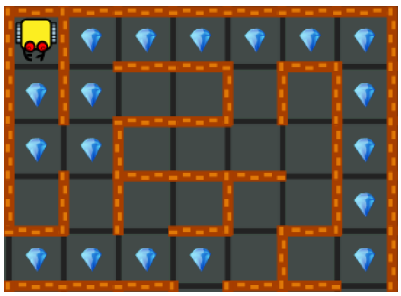
nclab
Online STEM Laboratory

First Course in Programming





with Karel the Robot and Python

Pavel Solin & Salih Dede

Revision Aug-26-2012. Copyright 2012 FEMhub Inc. All rights reserved.

About this Textbook

This free textbook is provided as a courtesy to NCLab users. It will help you discover the elegance and power of procedural and object-oriented programming with Karel the Robot and Python. The former is a famous educational language created at the Stanford University, the latter is a modern high-level dynamical programming language that is used in many areas of business, engineering, and science today. After taking this course, you will have solid theoretical knowledge and vast practical experience with computer programming.

About the Authors

Pavel Solin is Professor of Computational Science at the University of Nevada, Reno. Computer programming has been his deep personal passion since childhood, and later it became part of his profession. He has been developing advanced computer programs in many different languages, supervising large software projects, and teaching programming to college-level students. Salih Dede is Instructor of Computer Programming at the Coral Academy of Science High School in Reno. He has vast experience with teaching programming to high school students, and outstanding instructional achievements.

For Instructors

Review Book and Exercise Book containing hundreds of review questions with answers and programming exercises with solutions, respectively, are part of the NCLab-powered course *Intro to Programming with Karel the Robot and Python* that is available at

http://introtoprogramming.net

for a small subscription fee. In addition to standard cloud benefits such as anytime-anywhere accessibility through the web browser interface and working from mobile devices, the course comes with additional attractive features including automated grading and student progress tracking. The course will be opened in August 2012.

Table of Contents

I Karel the Robot

1 Introduction 5
- 1.1 Objectives 5
- 1.2 Brief history 5
- 1.3 Who is Karel? 5
- 1.4 What does this course have to offer? 5
- 1.5 Is Karel a toy language? 6

2 Launching Karel 6
- 2.1 Objectives 6
- 2.2 Karel modes 7

3 Manual Mode 8
- 3.1 Objectives 8

4 Programming Mode 9
- 4.1 Objectives 9
- 4.2 Typing commands 9
- 4.3 Algorithms, programs, and bugs 10

5 Counting Loop 10
- 5.1 Objectives 10
- 5.2 The `repeat` command 10

6 Working with Code and Text Cells 12
- 6.1 Objectives 12

7 Conditions 13
- 7.1 Objectives 13
- 7.2 Karel's five sensors 13
- 7.3 Programming hints 15

8 Conditional Loop 15
- 8.1 Objectives 15
- 8.2 The `while` command 16

9 Custom Commands 17
- 9.1 Objectives 17
- 9.2 Programming hints 17
- 9.3 Never replicate computer code 17
- 9.4 Defining new commands 18

10 Recursion 18
- 10.1 Objectives 18

10.2 How it works . . . 19
10.3 The base case . . . 21
10.4 When should recursion be used? . . . 21
10.5 Mutually recursive commands . . . 22
11 Variables and Lists . . . 23
11.1 Objectives . . . 23
11.2 Types of variables . . . 23
11.3 Reading GPS coordinates . . . 24
11.4 Custom Functions . . . 26
11.5 Creating numerical variables . . . 26
11.6 Changing values of numerical variables . . . 27
11.7 Comparison operations . . . 28
11.8 Lists . . . 29
11.9 Local and global variables . . . 33
12 Logic . . . 34
12.1 Objectives . . . 34
12.2 Logical expressions . . . 34
12.3 Logical variables . . . 34
12.4 Logical operations . . . 35
13 Appendix - Overview of Functionality by Level . . . 35
13.1 Level 0 (Section 3) . . . 35
13.2 Level 1 (Section 4) . . . 36
13.3 Level 2 (Sections 5 – 10) . . . 36
13.4 Level 3 (Sections 11, 12) . . . 36

II Introduction to Python

1 Introduction . . . 41
1.1 Objectives . . . 41
1.2 Compiled and interpreted programming languages . . . 41
1.3 Basic facts about Python . . . 41
1.4 Libraries - Python programmer's strong companion . . . 42
1.5 How does Python programming work in the cloud environment? . . . 42
1.6 Launching a Python project . . . 42
1.7 Hello, World! . . . 43
1.8 Code, output and text cells . . . 44
1.9 Various ways to run programs . . . 44
2 Using Python as a Calculator . . . 45
2.1 Objectives . . . 45

2.2 Addition, subtraction . . . 45
2.3 Multiplication . . . 46
2.4 Division . . . 46
2.5 Powers . . . 47
2.6 Modulo . . . 47
2.7 Order of operations . . . 47
2.8 Using empty characters makes you code better readable . . . 48
2.9 Using mathematical functions . . . 48
2.10 Complex numbers . . . 49
3 More on Functions . . . 50
3.1 Objectives . . . 50
3.2 Defining new functions . . . 51
3.3 Passing arbitrary arguments . . . 51
3.4 Returning multiple values . . . 52
3.5 Using default arguments . . . 53
4 Plotting . . . 56
4.1 Plotting functions of one variable . . . 56
4.2 Labels, colors, and styles . . . 57
4.3 Scaling axes and showing grid . . . 60
4.4 Adjusting plot limits . . . 61
4.5 Plotting multiple functions at once . . . 62
4.6 Plotting parametric 2D curves . . . 63
4.7 Plotting parametric 3D curves . . . 64
4.8 Plotting functions of two variables . . . 66
4.9 Plotting functions of two variables with WebGL . . . 68
5 More on Variables . . . 70
5.1 Objectives . . . 70
5.2 Assigning values to variables . . . 70
5.3 Dynamic type interpretation . . . 70
5.4 Changing values of variables . . . 71
5.5 Local and global variables . . . 72
5.6 Sure method to spoil your code . . . 73
5.7 Variable shadowing . . . 74
6 Review of Logic . . . 74
6.1 Objectives . . . 74
6.2 Boolean expressions . . . 75
6.3 Boolean variables . . . 75
6.4 Boolean operations . . . 76
7 More on Conditions and the Conditional Loop . . . 76

7.1 Objectives . . . 76
7.2 Generating random numbers in Python . . . 78
7.3 The `break` statement . . . 80
7.4 The `continue` statement . . . 80
7.5 A trip into scientific computing . . . 81
8 Strings . . . 83
8.1 Objectives . . . 83
8.2 Using quotes . . . 84
8.3 Multiline strings and backslashes . . . 84
8.4 Concatenation and repetition . . . 86
8.5 Referring to letters by their indices . . . 86
8.6 Parsing strings with the `for` loop . . . 87
8.7 Slicing strings . . . 88
9 Tuples, Lists, and Dictionaries . . . 89
9.1 Objectives . . . 89
9.2 Tuples . . . 89
9.3 Lists . . . 90
9.4 Dictionaries . . . 93
10 Counting Loop . . . 95
10.1 Objectives . . . 95
10.2 Combined with the `range()` function . . . 96
10.3 Combined with a general list or tuple . . . 96
10.4 Reverting a dictionary . . . 97
11 Exceptions . . . 97
11.1 Objectives . . . 97
11.2 Never underestimate the user . . . 98
11.3 `ZeroDivisionError` Exception . . . 99
11.4 The `assert()` function . . . 99
11.5 The `raise` statement . . . 100
11.6 Other types of exceptions . . . 100
12 Object-Oriented Programming . . . 101
12.1 Objectives . . . 101
12.2 From procedural to object-oriented programming . . . 101
12.3 Example: Moving to a different city . . . 101
12.4 Classes, objects, and methods . . . 102
12.5 An example from geometry . . . 102
12.6 Defining and using class `circle` . . . 103
12.7 Text string as a class . . . 104
13 Class Inheritance . . . 110

13.1 Objectives 110
13.2 Creating base class `geometry` 110
13.3 Deriving `polygon` from `geometry` 111
13.4 Deriving `circle` from `geometry` 112
13.5 Deriving `triangle` and `quad` from `polygon` 113
13.6 Deriving `rectangle` from `quad` 114
13.7 Deriving `square` from `rectangle` 114
13.8 Diagram of class structure 115
13.9 Creating sample instances 115
14 Python Libraries 117
14.1 Objectives 117
14.2 Python Standard Library 118
14.3 Pylab and Matplotlib 118
14.4 Scipy and Numpy 118
14.5 Sympy 118
15 Other recommended topics in Python 118
16 What next? 119

Foreword

This course provides a gentle yet efficient and comprehensive introduction to modern algorithmic design and computer programming. It consists of two programming languages – *Karel the Robot* and *Python*. Karel the Robot is a famous educational programming language that was created at the Stanford University. It features a robot that collects gems in a maze, and helps students develop advanced algorithmic skills using a handful of simple commands that do not involve math. Python is a modern high-level dynamical programming language that is widely used in business, science, and engineering applications. It is much easier to learn than C, C++ or Java.

Starting with Karel the Robot, students discover principles of computer programming effortlessly, without being exposed to technical details of conventional programming languages. Could such an introduction be done with a conventional programming language? Yes, but it would be obscurred and slowed down by technicalities, not mentioning that it would not be fun. The course starts out in Manual mode (Level 0) where the robot can be guided via clicking on five buttons *Go* (make one step forward), *Left* (turn left), *Right* (turn right), *Put* (put a gem on the groud) and *Get* (pick up a gem from the ground).

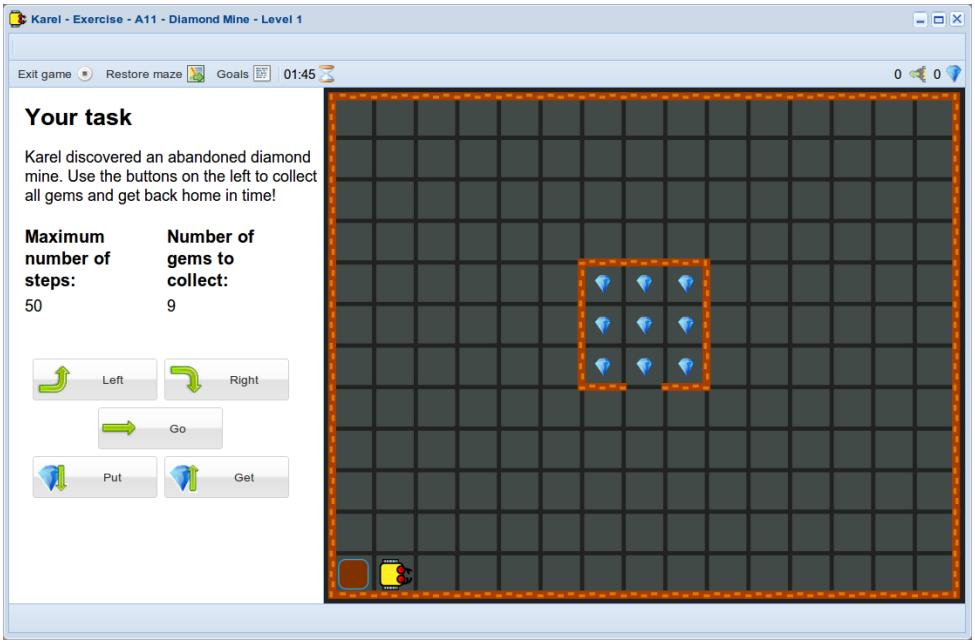


Fig. 1: Sample game *Diamond Mine* in Manual mode.

In the next level which is called *Bridge to Programming* students keep solving problems by typing the commands `go`, `left`, `right`, `put` and `get` instead of clicking on buttons.

The need for higher functionality such as loops, conditions, and custom commands arises naturally as game goals become more complicated. Students learn quickly that it is advantageous to break complex tasks into smaller ones, which is one of the most important principle of computer programming. The textbook is written by programming experts, and in addition to advanced programming skills the students gain an overview of good and bad programming habits.

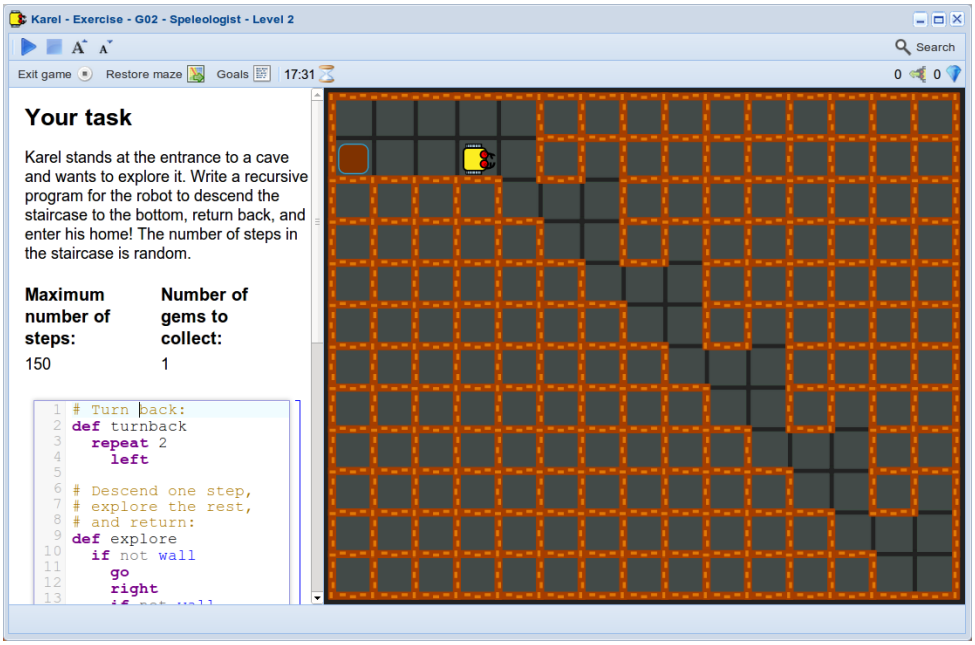


Fig. 2: Sample game *Speleologist* in Programming mode.

The syntax of Karel the Robot is very close to Python – in fact Python feels like Karel's older brother. The transition from Karel to Python is as seamless as the transition from one Karel's Level to another. In Python, students continue building their skills in the same way they did with Karel. They learn more advanced concepts including mathematical operations, plotting, local and global variables, strings, tuples, lists, dictionaries, and others. They get to solve realistic problems, but Python is still a lot of fun.

A strong companion of Python are its libraries. Besides the Standard Library that contains many build-in functions not present in lower-level languages such as Java, C, C++ or Fortran, Python also has powerful scientific libraries including Scipy, Numpy, Matplotlib, Pylab, Sympy and others. With these, students are ready to solve entry-level scientific and engineering problems. These libraries are discussed at the end of the course.

Part I

Karel the Robot

1 Introduction

1.1 Objectives

- Learn basic facts about the Karel language and its history.
- Learn how Karel differs from other programming languages.
- Learn what skills this course provides.

