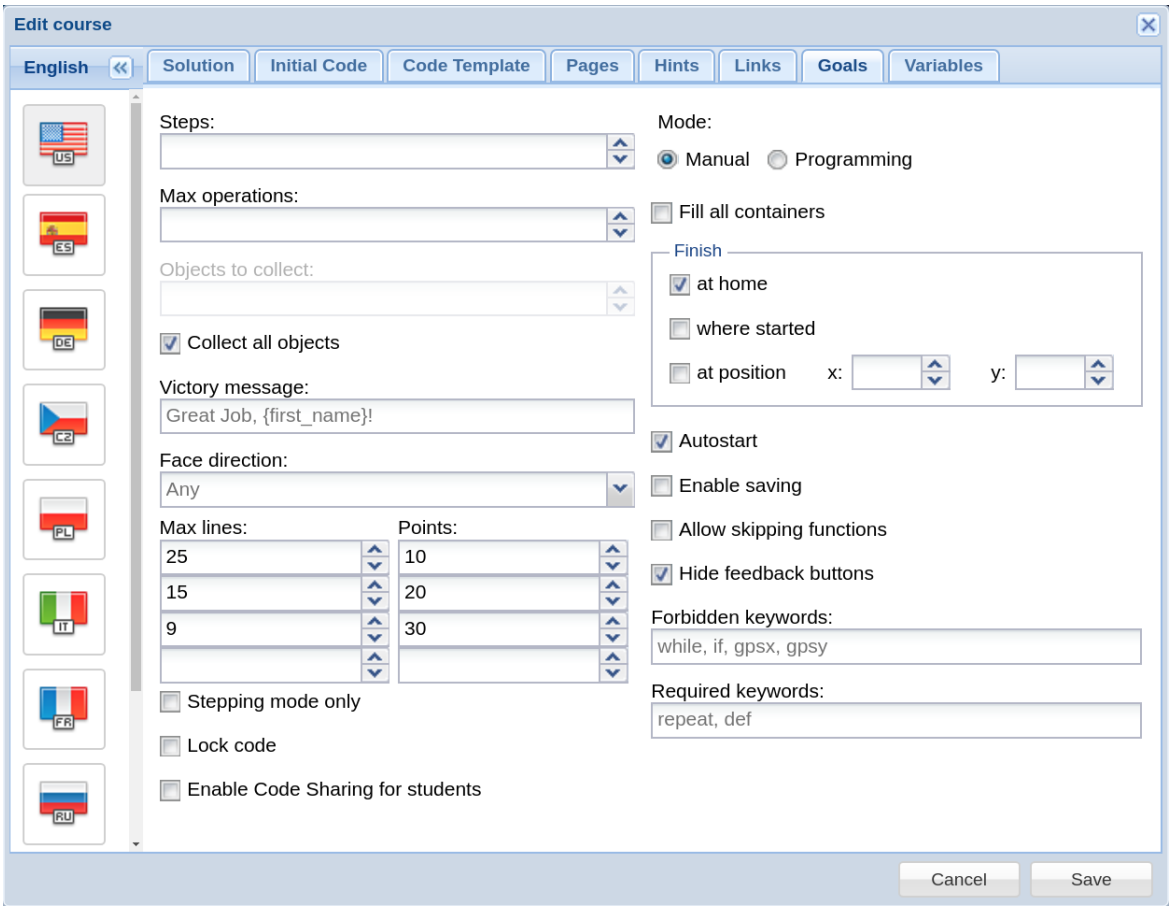
menu contains the following items:

- *Solution:* This is where you insert the correct solution (in Programming mode only).
- *Initial code:* This is where you insert the code that the user should see at the beginning (if any).
- *Code template:* This is where you insert help code (if any). The user will see he Code template button when they start the game.
- *Pages:* This is where you can add / edit / remove introductory pages with explanations.
- *Hints:* Here you can specify hints which the user will obtain after the first failure, second failure, etc.
- *Links:* Here you can specify links to a PDF document and a tutorial video.
- *Goals:* Here you define game goals. We will discuss this step separately.
- *Variables:* Here you can specify variables whose values should be checked at the end.