1.2 Brief history

The educational programming language Karel the Robot was introduced by Richard E. Pattis in his book "Karel The Robot: A Gentle Introduction to the Art of Programming" in 1981. Pattis first used the language in his courses at Stanford University, and now it is used at countless schools in the world to introduce students to algorithmic design and computer programming. The language is named after Karel Čapek, a Czech writer who invented the word "robot" in his 1921 science fiction play R.U.R. (Rossum's Universal Robots).

1.3 Who is Karel?

Karel is a little robot that lives in a maze and loves to collect gems. He was manufactured with only five simple commands in his memory:

- `go` ... make one step forward.
- `get` ... pick up a gem from the ground.
- `left` ... turn to the left.
- `right` ... turn to the right.
- `put` ... put a gem on the ground.

He also has five built-in sensors that allow him to check his immediate surroundings:

- `wall` ... true if he would crash into a wall by making one more step, false otherwise.
- `gem` ... true if he stands on a gem, false otherwise.
- `north` ... true if he is facing North, false otherwise.
- `home` ... true if he is at home, false otherwise.
- `empty` ... true if his bag with gems is empty, false otherwise.

1.4 What does this course have to offer?

Computer programming skills are highly valued today, and they will be even more valued in the future. Karel's language is so natural that we will be able to focus on designing great algorithms. This is the most important skill in computer programming. In other words, we could start learning programming with a technically complicated conventional programming language as well, but we would spend lots of time battling technical problems. The advantage of starting with Karel is that when moving on to other languages, we will be able to focus on the technical differences, as the programming concepts we will already understand.

1.5 Is Karel a toy language?

Definitely not! Despite its playful appearance, Karel features all elements of modern procedural programming. The complexity of algorithms that we will encounter in this course ranges from *extremely simple* to *extremely tough*. Towards the end of this course we will encounter tasks that will make our head spin. However, Karel is very light on technicalities, which is very good for someone who is just starting out.

The biggest conceptual difference between Karel and standard procedural programming languages such as Python, C, C++ or Fortran is that *the robot does not know math*. At least not until we get to advanced levels that are beyond the original Pattis' book. In the basic course, we will solve many exercises whose objective is to teach how to design great algorithms, and math is not needed for that. After finishing this tutorial, we will be able to transition smoothly to Python where we can do as much math as we like!

2 Launching Karel

2.1 Objectives

- Learn to launch Karel and work with the graphical application.
- Learn that Karel has several modes and how they differ from each other.

The simplest way to launch Karel is to click on the icon *Programming* and select *Karel* in the menu. This will launch the application in *Programming mode* with a randomly generated maze, as shown in Fig. 3.

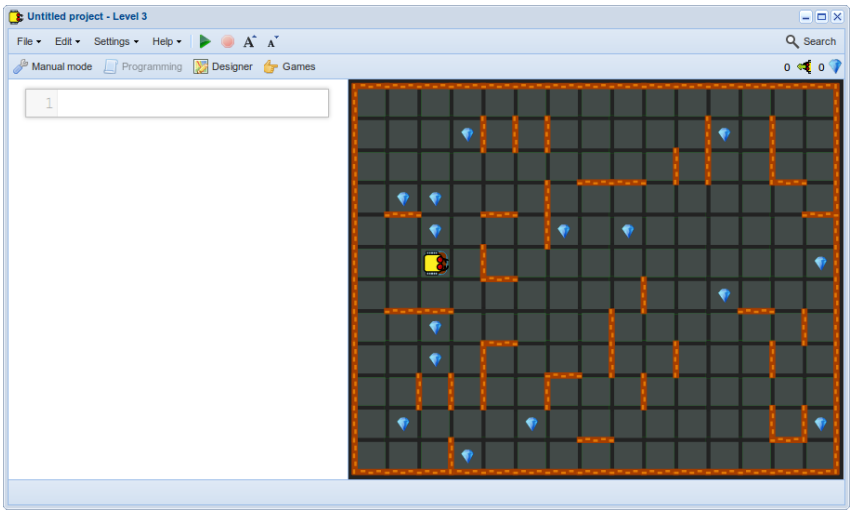


Fig. 3: Launching Karel in Programming mode, with a random maze.

The Programming mode is the most frequently used one. It is possible to switch to *Manual mode*, *Designer*, and *Games* in the menu. These modes will be discussed in Paragraph 2.2.

The application window contains two lines of menus and information on top, work area on the left, maze on the right, and status bar on the bottom. The menus are fairly intuitive, so let us just explain a few selected functions, starting with the *File* menu:

- *Open* will open a file in your account.
- *New* will generate a new random maze.
- *Clone* will copy a Displayed Project into your account.
- *Display* will submit a project for review. If you think that other users would find your program or game interesting, this is the way to make it available to them. Upon passing the review, the project will be added to Displayed Projects.

Edit menu enables operation with code and text cells (to be discussed in Section 6). In *Settings* one can change Karel's *Level* (to be discussed in Paragraph 2.2), change his speed, and adjust sound preferences. The green and red buttons are used to run and stop programs, respectively, and the two buttons next to them on the right can be used to increase and decrease font size. The pair of icons on far right is the step counter (that can be reset by clicking on it) and gem counter that indicates how many gems Karel has in his bag.

2.2 Karel modes

Karel operates in four modes:

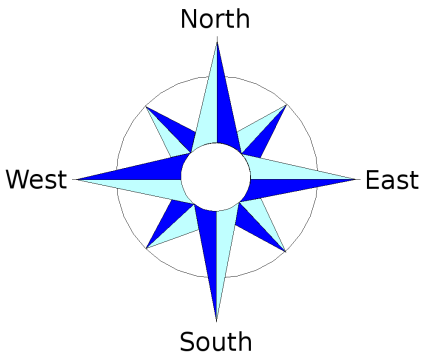
- *Manual mode:* The robot is controlled using the mouse and five buttons Go, Get, Left, Right, and Put. Watch out and do not crash!
- *Programming mode:* The robot is controlled using written programs (computer code). The Programming mode is split into several Levels:
  - Level 1 is a transition layer between the Manual and Programming modes. Programs are written using only five commands `go`, `get`, `left`, `right`, and `put` that exactly correspond to the buttons Go, Get, Left, Right, and Put in Manual mode.
  - Level 2 is where the actual programming begins. On top of the commands from Level 1, programs can contain conditions, loops, and custom commands.
  - In Level 3 Karel has a GPS device, and he is able to work with variables and lists. Functions can return results.
- *Designer:* This mode allows the user to create custom mazes.
- *Games:* Makes it possible to create and play games.

3 Manual Mode

3.1 Objectives

– Learn to operate the robot in Manual mode.

Before we begin, let us review the four directions on the compass:



When launching Karel through the Programming icon, switch to Manual mode using the corresponding menu button. Then, five buttons Left, Right, Go, Get and Put will appear in the panel on the left, as shown in Fig. 4.

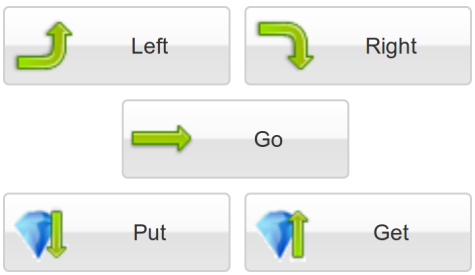


Fig. 4: Karel's buttons in Manual mode (robot facing East).

The function of the buttons is self-explanatory – pressing Left will turn the robot 90 degrees to the left, pressing Right will turn him 90 degrees to the right, and pressing Go will move him one step forward (watch out and do not crash!). Upon pressing Put the robot will reach into his bag with gems, take one, and put it on the ground where he stands. An indicator showing how many gems are in the bag can be found in the upper right corner of the window. Last, upon pressing

Get the robot will pick up a gem from the ground where he stands. If one asks the robot to get a gem where there is none, he will complain.

When the robot turns, the arrows on the buttons adjust automatically to his new direction. This is illustrated in Fig. 5.



Fig. 5: Karel's buttons in Manual mode (robot facing West).

4 Programming Mode

4.1 Objectives

- Start operating the robot in Programming mode.
- Understand the difference between *algorithm* and *program*.
- Learn the difference between *syntactical* and *logical* mistakes.
- Understand that *debugging* is an indivisible part of computer programming.

4.2 Typing commands

In Programming mode, commands for the robot are entered into an code cell located in the left panel. These commands are `left`, `right`, `go`, `get`, and `put`. Their function is the same as the function of the corresponding buttons in Manual mode. One or more commands form a *computer program (computer code)*. Often we just say *program* or *code*. There are two simple rules to remember:

1. Always type one command per line.
2. Do not enter empty characters in front of commands.

Ignoring these rules would not make the program invalid, but the code would be difficult to read. It is important to write a clean, transparent code.

4.3 Algorithms, programs, and bugs

Karel always obeys all commands *exactly*. Sometimes it may happen that we plan one thing but to our surprise the robot does something else. In most cases this happens when our *algorithm* is wrong. By an *algorithm* we mean a sequence of logical steps that the robot needs to follow in order to fulfill his task. Algorithms are presented using normal human language, not in terms of the robot's commands.

Program or *computer code* is created when the algorithm is expressed in terms of the robot's language. When our algorithm is good, then the program is easy to write.

Mistakes in algorithms are called *logical errors*. A logical error is, for example, when we crash the robot into a wall because we forgot to make a turn. Mistakes such as mis-spelling a command, writing "1o" instead of "10", or forgetting indentation are related to *syntax* and they are called *syntactical errors*. Of these two types, logical errors are usually much more difficult to find.

In general, mistakes or either kind are called *bugs* and the procedure of eliminating them is called *debugging*. Depending on how careful we were while preparing our 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.

When we commit a syntactical error, the robot will write an error message and do nothing. If our algorithm contains a logical error, then he will write an error message and stop executing the program. The most usual logical error are:

- Karel crashes into a wall.
- The robot tries to collect a gem where is none.
- He attempts to put a gem on the ground while his bag is empty.

5 Counting Loop

5.1 Objectives

- Learn to make the robot repeat something a given number of times.

In Section 4 we successfully crossed the bridge between manual control and programming. The bridge collapsed, there is no way back. But do not worry about that! The land of Programming is much more beautiful, and once you understand its beauty, you will never want to leave.

5.2 The `repeat` command

The *counting loop*, represented by the `repeat` command, can save us lots of writing when something is repeated a given number of times. For example, for Karel to make 15 steps forward we could type:

```
go
go
go
go
go
go
go
go
go
go
go
go
go
go
go
```

But this is neither efficient nor elegant. Instead, the same can be achieved by telling Karel to `repeat` the `go` command `7` times:

```
repeat 15
    go
```

There are a few simple rules that we need to remember when using the `repeat` command:

- Keep code readable – always write one command per line.
- Indentation – all commands to be repeated (the *body of the loop*) need to be indented. One can choose between 2-indent and 4-indent. The former yields more compact code with not-so-long lines, the latter is easier to read.
- Cancel the indentation for the first command that does not belong to the body of the loop.

To illustrate what the indentation does, let's look at a code that will move the robot 8 steps forward:

```
repeat 4
    go
    go
```

Compare to a code that will only move the robot 5 steps forward:

```
repeat 4
    go
go
```

Multiple `repeat` commands can be *nested*. This means that a `repeat` command can used in the body of another `repeat` command. Everything that was said about indentation still holds. Can you figure out what the following code does?

```
repeat 10
    repeat 5
        go
    repeat 2
        left
```

After you figured it out, launch Karel via the *Programming* menu, enter this code into the code cell, and run it! But before doing that, remember to switch to *Designer* and make some free space in front of the robot so that he does not crash into a wall.

6 Working with Code and Text Cells

6.1 Objectives

- Learn why it is good to include comments in the code.
- Learn how to add new code cells and descriptive text cells.
- Learn how to change the order of cells.
- Learn how to run all code cells at once, and how to run them individually.
- Learn how to clear, collapse, remove and merge cells.

It is a very good habit to add comments to programs (every line starting with the '`#`' symbol is a comment) and also to include descriptive texts in Karel worksheets, since descriptions make it easier for someone else to understand your program. Sometimes you may be in the position of the "someone else" yourself, when you return to your own program some time after you wrote it. Here are a couple of simple new rules to remember:

1. New text cell can be added under the current cell via the button *Add text cell* under the cell. After an empty text cell appears, click into it and add text. The text can be formatted using Restructured Text Format (RST). After the text is finished, click on the button *Save cell* under the text cell.
2. The most commonly used formatting operations in RST are
   - Making some text bold face. For this, use double asterisk '`**`' on either side of the text. For example, type `**This is bold face**` in RST.
   - Headings are obtained by undelining some text with the '=' symbols. An example of a heading is:
     ```
     This is a Heading
     =================
     ```

  - The RST format is very popular and so it is easy to google more information if needed.
3. New code cell can be added under a cell by clicking on *Add code cell* under the cell. Having multiple code cells can be useful, for example, to test various versions of our program, or if we want to run parts of the program separately.
4. Order of cells can be changed via the arrows under the cells.
5. Clicking on *Clear cell* under a cell will erase its contents.
6. Any cell can be collapsed by clicking on the *Collapse cell* button under the cell. Then a new button *Expand this cell* will appear instead of the original cell. The cell can be expanded by clicking on this button.
7. Clicking on the button *Remove cell* under a cell will remove it. All text or code in that cell will be lost.
8. Run programs via the green arrow button in the main menu (this will evaluates all code cells). Each cell can be also evaluated individually by clicking on the green arrow button under the cell. If there is just one code cell, both options are equivalent.
9. Programs can be stopped using the red button in the menu.
10. If a particular code cell needs to be stopped, then use the red button under that cell.
11. Cells can be merged via the options *Merge active cell up* and *Merge active cell down* in Edit menu.

7 Conditions

7.1 Objectives

- Understand the function of Karel's five sensors.
- Learn to use the sensors in conjunction with *conditions* to help Karel check his surroundings and react accordingly. For example, we will learn to first check whether wall is ahead before making a step forward, checking whether there is a gem on the ground before attempting to pick it up, etc.

Karel has built-in sensors to better navigate in the maze:

7.2 Karel's five sensors

`wall` sensor

The first one is an infrared sensor `wall` that the robot uses to determine whether it is safe to make one step forward, or whether there is a wall. This is illustrated in Fig. 6.

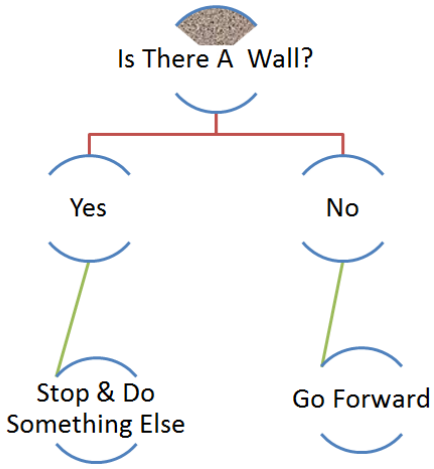


Fig. 6: This is how Karel uses the `wall` sensor!

The usage of the `wall` sensor in a program can be illustrated using a simple program "Careful step" where Karel first checks whether there is a wall ahead before making a step. If there is wall, he turns back:

```
# Program "Careful step".
if wall
    repeat 2
        left
else
    go
```

As we mentioned before, the symbol '`#`' introduces a comment, meaning that the line of code behind it is ignored by the robot. The `else` branch does not have to be there if it is not needed. Notice the indentation of the bodies of the `if` and `else` branches - this is analogous to how we indent the body of the `repeat` command. There are four more sensors:

`gem` sensor returns true if the robot stands on a gem, false otherwise.

`empty` sensor returns true if the robot's bag with gems is empty, false otherwise.

`north` sensor returns true if the robot is facing North, false otherwise.

`home` sensor returns true if the robot is at home, false otherwise.

Testing opposites

Karel can also use the reserved word `not` to test the opposites. For illustration, the previous program can be rewritten as follows, without changing its function:

```
# Program "Careful step".
if not wall
    go
else
    repeat 2
        left
```

7.3 Programming hints

Good programmer is a careful programmer! Errors can be avoided by always checking the appropriate sensor before making an action. A few examples of careful actions:

```
if not wall
    go
```

Another example:

```
if gem
    get
```

And a last one:

```
if not empty
    put
```

8 Conditional Loop

8.1 Objectives

– Learn to repeat a command or a sequence of commands when it is not known how many repetitions will be needed.

8.2 The `while` command

Often Karel needs to repeat something, *not knowing in advance how many repetitions there will be*. So, the `repeat` command is not practical. This can be the case, for example, when the robot is asked to walk straight ahead until he reaches the closest wall. Remember that he only can see walls that are right ahead of him – walls that are further away he can't see. Such a program would be:

```
while not wall
    go
```

Or, Karel may be asked to walk until he gets home:

```
while not home
    go
```

Beware though – **this program is dangerous** since the robot will crash into a wall if his home is not straight ahead of him!

Another example: The robot may be asked to empty his bag (he does not know how many gems are in it):

```
while not empty
    put
```

Or, he may be asked to collect all gems from a pile (he does not know how many gems there are):

```
while gem
    get
```

Or we may ask him to turn to face North (he does not know which direction he is facing):

```
while not north
    left
```

At last, let us face the following situation: Karel is asked to turn South, walk straight ahead until he reaches the closest wall, and collect all gems that he can find on the way:

```
# First turn North.
while not north
    left

# Then turn South.
```

```
repeat 2
    left

# Go straight ahead and pick all gems.
while not wall
    if gem
        get
    go

# Pick gem at the wall (if any).
if gem
    get
```

Notice that we first need to turn the robot to face North – this is because North is the only direction that he can check!

9 Custom Commands

9.1 Objectives

- Learn that replicating computer code is a very bad habit.
- Learn that splitting the big task into smaller ones will simplify the solution a lot.
- Learn to bring more structure and clarity into our programs by introducing new commands.

9.2 Programming hints

Always look for small tasks that can be solved independently of the big ones. Solve the small tasks first, and you will see that the big ones get much simpler. The importance of what we just said cannot be stressed more. Please read these three lines once more.
The fact that you are reading this line of text proves that you are not a perfect programmer yet. A perfect programmer would be stuck forever in the previous three lines which are an infinite loop!

9.3 Never replicate computer code

A new command should be defined whenever it becomes clear that the same action is repeated in the algorithm multiple times (yes, we talk about the algorithm, not about the program). When we start writing a program and then realize that the same code repeats itself at various places, then probably we did not do a good job designing the algorithm.

Sometimes it might be tempting to just replicate the same code several times in the program, because it does the same thing, but do not do it! This would be very bad programming and

sooner or later our own code would punish us for that. We would make a small change at one place but forget to do it in all the other places. Then our code would start acting strange, it would sometimes work and sometimes fail. We would spend lots of time looking for the mistakes, find some of them but not all, and our program would become unreliable and after some time irreparable. We would find that we need to rewrite it from scratch.

9.4 Defining new commands

New commands are defined using the reserved word `def`. For example, in a program where the robot needs to turn back many times, it is a good idea to define a new command `back` as follows:

```
def back
    repeat 2
        left
```

Note the indent – the body of a new command needs to be indented analogously to the bodies of loops and conditions.

10 Recursion

10.1 Objectives

- Understand what recursion is and when it can be useful.
- Learn to write good recursive algorithms.

By a *recursive* algorithm we mean an algorithm that uses itself. Doesn't it sound weird? But in our life we use recursion all the time, for example when we descend a staircase. With a bit of abstraction, our algorithm is:

```
Descend_staircase
    Descend one step
    If this was not the last step
        Descend_staircase
```

Recursion is not advantageous for all types of problems, but it can be really helpful, especially for problems where

- we can do some work to reduce the problem to the same one but smaller in size,
- we can apply the same algorithm to the smaller problem.

On program level, this means that some command calls itself, either directly or through other commands.

10.2 How it works

Consider the following program:

```
def reach_wall
    if not wall
        go
        reach_wall

reach_wall
```

Imagine that the initial position of the robot is like in Fig. 7.

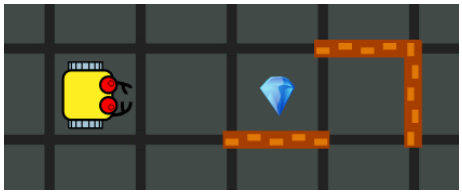


Fig. 7: Robot's initial position.

When the command `reach_wall` is first called, the robot stands three steps away from the wall and thus the `if not wall` condition passes. Then the command `go` follows and the robot's position changes as shown in Fig. 8.

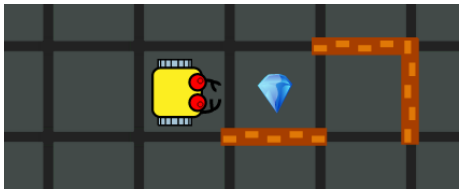


Fig. 8: Robot's position after the first `if not wall` condition passes and he makes the first step forward.

Next the robot executes the `reach_wall` command that follows the `go` command. A good way to understand what happens is to imagine that the command is replaced with its own body. The corresponding program would look as follows:

```
if not wall
    go
    if not wall
```

```
        go
        reach_wall
```

Since the robot is two steps away from the wall, the second `if not wall` condition passes and he makes a second step forward. His new position is shown in Fig. 9.

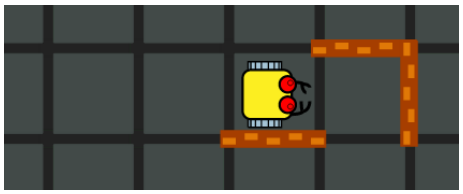


Fig. 9: Robot's position after the second `if not wall` condition passes and he makes a second step forward.

Next the robot executes the third `reach_wall` command. Again we can imagine that the command is replaced with its own body. The corresponding program would look as follows:

```
if not wall
    go
    if not wall
        go
        if not wall
            go
            reach_wall
```

Since the robot is one step away from the wall, the third `if not wall` condition passes and he makes a third step forward. His new position is shown in Fig. 10.

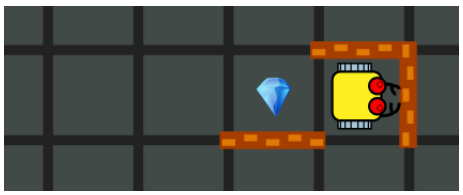


Fig. 10: Robot's position after the third `if not wall` condition passes and he makes a third step forward.

Next the robot executes the `reach_wall` command again and we can imagine that the command is replaced with its own body. The corresponding program would look as follows:

```
if not wall
    go
    if not wall
        go
        if not wall
            go
            if not wall
                go
                reach_wall
```

However, now the robot is facing the wall and thus the `if not wall` condition does not pass. This means that the program is finished!

10.3 The base case

In the previous example we have seen one important fact: To prevent infinite recursion, one always needs `if` or `if-else` statement of some sort where one branch makes a recursive call, and the other branch is either missing or it does not make a recursive call. The branch without a recursive call is called the *base case*. A bad example of a recursive command without a base case would be

```
def left_forever
    left
    left_forever
```

One can guess where the name of this command comes from! Fortunately, the program can be stopped using either the red button or the `stop` button under the code cell.

10.4 When should recursion be used?

The recursive command `reach_wall` that we defined above may not be the most useful example since the same functionality could be achieved more elegantly without recursion, just with

```
def reach_wall
    while not wall
        go
```

If there is a small task that needs to be solved repeatedly in order to get a bigger task done, then often one can write both a non-recursive and a recursive algorithm. Recall for example the Diamond Staircase example from Section 9. The small task there was to climb one step and pick up the gem, ending up facing east. The non-recursive version of the algorithm would be:

```
def climb_one_step
    left
    go
    right
    go
    get

while not home
    climb_one_step
```

The recursive version of the algorithm looks as follows:

```
def climb_the_stairs
    if not home
        left
        go
        right
        go
        get
        climb_the_stairs

climb_the_stairs
```

If it is easy to write a non-recursive version of an algorithm, then it should be used because in general, non-recursive algorithms are faster. Recursive version of an algorithm should be used if a non-recursive version would be difficult to design. For example, nearly all code written for tree-like structures is recursive. Also many sorting algorithms are more naturally written in recursive form. We will come to these subjects later.

10.5 Mutually recursive commands

Recursion can have interesting forms. In its simplest shape, a command calls itself in its own body. But, we can have a pair of commands that call themselves mutually, such as the commands `odd` and `even` in the following example (that also solves the Diamond Staircase problem by the way):

```
def climb_step
    left
    go
    right
    go
```

```
    get

def odd
    if not home
        climb_step
        even

def even
    if not home
        climb_step
        odd

odd
```

Obviously this program is not the most efficient one to solve the Diamond Staircase problem, but it is great for illustration purposes.

11 Variables and Lists

In this Section we are entering exciting Level 3! Karel grew up and left the home of his parents to experience a life of his own. Therefore, his home will not be present in the following exercises and games anymore. There are a few more changes that reflect Karel's growing up, including numerical and logical variables, functions that return values, and lists. Do not forget to adjust the level to Level 3 in Settings. A compact overview of new functionality in Level 3 can be found in Section 13.

11.1 Objectives

- Understand the concept of variables.
- Learn to work with numerical and logical variables.
- Learn to use functions that return values.
- Understand that variables defined inside functions are local.
- Learn to work with lists.

In programming, variables are used to store useful information for later use. This information can be a number, word, sentence, or anything else.

11.2 Types of variables

All of us are using variables in our lives. One of the first ones is our own name. With a bit of abstraction (and say that your name is "Melissa"), when you were about two years old, you did the following:

```
my_name = "Melissa"
```

Since then, each time someone called a name, you retrieved in your brain the value of the variable `my_name`, compared it to the name that you heard, and if you got a match then you turned around to see who was calling you. Imagine that we would not be able to use that variable!

The variable `my_name` stores a word (string of characters). We also use many numerical variables such as `seconds_per_minute` whose value is 60, `minutes_per_hour` whose value is 60 as well, `hours_per_day` whose value is 24, and we could go on. The last three variables do not change too often, most likely they will not change during our lives. But we also use variables whose values change, such as `days_per_year`, `number_of_my_pets`, etc. In Karel, we will only use integer numbers.

Next let's look at logical variables. Those are variables that only can store two possible values – `True` or `False`, and we use a huge number of them. One such example may be `I_speak_a_foreign_language`. For someone this variable has the value `False`, for someone it is `True`. The important thing is that when someone asks you about that, you do not need to go check the records in the language school – you just know it. The value is *stored*, it does not have to be *created* each time it is needed. Logical variables will be discussed in more detail shortly – first let us explore the numerical ones.

11.3 Reading GPS coordinates

Karel's coolest Christmas present was a new GPS device that allows him to determine his position in the maze. With it, he will never get lost again! He can retrieve his coordinates at any time via the kaywords `gpsx` and `gpsy`. He also has a new ability to output results via the `print` command. The usage of these commands is best illustrated using the following short program where Karel determines his coordinates in the maze and prints them:

```
print "Horizontal position:", gpsx
print "Vertical position:", gpsy
```

The south-west corner of the maze is the origin of the coordinate system and it has coordinates [0, 0]. Let's say that Karel stands as shown in Fig. 11.

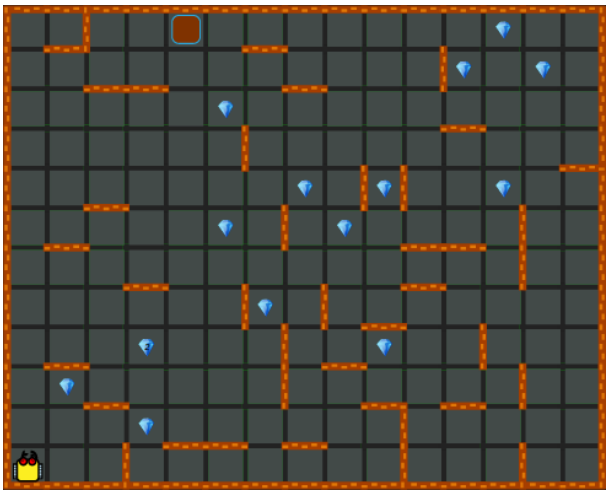


Fig. 11: South-west corner of the maze has GPS coordinates [0, 0].

Then the above program has the following output:

```
Horizontal position: 0
Vertical position: 0
```

The maze's width (in west-east direction) is 15 tiles, and its height (in south-north direction) is 12 tiles. If Karel stands in the north-east corner as shown in Fig. 12,

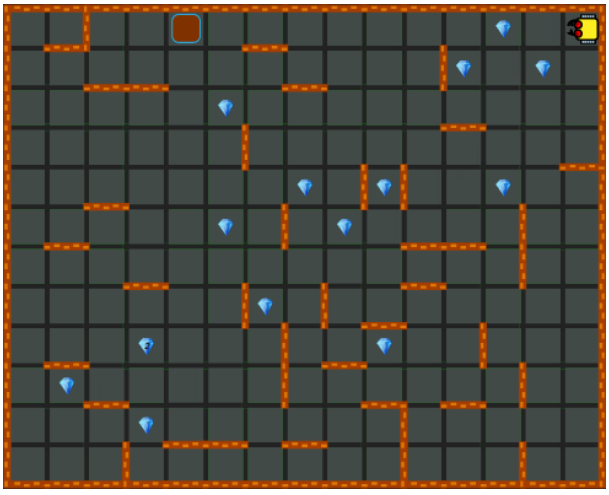


Fig. 12: North-east corner of the maze has GPS coordinates [14, 11].

then the output of the program is

```
Horizontal position: 14
Vertical position: 11
```

Try it! Also move Karel to other parts of the maze in Designer and run the program again to make yourself familiar with how the GPS device works.

11.4 Custom Functions

In Level 3 we can use the reserved word `return` inside the body of a command to return a value. Such commands are then called *functions*. They are also defined using the keyword `def`. For example, the following function `count_steps` will return the number of steps the robot needed to make in order to reach the closest wall in the direction that he was facing:

```
def count_steps
    n = 0
    while not wall
        go
        inc(n)
    return n
```

The function can be then used as follows:

```
num = count_steps
print "Number of steps:", num
```

11.5 Creating numerical variables

In Karel, numerical variables can be created in several different ways:

1. By setting them to an integer number. For example, new variable `a` is created and set to zero by typing
   ```
   a = 0
   ```
2. By setting them to `gpsx`. For example, new variable `var` is created and set to `gpsx` by typing
   ```
   var = gpsx
   ```
3. By setting them to `gpsy`. For example, new variable `pos1` is created and set to `gpsy` by typing
   ```
   pos1 = gpsy
   ```
4. Initialize them with an existing value. For example, if we already have a variable `var1`, then a new variable `var2` can be created as follows:
   ```
   var2 = var1
   ```
5. Initialize them with value returned by an existing function, as it was shown in the previous paragraph.