A.6. Define game goals

The panel used to define game goals is shown below. It contains the following items:

- *Steps:* Here you can limit the maximum number of steps Karel is allowed to make.
- *Max. operations:* Here you can limit the maximum number of operations.
- *Objects to collect:* In case you want Karel to collect some ojects in the maze but not all, here you can state their number.
- *Collect all objects:* Check this box if Karel is supposed to collect all objects in the maze.
- *Victory message:* Here you can type a custom victory message. The `first_name` in the curly braces will be replaced by the user's name as defined in his/her Settings.
- *Face direction:* If you want Karel to face a given direction (West, East, South or North) after the game finishes, select it here.
- *Max. lines / points:* This is where you specify how many points the user is getting for various numbers of lines. Typically, the shorter the program, the more points.
- *Stepping mode only:* This option will force the user to step through the code.
- *Lock code:* Checking this box will freeze the code - it will become non-editable.



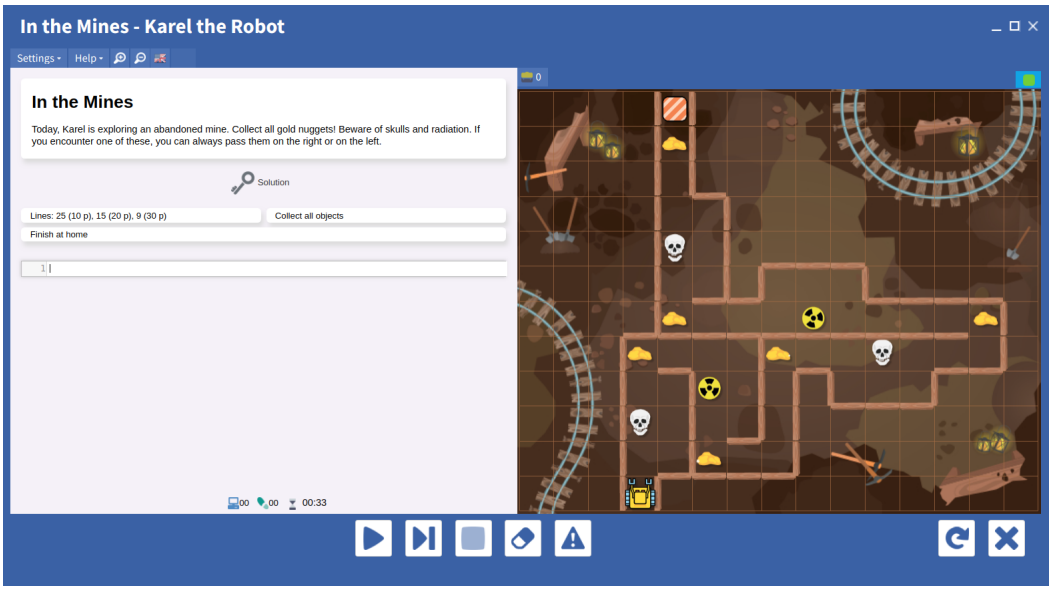
Game goals panel.

- *Enable Code Sharing for students:* This option is used when creating courses.
- *Mode:* Here you choose whether your game should be in Manual or Programming mode.
- *Fill all containers:* If the maze contains containers, here you can require that Karel fills them all.
- *Finish:* Here you choose where Karel should be when the program ends.

The rest are technical options used for course creation.

A.7. Final test

After game goals have been defined, you need to press the "Play game" button and try to solve the puzzle as a user would. This will make sure that all game goals can be fulfilled and the game is solvable.

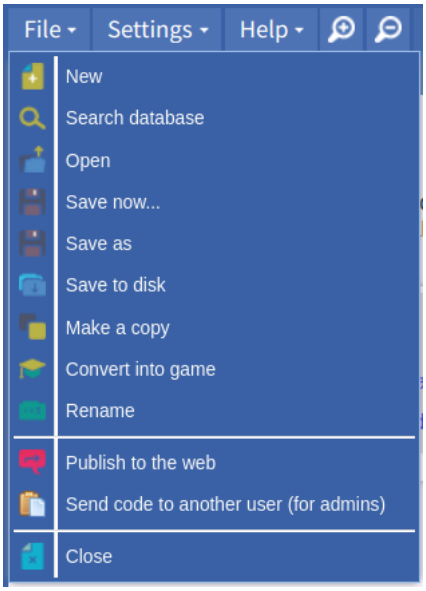


Playing the game in user mode.

That's it, your game is now ready! In the next section we will show you how to share it with others, and let them play it.

A.8. Publish your game in the Internet!

Let us now publish our example game and share it with others. For this, one needs to open the File menu and click on "Publish to the web":



The File menu offers "Publish to the web".