11.6 Changing values of numerical variables

The value of a numerical variable can be updated at any time by redefining it via one of the four options described in the previous paragraph. For example, let's say that Karel stands as shown in Fig. 13.

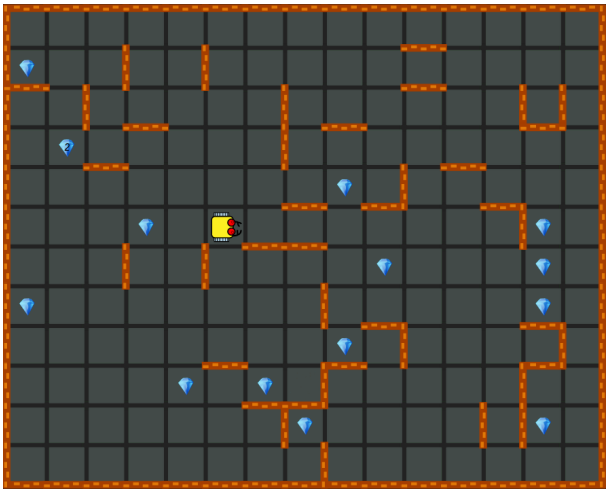


Fig. 13: Karel's initial position.

Then the program

```
a = gpsx
print "Start position:", a
repeat 5
    go
a = gpsx
print "End position:", a
```

will produce the following output:

```
Start position: 5
End position: 10
```

Another way to change the value of a numerical variable is to increase it by one or decrease it by one via the commands `inc()` and `dec()`, respectively. Commands `inc(a, num)` and `dec(a, num)` will increase / decrease the value of variable `a` by an integer number `num`.

Consider again Karel's initial position as shown in Fig. 13. Then the code

```
a = zero
while not wall
    go
    inc(a)
print "Went", a, "steps before reaching a wall."
```

will have the following output:

```
Went 7 steps before reaching a wall.
```

11.7 Comparison operations

Integer numbers and numerical variables can be compared using the standard operators `>`, `<`, `>=`, `<=`, `==`, `!=`, `<>`. In this order, they read "greater than", "less than", "greater than or equal to", "less than or equal to", "equal to", "not equal to" and "not equal to" (the last two operations have the same meaning). The result of each such operation is a logical value `true` or `false`, and it can be either used in a condition or conditional loop, or assigned to a variable. For example,

```
a = 1
b = 5
print a < b
```

yields the output

```
True
```

The code

```
a = 1
b = 5
c = a > b
print c
```

yields

```
False
```

The code

```
a = 0
while a <= 5
  print a
  inc(a)
```

yields

```
0
1
2
3
4
5
```

11.8 Lists

Lists are extremely useful data structures that can be used to store multiple objects in one variable at the same time. Such as, for example, integer variables, text strings, and even other lists. Objects in a list are ordered and they can be added to the end of a list using the `append()` function, accessed by their index, and deleted via the `pop()` function or the `del` command. Let us illustrate their usage on examples.

Creating a list

An empty list `U` is created via

```
U = []
```

Lists can be also created non-empty:

```
V = [1, 2, 3, 4, 5]
```

One can use variables when creating a list, as can be illustrated using the code

```
c = 100
W = [0, 50, c]
print W
```

whose output is

```
[0, 50, 100]
```

Integer numbers can be combined with strings:

```
X = [1, "Hello", 2]
```

Lists can contain other lists as their elements:

```
Y = [1, "Hello", 2, [1, 2, 3]]
```

One can print a list as expected:

```
print "This is the list Y:", Y
```

Output:

```
This is the list Y: [1, "Hello", 2, [1, 2, 3]]
```

Accessing list items by indices

Any list item can be accessed and either printed, assigned to a variable, or used in an operation, via its index. It is important to remember that the indices start from zero. In other words, `L[0]` is the first item in the list, `L[1]` is the second one, etc. For illustration, let us show a sample code

```
G = [8, 12, 16, 20]
print "First item:", L[0]
print "Second item:", L[1]
print "Third item:", L[2]
print "Fourth item:", L[3]
```

whose output is

```
First item: 8
Second item: 12
Third item: 16
Fourth item: 20
```

Appending items to a list

An arbitrary integer number, text string, or another list `obj` can be appended to the end of an existing list `L` as follows:

```
L.append(obj)
```

This can be illustrated using the code

```
L = [1, 11]
L.append(21)
print L
```

whose output is

```
[1, 11, 21]
```

Removing items via the `pop()` *function*

The `i`-th item can be deleted from a list `L` and assigned to a variable `x` via

```
x = L.pop(i)
```

Recall that indices start from zero. The usage can be illustrated using the code

```
L = ["Monday", "Tuesday", "Wednesday"]
day = L.pop(1)
print day
print L
```

whose output is

```
Tuesday
['Monday', 'Wednesday']
```

If the `pop()` function is used without an index, it removes and returns the last object of the list.

Deleting items via the `del` *command*

The purpose of the `del` command is similar to the `pop()` function except that the deleted object is destroyed (it cannot be assigned to a variable). The `i`-th item can be deleted from a list `L` via

```
del L[i]
```

For illustration, the output of the code

```
L = ["Monday", "Tuesday", "Wednesday"]
del L[0]
print L
del L[0]
print L
```

is

```
['Tuesday', 'Wednesday']
['Wednesday']
```

Length of a list

One can use the function `len(X)` to obtain the length of a list `X` – an integer number that can in turn be used to parse the list via a `repeat` command. The following sample code defines a list `M` consisting of four numbers `1, 2, 3` and `4` and prints all of them increased by two:

```
M = [1, 3, 5, 7]
n = len(L)
i = 0
repeat n
    c = M[i]
    print inc(c, 2)
    inc(i)
```

Storing the robot's path in a list

Lists can be used to store the robot's path. The following is a simple program that only tells the robot to go straight to the nearest wall (this part does not really matter) and store the GPS coordinates in a list `L`:

```
L = []
while not wall
    L.append([gpsx, gpsy])
    go
L.append([gpsx, gpsy])
print "Path: ", L
```

In fact, each time we are appending a list consisting of the horizontal and vertical GPS coordinates – there is no other way to represent two numbers in one variable unless the variable is a list. When the robot's initial position is as shown in Fig. 14,

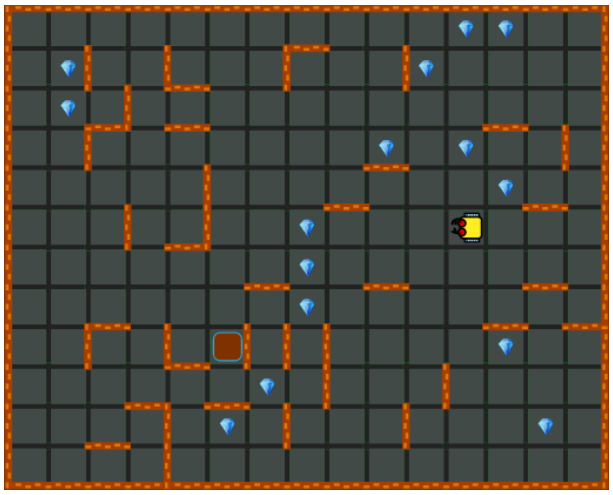


Fig. 14: Robot stands at position [11, 6] facing west.

then the output of the above program is

```
Path: [[11, 6], [10, 6], [9, 6], [8, 6], [7, 6], [6, 6], [5, 6]]
```

To keep Karel's language simple, we do not allow double indices. Nevertheless there is a simple way to get to the items of lists that are contained in other lists:

```
a = L[0]
print "Horizontal coordinate of initial position:", a[0]
print "Vertical coordinate of initial position:", a[1]
a = L[1]
print "Horizontal coordinate after one step:", a[0]
print "Vertical coordinate after one step:", a[1]
...
```

11.9 Local and global variables

Note that a variable that is defined inside a function is *local to that function*, meaning that it can be used in the function only. If we attempt to use it outside, an error is thrown. For illustration, the code

```
def myfunction
  a = 1

myfunction
print a
```

throws an error message

```
Unknown variable/procedure "a"
```

Although using local variables might seem constraining, in reality it helps us to keep our code fit. It is a very good habit to keep variables in our programs as local as possible. More about local and global variables will be said in Part II, Subsection 5.5.

12 Logic

12.1 Objectives

– Learn to work with elementary and more complex logical expressions.

12.2 Logical expressions

As we already know, logical expressions are expressions that can be answered with either `True` or `False`. We say that the `True` or the `False` is their value. Here are some real-life examples, try to answer them with `True` or `False`:

– "I am 15 years old."
– "My dad is a teacher."
– "My school's name is Coral Academy."

And here are some Karel examples:

– wall (`True` if the robot is facing a wall, `False` otherwise)
– home (`True` if the robot is home, `False` otherwise)
– gem (`True` if the robot stands on a gem, `False` otherwise)
– north (`True` if the robot is facing North, `False` otherwise)
– empty (`True` if the robot does not have any gems on him, `False` otherwise)

12.3 Logical variables

In programming as well as in real life we often deal with logical expressions that are fairly complex. Often we use two or more simple logical expressions in one sentence, and moreover combine them with logical operations *and*, *or* or *not*.

For example, the sentence "I will go skiing on Saturday if weather is good and if Michael goes as well." includes three simple logical expressions. Let's call them for brevity

A = "I will go skiing on Saturday."
B = "The weather is good."

C = "Michael goes as well."

There is a logical operation *and* between the expressions B and C.

The original sentence can be written briefly as "if (B and C) then A". We love this kind of brevity in programming! A, B and C are *logical variables*. Logical variables can only represent `True` or `False`, and their purpose is to ease the operation with longer expressions. Regarding the original sentence, we could go one step further and define a new logical variable

D = B *and* C.

Then, the sentence would become just "if D then A"!

12.4 Logical operations

It is worth mentioning the following properties of the logical operation *and*:

(A *and* B) is `True` only if both A and B are `True`. Otherwise A *and* B is `False`.

The logical operation *or* has the following properties:

(A *or* B) is `True` if at least one of A, B is `True`. If both A. B are `False`, (A *or* B) is `False`.

We also use the logical operation *not* with the following property:

(*not* A) is `True` if A is `False` and vice versa.

13 Appendix - Overview of Functionality by Level

13.1 Level 0 (Section 3)

In Level 0, Karel can be guided by clicking on buttons:

- Go ... make one step forward.
- Left ... turn left.
- Right ... turn right.
- Put ... put a gem on the ground.
- Get ... pick up a gem from the ground.

13.2 Level 1 (Section 4)

In Level 1, Karel can be guided by typing the commands:

- `go` ... make one step forward.
- `left` ... turn left.
- `right` ... turn right.
- `put` ... put a gem on the ground.
- `get` ... pick up a gem from the ground.

13.3 Level 2 (Sections 5 – 10)

New commands:

- `if - else` ... condition.
- `repeat` ... counting loop (repeat an action a given number of times).
- `while` ... conditional loop (repeat an action while a condition is satisfied).
- `def` ... define a new command.

Additional new keywords:

- `wall` ... sensor that checks True if the robot faces a wall.
- `gem` ... sensor that checks True if there is a gem within the robot's reach.
- `empty` ... sensor that checks True if the robot's bag with gems is empty.
- `home` ... sensor that checks True if the robot is at home.
- `north` ... sensor that checks True if the robot faces North.

New functionality:

- Recursion (a command can make a call to itself).

13.4 Level 3 (Sections 11, 12)

New keywords:

- `print` ... print strings and variables.
- `gpsx` ... GPS coordinate in the horizontal direction.
- `gpsy` ... GPS coordinate in the vertical direction.
- `a = 0` ... create a new variable `a` and initialize it with zero (or another integer number). For additional ways to initialize variables see Subsection 11.5.
- `inc(a)` ... increases the value of variable `a` by one.
- `inc(a, value)` ... increases the value of variable `a` by `value`.
- `dec(a)` ... decreases the value of variable `a` by one.
- `dec(a, value)` ... decreases the value of variable `a` by `value`.

- `rand` ... random command (returns randomly True or False).
- `return` ... returns a value, to be used in functions.
- `and` ... binary logical *and*.
- `or` ... binary logical *or*.
- `not` ... unary logical *not*.
- `len(L)` ... length of list `L`.
- `L[i]` ... object at position `i` in list `L`. Note: `L[0]` is the first item in the list.
- `L.append(x)` ... appends `x` to the end of list `L`.
- `x = L.pop()` ... removes the last item of list `L` and assigns it to `x`.
- `del L[i]` ... removes from list `L` object at position `i`.

New functionality:

- Numerical and logical (Boolean) variables.
- Complex logical expressions.
- Functions that return values.
- Lists.

Part II

Introduction to Python

1 Introduction

1.1 Objectives

- Learn the difference between compiled and interpreted programming languages.
- Learn basic facts about Python and its powerful accompanying libraries.
- Understand how Python programming works in the cloud setting.
- Write and run your first Python program.

1.2 Compiled and interpreted programming languages

Compilation is a process where human-readable *source code (text)* is translated by means of a *compiler* and a *linker* into a *binary (executable) file* which then can be run on the concrete computer. The same source code, compiled on different hardware architectures, yields different binaries.

Interpreted (scripting) programming languages are more recent than the compiled ones. A program that is written using an interpreted language is "read" (parsed) at runtime – it is not compiled and there are no binaries. Programs written using compiled languages are usually more efficient than programs written using the interpreted ones because the former take better advantage of the underlying hardware. On the other hand, interpreted languages are usually more universal and easier to use. Compiled programming languages include Pascal, C, C++, Fortran, and many others. Few examples of interpreted languages are Python, Lua, Perl and Ruby.

1.3 Basic facts about Python

Python is a powerful modern programming language that is used in many areas of business, engineering, and science. Its interpreted character along with a very intuitive syntax make Python an ideal language for beginners.

Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands. Rossum also gave it a name originated in his favorite television series *Monty Python's Flying Circus*. Python 2.0 was released in October 2000 and Python 3.0 in December 2008. Python was awarded the TIOBE *Programming Language of the Year* award twice (2007, 2010), which is given to the language with the greatest growth in popularity over the course of a year.

Python is a multi-paradigm programming language. Rather than forcing programmers to adopt a particular style of programming, it permits several styles: *structured (procedural) programming* and *object-oriented programming* are fully supported, and there are a number of language features which support *functional programming*.

In this course you will learn the most important aspects of the language and you will be able to use the programming language to solve a large variety of problems. Depending on your

objectives, this course may be all you will ever need. We hope that you will like Python and want to learn more – and there is much more to learn out there. References to materials covering more advanced topics and object-oriented programming are given in Section 15.

1.4 Libraries - Python programmer's strong companion

Python comes with a powerful *standard library* that contains plentiful functionality to ease work with constants, types, functions, exceptions, strings, data, mathematics, cryptography, parallel processing and other things. Python also has a number of scientific libraries. Most important of them are Scipy, Numpy, Pylab, Matplotlib, and Sympy.

1.5 How does Python programming work in the cloud environment?

In the browser, Python code is typed into one or more code cells in the *Python worksheet*. The code is sent (as a text string) to a remote server where it is interpreted, and the results are sent back to your computer / laptop / tablet, and displayed in your web browser. Let's try this!

1.6 Launching a Python project

There are multiple ways to launch a Python project. First, it is possible to clone and experiment with many existing displayed Python projects via File Manager → Project → Clone. New (empty) Python project can be launched via the *Programming* menu or via File Manager → Project → New → Python. An empty Python worksheet is shown in Fig. 15.

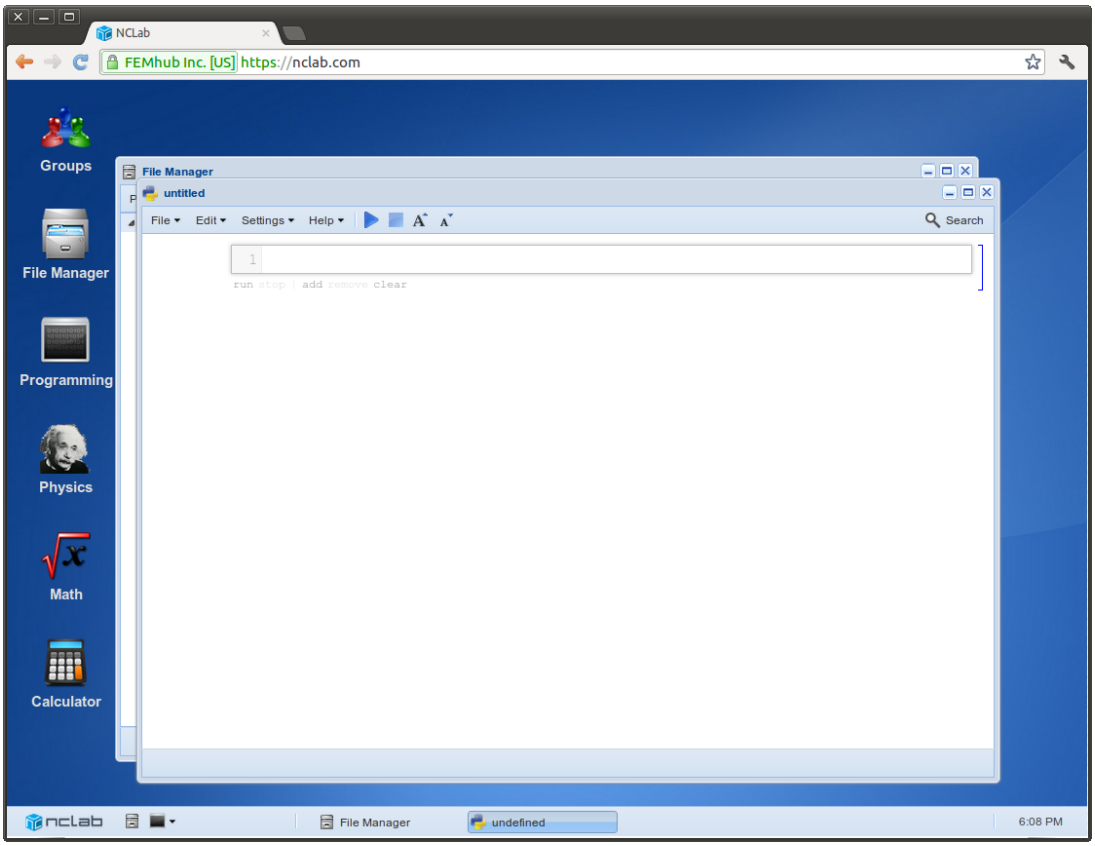


Fig. 15: Launching a new Python project.

1.7 Hello, World!

Click into the code cell and type `print "Hello, World!"`. Then click on `run` link under the code cell, and the text "Hello, World!" will be displayed in a new yellow *output cell* as shown in Fig. 16.

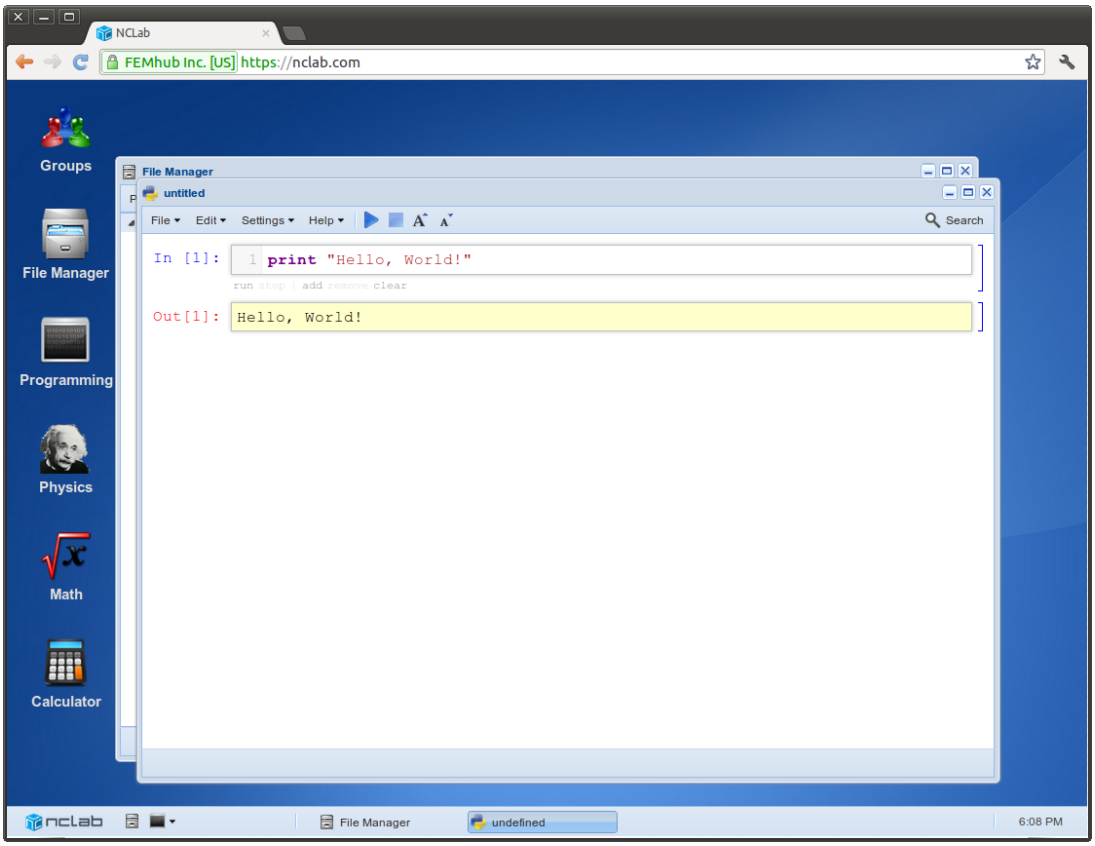


Fig. 16: Response received from the cloud is shown in a yellow output cell.

1.8 Code, output and text cells

As in Karel, there are three types of cells in the Python worksheet - code cells, output cells, and descriptive text cells. code cells as well as text cells can be added via the *Edit* menu. New code cells can be also added easily by clicking on `add` under an existing code cell. code cells and text cells can be removed by clicking on `remove` under them. The easiest way to remove a selected output cell is to click on it with the mouse and press DELETE on the keyboard. All output cells can be removed at once via *Remove all output* in the *Edit* menu.

1.9 Various ways to run programs

Pressing the green arrow button will run all code cells in the project. Clicking on the green arrow under an code cell will run only the contents of that code cell. Same effect can be achieved by

holding CTRL and pressing ENTER. Holding SHIFT and pressing ENTER will evaluate the current code cell and create an additional one under it.

2 Using Python as a Calculator

2.1 Objectives

- Learn how to use Python for advanced math operations.
- Plot graphs of functions of one and two variables, as well as parametric 2D and 3D curves.

The Python worksheet can be used as a powerful calculator and graphing utility. Let us begin with the simplest math operations.

2.2 Addition, subtraction

Launch a new Python project and in the code cell type:

```
3 + 6
```

Then click on the `run` link under the code cell. The output should be displayed quickly:

```
9
```

You can add real numbers too,

```
3.2 + 6.31
```

Output:

```
9.51
```

Two numbers can be subtracted as follows,

```
7.5 - 2.1
```

Output:

```
5.4
```

2.3 Multiplication

Multiplication is done using the '$\star$' symbol as in

```
3 * 12
```

Output:

```
36
```

Of course, real numbers can be multiplied as well:

```
3.7 * 12.17
```

Output:

```
45.029
```

2.4 Division

With division, we need to be a bit careful. Look at this:

```
30 / 5
```

Output:

```
6
```

And then look at this:

```
33 / 5
```

Output:

```
6
```

In all major computer languages including C, C++, Fortran, Python and others:

The result of division of two integers is an integer!

So, when dividing two integers, everything behind the decimal point is lost. In order to stay on the safe side, before you perform division of two numbers, always convert at least one of them to a real number. This is simple: While `5` is an integer, `5.` and `5.0` are reals. It is also possible to say `float(5)`. In fact this is the best way since it also works for variables. Such as, when we are dividing `a / b`, to make sure that the result will be correct also for integers, we type `float(a) / b`. When at least one number is a real, then the result of the division is a real. Variables will be discussed shortly.

2.5 Powers

Sometimes we need to use exponents, such as in 2^4. Python has a double-star symbol `**` for this:

```
2**4
```

Output:

```
16
```

2.6 Modulo

The last of the common arithmetic operations is *modulo*. Recall that this is the remainder after integer division. In Python modulo is done using the per cent symbol:

```
6 % 4
```

Output:

```
2
```

Of course, one can use brackets:

```
5 * (7 - 3)
```

Output:

```
20
```

2.7 Order of operations

Python respects the standard order of operations:

- Round brackets have the highest priority,
- then exponentiation,
- then multiplication and division (same priority),
- the lowest priority have addition and subtraction.

Note that no other brackets such as `{ }` and `[ ]` are admissible in mathematical expressions since they have a different function in the programming language.

To illustrate the priority of operations, we can try the following:

```
3**4 / 27 * 5 + 3 * 5
```

Output:

```
30
```

2.8 Using empty characters makes you code better readable

Your code will be much better readable if you use empty characters on either side of arithmetic symbols, as well as after commas. Hence, you should never write things like `3**4/27*5+3*5`.

2.9 Using mathematical functions

In order to calculate square roots, exponentials, sines, cosines, tangents, and many other math functions, the best way is to import Numpy. Numpy is a powerful Python library for numerical computations. To import it, just include the following line in your code:

```
from numpy import *
```

Here the symbol '`*`' stands for "everything". If you wanted to import just one or two functions, you could do that as well by just giving their names, separated by commas.

After Numpy is imported, we can calculate, for example, e^2:

```
exp(2)
```

Output:

```
7.3890560989306504
```

Elementary functions (and constants) that one can import from Numpy are listed below. We also show their arguments for clarity, but the functions are imported without them. For example, the absolute value function is imported via `from numpy import abs`.

pi	π
abs(x)	absolute value of x
arccos(x)	inverse cosine of x
arccosh(x)	inverse hyperbolic cosine of x
arcsin(x)	inverse sine of x
arcsinh(x)	inverse hyperbolic sine of x
arctan(x)	inverse tangent of x
arctanh(x)	inverse hyperbolic tangent of x
arctan2(x_1, x_2)	arc tangent of x_1/x_2 choosing the quadrant correctly
cos(x)	cosine of x
cosh(x)	hyperbolic tangent of x
exp(x)	e^x
log(x)	natural logarithm of x
pow(a, b)	a^b (same as "a**b")
sin(x)	sine of x
sinh(x)	hyperbolic sine of x
sqrt(x)	square root of x
tan(x)	tangent of x
tanh(x)	hyperbolic tangent of x

2.10 Complex numbers

Complex numbers are always represented as two floating point numbers, the real and imaginary part. Appending '`j`' or '`J`' to a real number makes it imaginary:

```
1j * 1J
```

Output:

```
(-1+0j)
```

This is one way to define complex numbers:

```
1 + 3j
```

Output:

```
(1+3j)
```

Another way is to use the command `complex`:

```
complex(1, 3)
```

Output:

```
(1+3j)
```

All arithmetic operations that are used for real numbers can be used for complex numbers as well, for example:

```
(1 + 2j) / (1 + 1j)
```

Output:

```
(1.5+0.5j)
```

To extract the real and imaginary parts of a complex number `z`, use `z.real` and `z.imag`. Use `abs()` to get the absolute value:

```
a = 3 + 4j
a.real
a.imag
abs(a)
```

Output:

```
3
4
5
```

3 More on Functions

3.1 Objectives

- Review and refine your knowledge of functions.
- Learn to use default arguments and return multiple values.

Custom *functions* in Python are the same thing as in Karel, but in Python they can moreover *accept arguments*. Recall that functions in Karel were defined to make some functionality easily reusable – in Python they are defined for the same reason.

3.2 Defining new functions

Definition of a new function begins with the keyword `def` as in Karel. In Python, we moreover use round brackets for arguments and a colon at the end of the line. For example, the following function adds two numbers and returns the result:

```
def add(a, b):
    return a + b
```

The round brackets in the function definition are mandatory even if no arguments are passed, and the `return` statement can be omitted if not needed. The two lines of code above are a *function declaration* – they do not produce any output. In order to call the function, we need to write one additional line, such as:

```
print "5 + 3 is", add(5, 3)
```

This will produce the following output:

```
5 + 3 is 8
```

To show another example, the following function does not take any arguments and it does not return any results:

```
def hello():
    print "Hello!"
```

3.3 Passing arbitrary arguments

Python does not require that we specify the type of function arguments. What does it mean? The above function `add(a, b)` works for real numbers, complex numbers, vectors, strings, and any other objects where the operation '+' is defined. Let's try this:

```
def add(a, b):
    return a + b

word1 = "Good "
word2 = "Evening!"
print add(word1, word2)
```

Output:

```
Good Evening!
```

Things like this make Python very intuitive and easy to use. In conventional languages such as C or C++, we would have to define different functions: `add_real_numbers(a, b)` for real numbers, `add_complex_numbers(a, b)` for complex numbers numbers, `add_strings(a, b)` for strings, etc. In C, they would be as follows:

```
double add_real_numbers(double a, double b)
{
  return a + b;
}
```

and

```
# include <complex.h>
complex add_complex_numbers(complex a, complex b)
{
  return a + b;
}
```

and

```
void add_strings(const char* a_in, const char* b_in, char* c_out)
{
  strcpy(c_out, a_in);
  strcat(c_out, b_in);
  return;
}
```

3.4 Returning multiple values

Python functions can return multiple values which often comes handy. For example, the following function returns the second, third, and fourth powers of a number:

```
def powers(a):
    return a**2, a**3, a**4
```

This is how the function is used:

```
var1 = 2
print "Powers are", powers(var1)
```

Output:

```
Powers are (4, 8, 16)
```

We can also store the returned values in three separate variables:

```
var1 = 3
p2, p3, p4 = powers(var1)
print "Powers are", p2, p3, p4
```

Output:

```
Powers are 9 27 81
```

Look how complicated the same program would be in C/C++:

```
void powers(double a, double* p2, double* p2, double* p3)
{
  *p2 = pow(a, 2);
  *p3 = pow(a, 3);
  *p4 = pow(a, 4);
  return;
}

int main()
{
  double p1, p2, p3;
  double a = 2;
  powers(a, &p2, &p2, &p3);
  printf("Powers are %g, %g, %g.\n", p2, p3, p4);
}
```

3.5 Using default arguments

Holland is the most bicycle friendly country in the world. Imagine that you work for the Holland Census Bureau. Your task is to ask 10000 people how they go to work, and enter their answers into a database. The program for entering data into the database was written by one of your colleagues, and it can be used as follows:

```
add_database_entry("John", "Smith", "walks")
```

or

```
add_database_entry("Louis", "Armstrong", "bicycle")
```

or

```
add_database_entry("Jim", "Bridger", "horse")
```

etc. Since you are in Holland, it can be expected that 99% of people are using the bicycle. In principle you could call the function `add_database_entry()` to enter each answer, but with 9900 bicyclists out of 10000 respondents you would have to type `"bicycle"` MANY times.