A window “Publishing options” should open:



Selection of “Anyone can view and run”.

Here, you will be offered several options for publishing your game. This is the generic publishing menu for all programs in NCLab, so some of the options may not be relevant at this moment.

We recommend the option “Anyone can view and run”. This will allow the users to view and play your game, but they will not be given access to the Edit menu and to the Designer - hence they will not be able to modify their copy of the game. If you wanted to enable them to do that, you would select the last menu option with the most access. BTW, do not worry about the users changing your original game worksheet - they cannot. Even if they are allowed to make changes, it is only in their copy.

Finally, you can post the public link to the game directly to social networks using the corresponding buttons. Or, you can copy the link and share it with others, for example, via email. Anybody with this link will be able to click on it and play your game.

B. Self-Paced Karel Course in NCLab

NCLab provides a self-paced gamified Karel course based on this textbook. Students beat levels like in a computer game, and they collect points and certificates. The course is popular with home schooled students, students in schools, and adult learners alike. It is not easy but it is rewarding. The course helps develop logic, computational thinking, problem solving, and perseverance. These skills are needed very much in computer programming but also in many other careers. After finishing the Karel course, the students have a solid foundation of logic and computatonal thinking, and moreover they are used to write programs in a simplified Python language. Therefore, students are ready for a quick and seamless transition to Python programming.

B.1. Age groups and prerequisites

There is no age limit - the course teaches skills which are age-independent, and is popular with students of all ages. The only prerequisite is keyboarding. Therefore we do not recommend it for children under 6 years.

B.2. Brief overview

The Karel language is the perfect language for beginning programmers. It will teach you how to design algorithms and write working computer programs without struggling with technical complications of mainstream programming languages. Also, Karel teaches all important concepts of modern procedural programming languages including correct code formatting, counting loops, if-else conditions, conditional loops, custom commands, functions, variables, lists, and recursion. The course comes with examples and tutorial videos that ensure that every student can progress at his or her own pace.

B.3. What does it take to be an instructor

The course can be taken at home without an instructor. It contains many examples and videos which make it completely self-paced. For schools and institutions: The instructor does not have to be an expert in computer programming. In fact, many educators use the Karel course to learn computational thinking skills themselves, and adult learners in general enjoy the course as much as K-12 students do.

B.4. Role of the instructor

Although the course is self-paced, the instructor (or homeschooling parent) still plays an important role as a coach. The course forces the students to change their problem-solving habits. Without thinking first, they fail to solve the task. This can lead to an initial frustration which is easier to overcome with the guidance and help of an instructor or parent. The students will gradually become independent thinkers and problem solvers, and will require your assistance less and less often.

B.5. Instructor training

NCLab provides instructor training. Contact us at `office@nclab.com` or (775) 300-7667 for details.

B.6. Course structure

The course has two parts - a self-paced part and creative projects. The self-paced part comprises 5 Units with 5 Sections each. Every Section contains eight game levels and a quiz. This makes for a total of 225 game levels. Students receive a new certificate after finishing each Unit. There is a project for each section (25 projects total). More details about the projects will be given in Section B.9.

B.7. Syllabus, lesson plans, pacing guide

The syllabus, lesson plans, and pacing guide can be downloaded from the Instructor Resources page `https://nclab.com/karel-resources/`.

B.8. Student journals, cheat sheets, solution manuals

The resources page `https://nclab.com/karel-resources/` also provides student journals, a cheat sheet with a quick overview of Karel commands, and a solution manual for instructors. An instructor also has access to solutions directly in the game levels.

B.9. Creative projects

The course comes with creative projects which are designed to allow the students to apply what they learned in each Section. There are 25 projects:

- Make a Maze
- Draw Your Name
- Stuff, Stuff, Stuff
- Fix It!
- Rules of Fun
- No Walls
- Detective
- Maze Master
- 12345
- Fruit loops
- Def-initely
- Strange Shapes
- Hard Fun
- Counting Up
- Three Rooms
- More Rooms
- True and False
- Minimal Maze
- Guide on the Side
- Uncertain Times
- Access Denied
- Maximm Maze
- Full Circle
- Real Robots
- Ball-in-a-maze



Project "Make a Maze".

B.10. Explore NCLab

Great work - congratulations on finishing this textbook! The next step is to explore NCLab at `http://nclab.com/`.

B.11. Contact us

Our main office is located at 450 Sinclair St, Reno, NV 89501, U.S.A. Whether you are a homeschooling parent looking to teach computer programming to your kids, or an educator in the K-12 or higher-ed systems, we can help you achieve your goals. You will find our team extremely friendly and responsive. We can be reached using the email `office@nclab.com` or by calling (775) 300-7667. We hope to hear from you!

Index

False, 185
True, 185
and, 69, 183
append, 225
dec, 132
def, 101
del, 233
do-while, 94
else, 61
empty, 67, 192
for, 25, 94
get, 8, 11
go, 8, 11
gpsx, 187
gpsy, 187, 275
home, 73, 192
if, 54, 194
if-elif-else, 73
if-else, 75
in, 161, 229
inc, 132
left, 8, 11
north, 64, 192
not, 66
or, 70, 184
pass, 65
pop, 231
print, 63, 130
put, 8, 11
repeat, 25, 28, 32, 36
return, 141
right, 8, 11
until, 94
wall, 54, 192
while, 79, 195
while True, 196

algorithm, 12
argument, 149

body
  of condition, 55
  of loop, 80
Boolean
  algebra, 182, 184
  function, 75, 190, 192, 273
  values, 185, 190
  variables, 185, 186
Braille, i, 266
Bubble sort, 263
bug, 18

Cardan grille, 270
case-sensitive, 21
code, 13
coin toss, 214
collectible objects, 58
comment, 26
commenting code, 26
conditions, 53
containers, 60
control prints, 195
cryptography, 200, 270
custom commands, 97

debugging, 18, 194

East, 9
efficiency, 119
Eight Queens puzzle, 276
ellipsis, 73
error
  logical, 12, 15
  message, 30
  syntax, 17, 30

Fibonnacci, 137
FIFO, 21
FILO, 20
First Maze Algorithm, 87
  failure, 89
  right-handed, 91
function, 141

G. Boole, 182
GPS, 187, 272, 278

immutable, 151
implementation, 13
indentation, 25, 29

Karel Čapek, 1
keyboard controls, 10

Law of Large Numbers, 217
list, 223
  for loop, 228
  addition, 232
  appending items, 225
  deleting items, 233
  empty, 224
  indices, 226
  length, 226
  multiplication, 232
  nonempty, 224
  of lists, 227
logic, 183
loop
  for, 25, 156, 228
  repeat, 25, 28, 36, 79
  while, 195
  conditional, 79
  counting, 25
  infinite, 197
  nested, 39, 40, 42–44, 47–50

maximum, 206
minimum, 211
mode
  designer, 282
  games, 282, 284
  manual, 8, 10, 282
  programming, 11, 281, 282
Morse, 267
Morse code, 172, 173, 175, 176, 178, 181

North, 9

obstacles, 56
operation, 10, 119, 120
operator
  and, 183
  or, 184
  arithmetic, 131, 132
  assignment, 130
  comparison, 133, 135, 190

parameter, 149
Pascal, 3
print, 197
probability, 214
program, 13
Python, 3, 5, 21, 25, 63, 94, 132, 156

queue, 21

R.E. Pattis, 2
R.U.R., 1
random
  Booleans, 213
  integers, 196, 201

  walks, 218
randomness, 200
recursion, 243, 245, 249, 258, 260, 262
  adding numbers, 254
  binary tree, 262
  infinite, 250
  parsing lists, 255
  stopping condition, 247, 249, 253, 262
repeating pattern, 30, 39

S.F.B. Morse, 172
scope
  global, 147
  local, 146
Second Maze Algorithm, 113
  right-handed, 117
sensor
  banana, 59, 60
  beeper, 59
  coconut, 61
  empty, 67, 68
  fire, 56
  home, 73
  north, 64
  spider, 59
  wall, 54
  water, 56
  acid, 72
sensors, 192
solution
  efficient, 15
  inferior, 15
  superior, 14
South, 9
stack, 20
step, 10
stepping, 80
subtask, 101
syntax, 16
testing, 172
text string, 155
  for loop, 156
  addition, 155
  as code, 167
  displaying quotes, 160
  indices, 157
  length, 156
  multiplication, 155
  reversing, 156
  search and replace, 164
  slicing, 159
  substring, 161–166
truth table, 183, 184

variable, 128
  Boolean, 128
  displaying, 130
  global, 147
  initialization, 130
  local, 146
  name, 129
  numerical, 128
  text string, 128

West, 9