Fortunately, Python offers a smarter way to do this. We can define a new function

```
def enter(first, last, transport = "bicycle"):
    enter_into_database(first, last, transport)
```

This is a simple (*"thin"*) *wrapper* to the function `add_database_entry()` that allows us to omit the third argument in the function call and autocomplete it with a *default argument* `"bicycle"`. In other words, now we do not have to type `"bicycle"` for Louis Armstrong or any other bicyclist:

```
enter("Louis", "Armstrong")
```

Only if we meet a rare someone who uses a car, we type

```
enter("Niki", "Lauda", "car")
```

Another example: Let us return to the function `add()`, and let us imagine that very often (but not always) the second number that we are adding is 10. Then it makes sense to write the function `add()` as follows:

```
def add(a, b = 10):
    return a + b
```

The function will work as before when called with two arguments:

```
A = 5
B = 1
add(A, B)
```

Output:

```
6
```

But it can be also called with the second argument omitted:

```
A = 5
add(A)
```

Output:

```
15
```

Some rules to remember:

- Default arguments need to be introduced **after** standard (non-default) arguments. In other words, the following code will result into an error:

  ```
  def add(a = 5, b):
      return a + b
  ```

  This is the error message:

  ```
    File "<nclab>", line 1
  SyntaxError: non-default argument follows default argument
  ```

- If multiple default arguments are present, they have to follow the non-default ones. If the number of arguments in the function call is less than the total number of default and non-default arguments, then first all non-default arguments are assigned, and then the default ones from left to right. To illustrate this, assume the function:

  ```
  def add(x, a = 2, b = 3):
      return x + a + b
  ```

  This function can be called as follows:

  ```
  print add(1)
  ```

  In this case, the value `1` is assigned to `x` and `a` and `b` take the default values. The output will be

  ```
  6
  ```

  When the function is called with just two values,

  ```
  print add(5, 6)
  ```

  then the value `5` is assigned to `x`, `6` to `a`, and `b` takes the default value `3`. The output is

  ```
  14
  ```

  However, it is always a good idea to be as transparent as possible. Therefore a better code is

  ```
  print add(1, a = 6)
  ```

  Then the result is the same, `14`, but it is clear even to non-experts what is going on. Moreover, in this way we can also define the value of `b` without having to define the value of `a`:

  ```
  print add(1, b = 6)
  ```

  Now `x` is `1`, `a` has the default value `2` and the result is `9`.

4 Plotting

4.1 Plotting functions of one variable

Plotting can be done via the Pylab library. The Pylab `plot` command takes two arrays: x-coordinates and y-coordinates of points on a curve. Between the points, the curve is interpolated linearly. Let us illustrate this on a simple example with just five points [0, 0], [1, 2], [2, 0.5], [3, 2.5] and [4, 0]:

```
from pylab import *
x = [0,0, 1.0, 2.0, 3.0, 4.0]
y = [0.0, 2.0, 0.5, 2.5, 0]
clf()
plot(x, y)
lab.show()
```

The commands `clf()`, `plot()` and `lab.show()` do clear the canvas, plot the graph, and show the graph, respectively. The output is shown in Fig. 17.

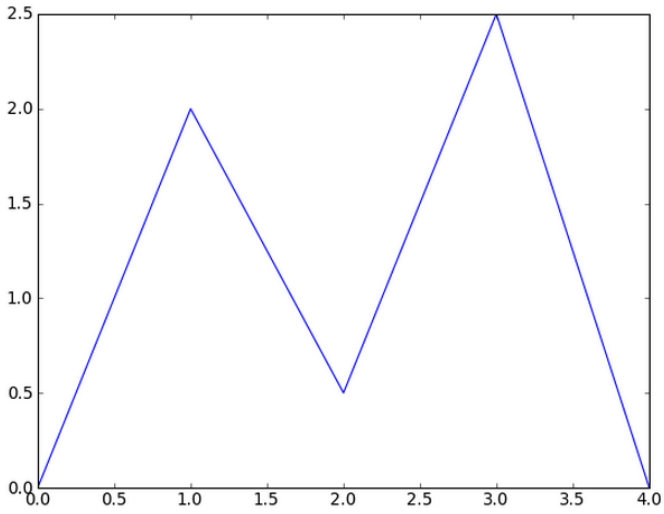


Fig. 17: Piecewise-linear curve with five points.

In the following we will discuss more options and show some useful techniques. Let's say, for example, that we want to plot the function $f(x) = \sin(x)$ in the interval $(0, 2\pi)$. The array of x-coordinates of equidistant points between 0 and π can be created easily using the command `linspace`:

```
from numpy import *
x = linspace(0, 2*pi, 100)
```

Here 100 means the number of points in the division (including endpoints). Increasing this number will improve the resolution and vice versa. Next, the array of y-coordinates of the points is obtained via

```
y = sin(x)
```

The last part we already know:

```
clf()
plot(x, y)
lab.show()
```

The output is shown in Fig. 18.

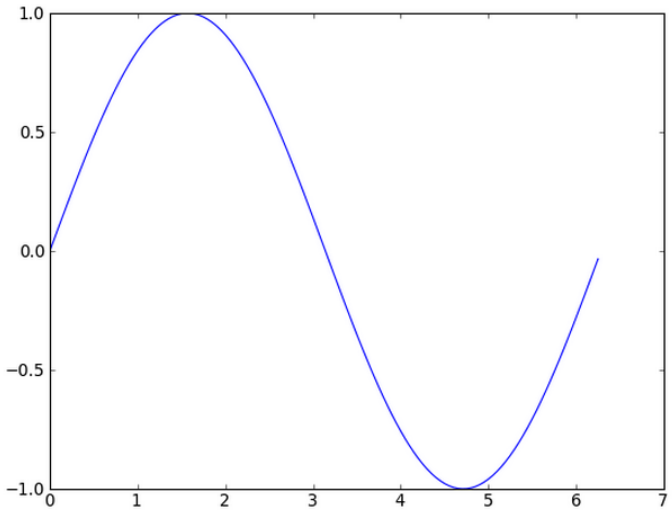


Fig. 18: Plotting $\sin(x)$ in interval $(0, 2\pi)$ with subdivision step 0.05.

4.2 Labels, colors, and styles

The plot can be made nicer by adding a label, and also the color and the line style can be changed. Let us start with adding a label:

```
plot(x, y, 'b-', label = "Solid blue line")
legend()
lab.show()
```

The output is shown in Fig. 19.

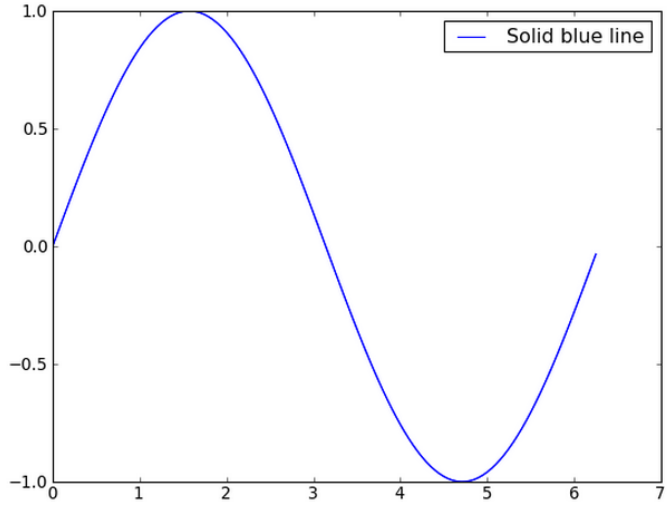


Fig. 19: Adding a label.

Next let us change the color to red and line style to dashed:

```
clf()
plot(x, y, 'r--', label = "Dashed red line")
legend()
lab.show()
```

The output is shown in Fig. 20.

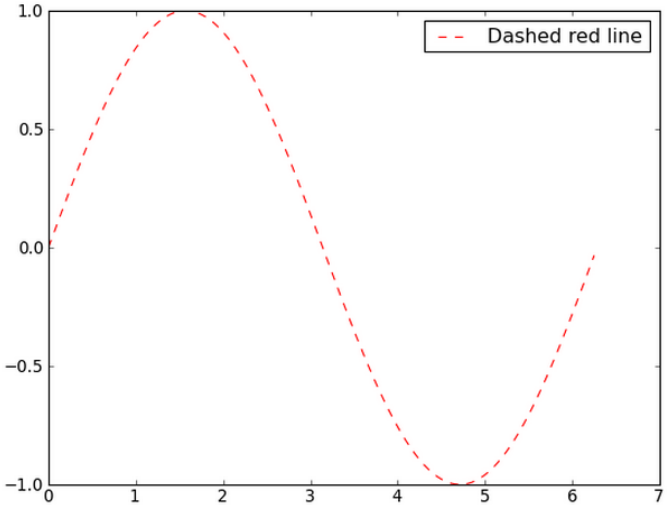


Fig. 20: Same graph using dashed red line.

The graph can be plotted using green color and small dots rather than a solid or dashed line:

```
clf()
plot(x, y, 'g.', label = "Dotted green line")
legend()
lab.show()
```

The output is shown in Fig. 21.

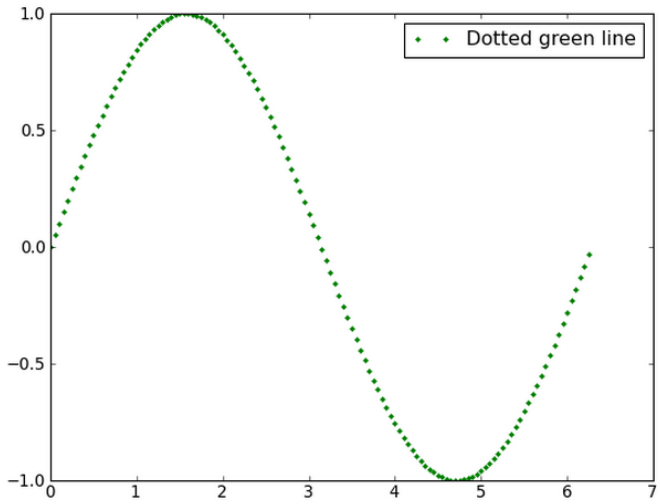


Fig. 21: Same graph using dotted green line.

Last let us stay with green color but make the dots larger:

```
clf()
plot(x, y, 'go', label = "Bigger green dots")
legend()
lab.show()
```

The output is shown in Fig. 22.

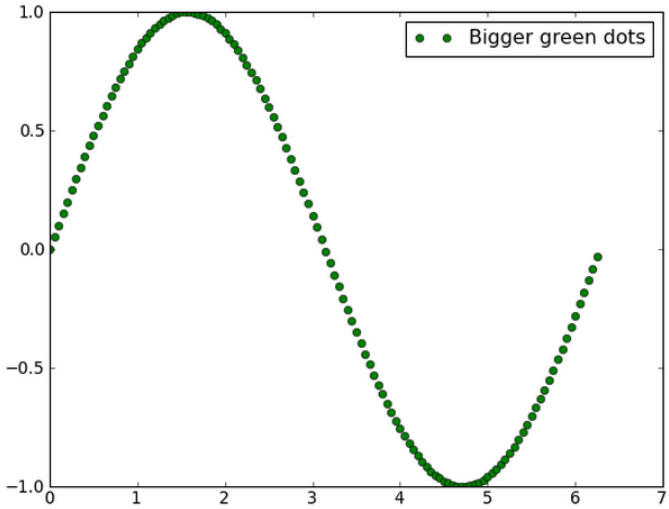


Fig. 22: Same graph using large green dots.

4.3 Scaling axes and showing grid

In the previous plots, function graphs were fitted into the display window, which means that the horizontal and vertical axes were scaled differently. This can be changed by including the `axis('equal')` command after calling `plot()`. Also, grid can be displayed by using the `grid()` command:

```
from numpy import *
from pylab import *
x = linspace(0, 2*pi, 100)
y = sin(x)
clf()
plot(x, y, label="sin(x)")
axis('equal')
grid()
legend()
lab.show()
```

The output is shown in Fig. 23.

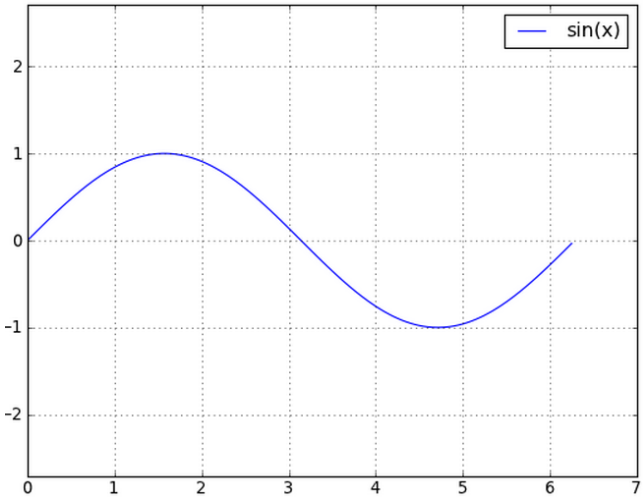


Fig. 23: Scaling axes equally and showing grid.

The scaling of axes can be returned to the automatic fit option via `axis('auto')` if needed.

4.4 Adjusting plot limits

Plot limits can be set using the `xlim()` and `ylim()` functions after `plot()`. For example, we can stretch the sine function from Fig. 23 to span the entire width of the canvas as follows:

```
from numpy import *
from pylab import *
x = linspace(0, 2*pi, 100)
y = sin(x)
clf()
plot(x, y, label="sin(x)")
xlim(0, 2*pi)
axis('equal')
grid()
legend()
lab.show()
```

The output is shown in Fig. 24.

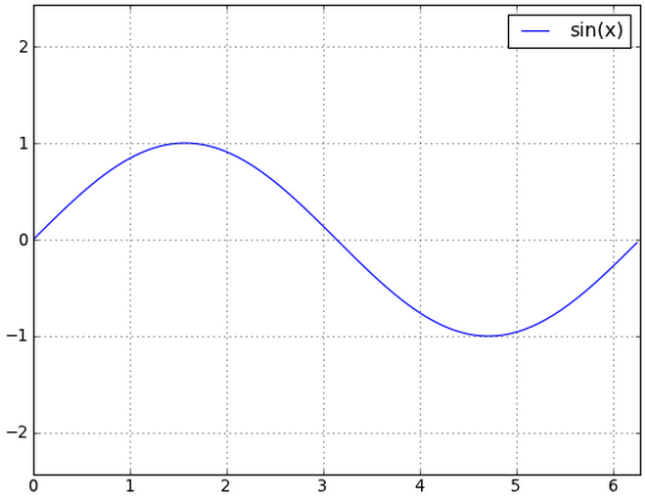


Fig. 24: Setting plot linits on the horizontal axis to 0 and 2π.

4.5 Plotting multiple functions at once

This can be done very easily, just do not use the `clf()` command between the plots and you can have as many graphs in one figure as needed. Let us do three:

```
from numpy import *
from pylab import *
x = linspace(0.5, 5, 100)
y1 = 1./x
y2 = 1. / (1 + x**2)
y3 = exp(-x)
axis=("equal")
clf()
plot(x, y1, label="y1")
plot(x, y2, label="y2")
plot(x, y3, label="y3")
legend()
lab.show()
```

The output is shown in Fig. 25.

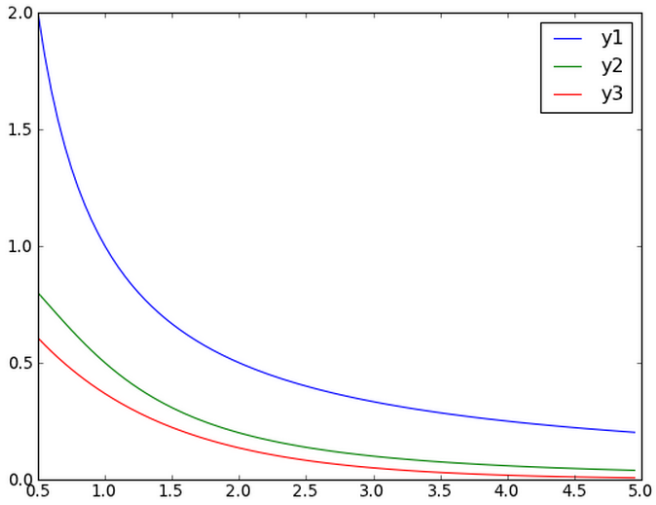


Fig. 25: Plotting graphs of functions $1/x$, $1/(1+x^2)$ and e^{-x} in interval $(0.5, 5)$.

The `plot()` command in Pylab is much more powerful, we just saw a small fraction of its functionality. For a complete list of options visit the Pylab page `http:// www.scipy.org/PyLab`.

4.6 Plotting parametric 2D curves

The concept of plotting based on two arrays of x and y coordinates allows us to do much more than only plot graphs of functions of one variable. We can easily plot more general curves such as circles, spirals and others. Let us illustrate this on a spiral that is parameterized by

$$x(t) = t\cos(t), \quad y(t) = t\sin(t)$$

in the interval $(0, 10)$ for t. The complete code is

```
from pylab import *
from numpy import *
t = linspace(0, 10, 100)
x = t*cos(t)
y = t*sin(t)
clf()
plot(x, y)
lab.show()
```

The output is shown in Fig. 26.

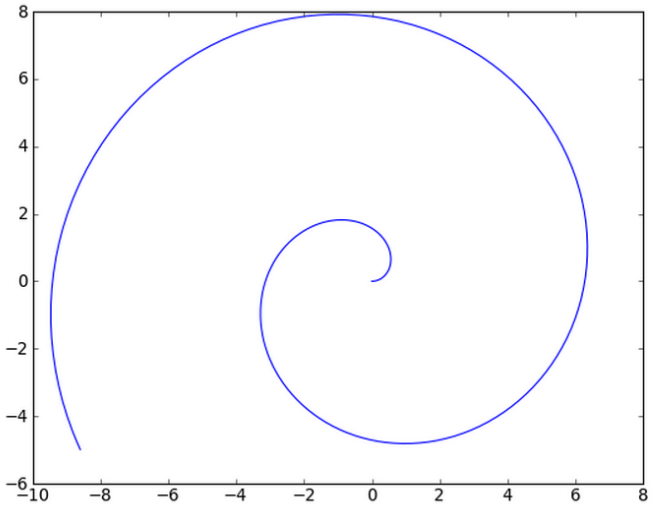


Fig. 26: Plotting a spiral.

4.7 Plotting parametric 3D curves

For 3D plots it is practical to use the `mplot3d` toolkit of the Python library Matplotlib. Let us begin with parametric 3D curves since this is analogous to how we handled planar curves in the previous paragraph. 3D curves are sequences of linearly interpolated 3D points represented via three arrays of x, y and z coordinates. As an example we will plot the curve $x(t), y(t), z(t)$ where

$$x(t) = (1 + t^2)\sin(2\pi t), \quad y(t) = (1 + t^2)\cos(2\pi t), \quad z(t) = t,$$

and where the parameter t lies in the interval $(-2, 2)$.

```
# Import Numpy and Matplotlib:
from numpy import *
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Set legend font size (optional):
mpl.rcParams['legend.fontsize'] = 15

# Setup the 3D plot:
fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
# Define interval for parameter 't' and its
# division:
t = linspace(-2, 2, 100)

# Define the curve:
x = (1 + t**2) * sin(2 * pi * t)
y = (1 + t**2) * cos(2 * pi * t)
z = t

# Plot the curve
ax.plot(x, y, z, label='Parametric 3D curve')
ax.legend()

# Show the plot:
lab.show()
```

The output is shown in Fig. 27.

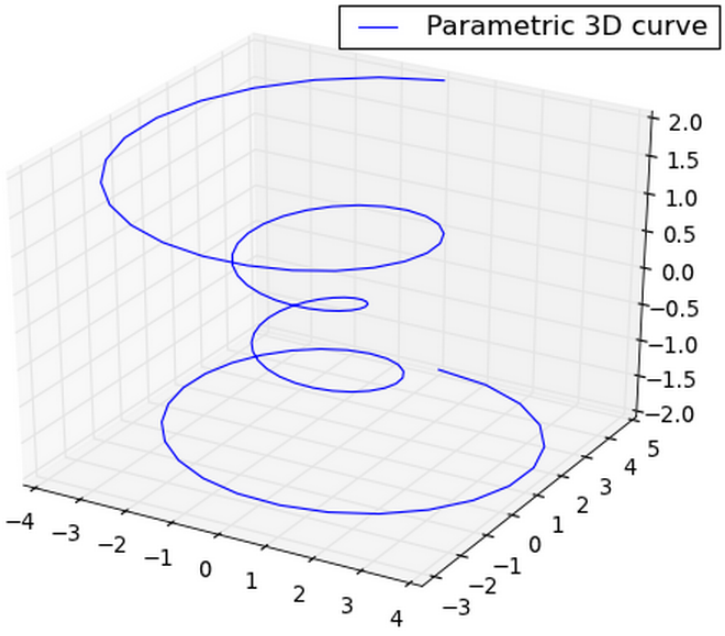


Fig. 27: Plotting a parametric 3D curve.

4.8 Plotting functions of two variables

There are several ways to plot graphs of functions of two variables, first let us do this with Matplotlib's `mplot3d` toolkit, then we will use WebGL. We will plot the graph of the function

$$f(x, y) = \sin(x)\sin(y)$$

in the square $(-\pi, \pi) \times (-\pi, \pi)$.

```
# Import Numpy and Matplotlib:
from numpy import *
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt

# Setup the 3D plot:
fig = plt.figure()
ax = fig.gca(projection='3d')

# Define intervals on the 'x' and 'y' axes and
# their divisions:
x = linspace(-pi, pi, 30)
y = linspace(-pi, pi, 30)

# Create Cartesian grid:
X, Y = meshgrid(x, y)

# Calculate values at grid points:
Z = sin(X) * sin(Y)

# Plot the data:
ax.plot_wireframe(X, Y, Z, rstride=1, cstride=1)

# Show the plot:
lab.show()
```

The parameters `rstride` (row stride) and `cstride` (column stride) can be used to make the plot coarser when they are set to 2, 3, etc. The output is shown in Fig. 28.

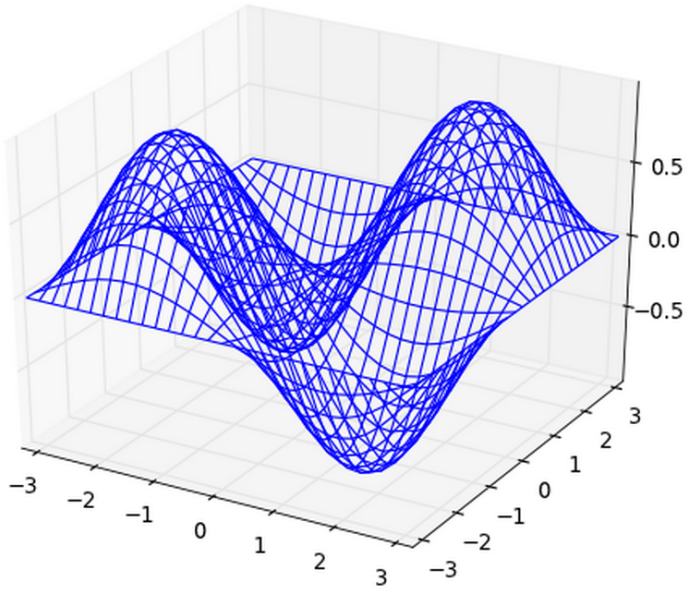


Fig. 28: Wireframe plot of the function $f(x, y)$.

Solid surface plot can be obtained by replacing in the above code `plot_wireframe` with `plot_surface`. The output is shown in Fig. 29.

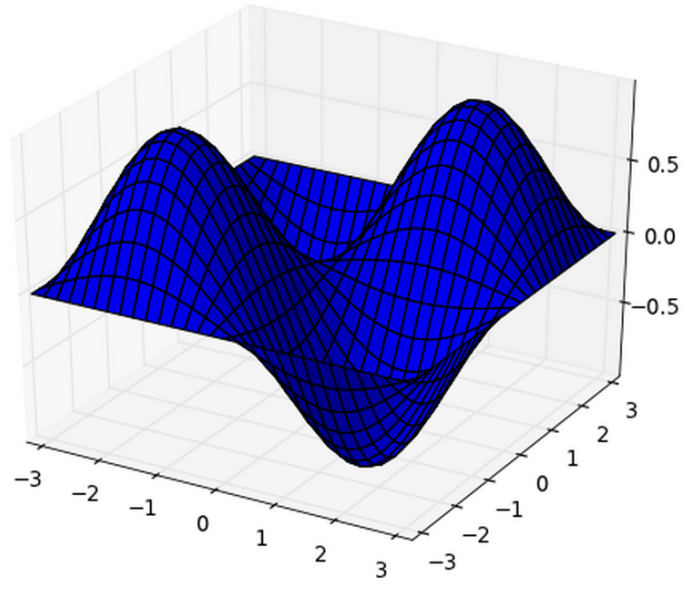


Fig. 29: Surface plot of the function $f(x, y)$.

Contour plot can be obtained by replacing in the above code the line

```
ax.plot_wireframe(X, Y, Z, rstride=1, cstride=1)
```

with

```
cset = ax.contour(X, Y, Z)
ax.clabel(cset)
```

The output is shown in Fig. 30.

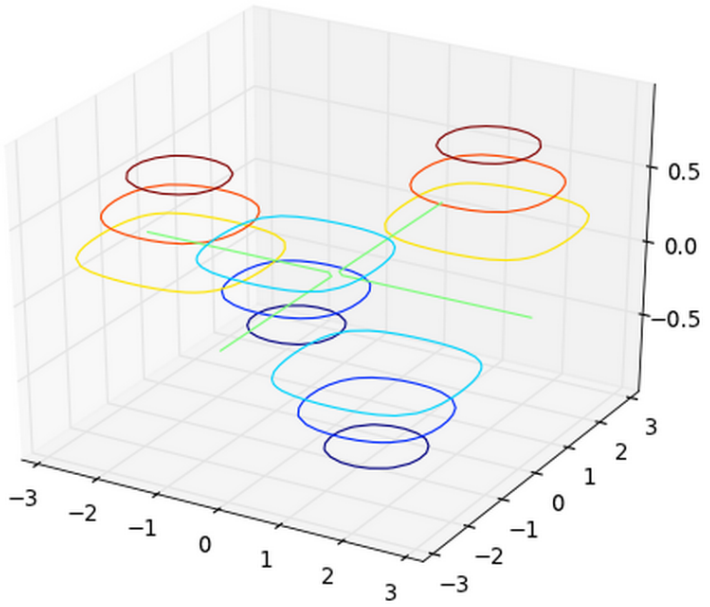


Fig. 30: Contour plot of the function $f(x, y)$.

We have only mentioned the most basic Matplotlib's functionality, for more options we refer to the tutorial at `http://matplotlib.sourceforge.net/mpl_toolkits/mplot3d`.

4.9 Plotting functions of two variables with WebGL

For this functionality, your browser has to support WebGL (most of modern browsers do, with the exception of Internet Explorer). See the introductory tutorial *Welcome to NCLab* on the page `http://femhub.com/nclab-tutorials/` for detailed instructions on how to enable WebGL. In the following example, we plot the graph of the function

$$f(x, y) = \sin(\sqrt{x^2 + y^2})$$

in the square $(0, 10) \times (0, 10)$.

```
from numpy import sin, sqrt, arctan

# Define intervals on the x and y axes.
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0

# Define the corresponding divisions.
nx = 100
ny = 100

# Define the function:
def f(x, y):
    return sin(sqrt(x**2 + y**2))

# Render the surface using WebGL.
lab.surface((x0, x1, nx), (y0, y1, ny), f)
```

The output is shown in Fig. 31.

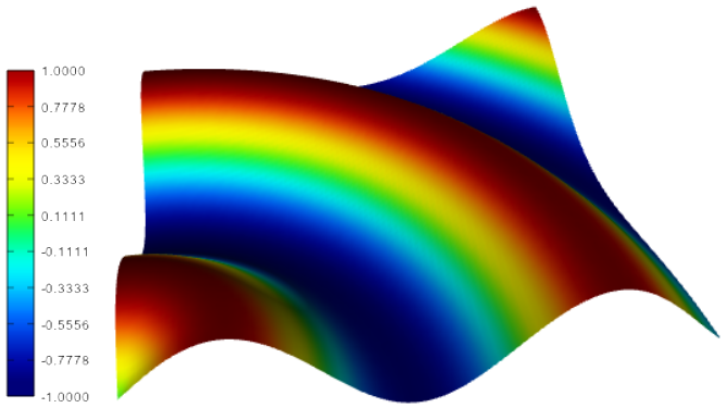


Fig. 31: Plotting functions of two variables with WebGL.

5 More on Variables

5.1 Objectives

- Review and refine your knowledge of variables.
- Understand dynamic type interpretation.
- Learn about *local* and *global* variables.
- Understand why global variables should be avoided.
- Learn about *shadowing* of variables.

The reason why we use variables is the same in Karel the Robot, Python, and all other programming languages – to store useful information for later use. Also the way we work with variables is almost the same, so we will just mention a few aspects that are new in Python.

5.2 Assigning values to variables

In Python, several variables can be initialized at the same time:

```
x = y = z = 0.0
print "x =", x
print "y =", y
print "z =", z
```

Output:

```
x = 0.0
y = 0.0
z = 0.0
```

As in Karel, variables must be defined (have a value assigned) before they can be used.

5.3 Dynamic type interpretation

Python is very generous with the types of variables – it will figure out the type of a variable at runtime, depending on the value that we store in it. The type can even change at runtime. For example, let us look at the following code:

```
a = 5
print a
a = "Hi!"
print a
a = True
print a
```

The output is

```
5
Hi!
True
```

This is very different from compiled languages such as C, C++ where one needs to declare the type of every variable before its first use, and the type then cannot be changed.

5.4 Changing values of variables

The simplest way to increase the value of a numerical variable in Python by a given number is to use the '+=' command:

```
v = 1
v += 3
print "v =", v
```

Output:

```
v = 4
```

We can also subtract a number from a numerical variable:

```
v -= 1
print "New value of v is", v
```

Output:

```
New value of v is 3
```

We can multiply a numerical variable with a number:

```
v *= 4
print "Now v is", v
```

Output:

```
Now v is 12
```

And finally, we can divide a numerical variable with a number:

```
v /= 6
print "Finally, v is", v
```

Output:

```
Finally, v is 2
```

It is possible to use existing variables to assign a value to a new variable:

```
a = 1
b = 2.5
c = 0.5
d = (a + b) / c
print "d =", d
```

Output:

```
d = 7.0
```

Of course, apples cannot be mixed with oranges. When we try to add a number to a text string,

```
a = "My car is a Ferrari."
b = 3.5

c = a + b
```

then the interpreter rightfully complains:

```
Traceback (most recent call last):
  File "<nclab>", line 4, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

5.5 Local and global variables

As we know from Part I, Subsection 11.9, a variable defined inside a function is *local* to that function. If we try to access the variable outside the function, even though the function was already called, the variable is unknown. Look at the following code:

```
def add(a, b):
    c = a + b
    return c

print "Sum of 3 and 4 is", add(3, 4)
print c
```

The output is:

```
Sum of 3 and 4 is 7
Traceback (most recent call last):
  File "<nclab>", line 6, in <module>
NameError: name 'c' is not defined
```

In computer programming, *we want all our variables to be local*, because this avoids name clashes and minimizes the risk of unwanted side effects. Mistakes such as unwanted overwriting of the value stored in a global variable belong to the most difficult ones to find.

5.6 Sure method to spoil your code

Once a variable is used in the main program, then it is *global*, meaning that you could use it in any function that you define. Like in the following code:

```
val = 5.0

def msg():
    print "val =", val

msg()
```

Brrr, this code is ugly! But it works (unfortunately), see the output:

```
val = 5.0
```

Remember:

> *Passing global variables to functions is a bad habit that makes your code difficult to understand, prone to mistakes, and extremely difficult to fix.*

A function should always obtain all the data it needs through its arguments. Imagine that a unit of soldiers would act based not only on orders it receives (that's the argument list), but also based on rumors that some of the soldiers read in a Sunday magazine. That might not work well!

A clean version of the above code is

```
def msg(v):
    print "val =", v

val = 5.0

msg(val)
```

Can you see the difference? Now the function `msg` is *readable* – you do not need to look elsewhere in the program to understand what it does. The output is:

```
val = 5.0
```

5.7 Variable shadowing

Sometimes, by accident or intentionally, we may have two different variables of the same name in the code. Look, for example, at this short program:

```
c = 5

def add(a, b):
    c = a + b
    print "c =", c
    return c

add(1, 2)
print "c =", c
```

The output is

```
c = 3
c = 5
```

There is no problem with the last line of the program because the local variable `c`, defined inside of the function `add`, is no longer defined there. But inside of the function `add`, both the global variable `c` and the local variable `c` are defined at the same time. This is commonly referred to as *variable shadowing*, and we need to remember that if this happens, **the local variable is used**.

6 Review of Logic

6.1 Objectives

– Review Boolean expressions, operations, and variables.

Logic is always a useful review topic, so let's return to it briefly although logic in Python is virtually the same as in Karel the Robot.

6.2 Boolean expressions

Let's begin with creating a variable and printing it:

```
a = 1
print a
```

Output:

```
1
```

Now type

```
print a > 0
```

Output:

```
True
```

Although the syntax looks a bit unusual, this is a correct Python code – since `a` is one, the Boolean expression `a > 0` is `True`. This is the value that gets printed. We can try it the other way round:

```
print a < 0
```

Output:

```
False
```

6.3 Boolean variables

We can also store the value of the expression `a > 0` in a Boolean variable, and print it:

```
b = a < 0
print b
```

Output:

```
False
```

Boolean expressions involving numbers often contain the following operators:

Symbol	Meaning
>	greater than
>=	greater than or equal to
<=	less than or equal to
<	less than
==	equal to
!=	not equal to
<>	not equal to (same as !=)

6.4 Boolean operations

With Boolean variables we can do logical operations such as `and`:

```
a = 1
v1 = a > 0
v2 = a < 5
print v1 and v2
```

Output:

```
True
```

Logical `or`:

```
v3 = a > 10
print v1 or v3
```

Output:

```
True
```

Negation:

```
v4 = not v1
print v4
```

Output:

```
False
```

Sometimes logical expressions can be more complicated, but this is no problem as we always can use round brackets:

```
v5 = not ((v1 and v2) or (v3 and not v4))
```

7 More on Conditions and the Conditional Loop

7.1 Objectives

- Review your knowledge of conditions and learn about the `elif` statement.
- Learn to generate random numbers in Python.
- Learn to use the `break` statement to terminate the loop.

– Learn to use the `continue` statement to skip the rest of the loop's body.
– Use the `while` loop to make your first trip into scientific computing.

Conditions in Python are almost the same as in Karel the Robot. There is one practical new feature though – the `elif` statement that simplifies cases with more than two options. If you think that "elif" sounds a lot like "else if" then you are absolutely right! Typing

```
if logical_expression_1:
    do_action_1
elif logical_expression_2:
    do_action_2
else:
    do_action_3
```

means exactly the same as

```
if logical_expression_1:
    do_action_1
else:
    if logical_expression_2:
        do_action_2
    else:
        do_action_3
```

Clearly, the latter involves more indentation. Let us now show a concrete example:

Imagine that we have five boxes for apples: Box A is for apples that weight under 5 ounzes, box B for apples that weight at least five but less than 10 ounzes, box C for apples that weight at least 10 but less than 15 ounzes, etc. Apples weighting 20 ounzes or more will all be put into box E. Here is a function that, given a weight of an apple, chooses the correct box for it:

```
def box(weight):
    if weight < 5:
        return "A"
    elif weight < 10:
        return "B"
    elif weight < 15:
        return "C"
    elif weight < 20:
        return "D"
    else:
        return "E"
```

Note the mandatory colon ':' after every `if`, `elif` and `else` statement. Also the `while` loop in Python has a mandatory colon, this is virtually the only (and cosmetic) difference between Python and Karel.

To make sure that we understand the advantage of the `elif` statement, let us rewrite the function `box(weight)` without it:

```
def box(weight):
    if weight < 5:
        return "A"
    else:
        if weight < 10:
            return "B"
        else:
            if weight < 15:
                return "C"
            else:
                if weight < 20:
                    return "D"
                else:
                    return "E"
```

There is another, even less elegant way:

```
def box(weight):
    if weight < 5:
        return "A"
    if weight >= 5 and weight < 10:
        return "B"
    if weight >= 10 and weight < 15:
        return "C"
    if weight >= 15 and weight < 20:
        return "D"
    if weight >= 20:
        return "E"
```

So we can see that the `elif` command is a useful one!

7.2 Generating random numbers in Python

Same in Karel, Python, and all other programming languages – the `while` loop is a conditional loop that should be used whenever the number of repetitions is not known a-priori. Let us

illustrate this using random numbers – what can be more uncertain than that? Random numbers between zero and one can be generated in Python as follows:

```
from random import *
a = random()
print "This is a random number between 0 and 1:", a
```

Output:

```
This is a random number between 0 and 1: 0.999415184986
```

As a simple exercise, let us generate and print random numbers between zero and one as long as they are less than 0.75. We certainly do not know how many repetitions will be done!

```
from random import *
a = random()
while a < 0.75:
    print a
    a = random()
```

Output:

```
0.383771890647
0.068637471541
0.0850385942776
0.682442836322
0.168282761575
0.121694025066
0.185099163429
0.606740825997
0.209771333501
0.186191737708
0.608456569449
0.591009262698
```

Running the program once more gives another output:

```
0.637423013735
0.533383497321
0.240830918488
0.343139647492
0.718014397238
0.698931548709
0.363558456088
```

7.3 The `break` statement

The last example can be used to introduce the `break` statement. This is a useful way to exit a loop at any time, no questions asked. If multiple loops are embedded, then the `break` statement only breaks the closest one. We can rewrite the above program as follows:

```
from random import *
while True:
    a = random()
    if a >= 0.75:
        break
    print a
```

Output:

```
0.486690683395
0.232518153456
0.38084453626
```

Combining `while True:` with the `break` statement in this way is quite popular among Python programmers. The technique should not be overused though since it is not a perfect example of structured programming. It is just a shortcut whose purpose is to save one additional `if - else` statement inside the loop's body. Nevertheless, it is good to know that the `break` statement exists and one day you will find use for it.

7.4 The `continue` statement

Sometimes it is practical to skip the rest of the loop's body and start over. Python has the `continue` command for this. To illustrate its use, let us write a program that will generate triplets of random numbers between zero and one until all three of them are greater than 0.9. We also want to know how many attempts were needed.

Clearly, after the first number is generated and it is less or equal to 0.9, it does not make any sense to generate the other two. That's when we use the `continue` command. If the first number is greater than 0.9 but the second one is less or equal to 0.9, it is time to use the `continue` command once again. Here is the code:

```
from random import *
counter = 0
while True:
    counter += 1
    a1 = random()
```

```
    if a1 <= 0.9:
        continue
    a2 = random()
    if a2 <= 0.9:
        continue
    a3 = random()
    if a3 <= 0.9:
        continue
    print "Finally the desired triplet:"
    print a1, a2, a3
    print "It took", counter, "attempts to get it!"
    break
```

Output:

```
Finally the desired triplet:
0.921077479893 0.956493808495 0.917136354634
It took 1168 attempts to get it!
```

Running the program once more:

```
Finally the desired triplet:
0.922450872182 0.964451255861 0.969381004134
It took 2757 attempts to get it!
```

7.5 A trip into scientific computing

We have used the `while` loop in Karel a lot, but Karel usually did at most few dozens of cycles. In reality, computers can easily do millions or billions of cycles or more. In computing, we call them *iterations*. Actually, our programming skills are already good enough to tackle an entry-level scientific computing problem, so let's give it a try!

We are interested in finding an angle x that is equal to its own cosine. Let's our Python plotting skills shine and write a code to plot the graphs of the functions $\cos(x)$ and x:

```
from numpy import pi, cos
from pylab import *
x = linspace(0, pi/2, 100)
y = cos(x)
z = x
clf()
```

```
plot(x, y, label="cos(x)")
plot(x, z, label="x")
legend()
lab.show()
```

The output is shown in Fig. 32.

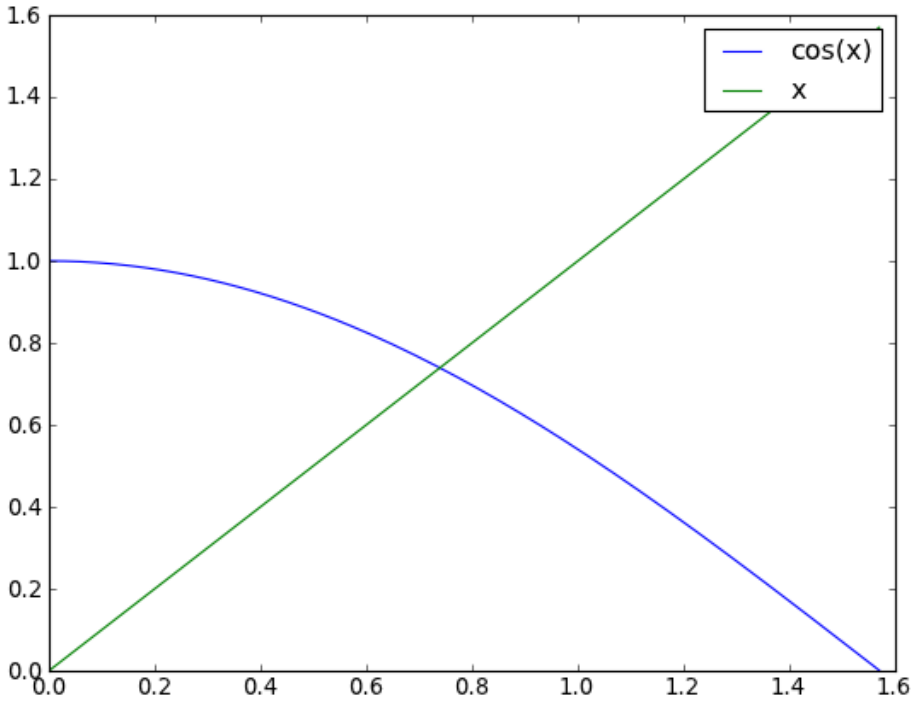


Fig. 32: Graphs of the functions $\cos(x)$ and x in the interval $(0, \pi/2)$.

Looking at the graphs, clearly the solution is somewhere between 0.7 and 0.8. An engineer would not be extremely happy with such an answer though, so we need to make this guess more accurate. Looking at the graphs again, we can see that in order to reach the intersection point, we can depart from zero and make very small steps to the right while the graph of $\cos(x)$ is above the graph of x. Let's do this, and stop at the first point where $\cos(x)$ will be less than x. Our step will be called `dx`, and our moving point on the x-axis will be called simply `x`. We will also count steps via the variable `n`. Choosing `dx = 1e-6` will give us the result accurate to five decimal digits:

```
from numpy import cos
x = 0
```

```
dx = 1e-6
n = 0
while cos(x) > x:
    x += dx
    n += 1
print "Result is approximately", x
print "Steps made:", n
```

Output:

```
Result is approximately 0.739086
Steps made: 739086
```

In other words, the `while` loop run 739086 times. Can you imagine doing this on your pocket calculator?

We should remark that the numerical method that we used was rather naive and in fact it took quite a bit of CPU time. Scientific computing is an exciting field where researchers develop new methods that can solve problems like this more accurately and using less CPU time. In particular, for our problem there are much more powerful methods that can do the same or better job using just five or six steps (!). One of them is the *Newton's method*. If you like what we did here, we encourage you to explore more math functionality and in particular numerical methods in NCLab. There are displayed projects for the Newton's method and other methods to solve nonlinear equations and nonlinear equation systems.

8 Strings

8.1 Objectives

- Learn basic operations with single and multiline strings.
- Learn to use quotes and backslashes, and to concatenate and repeat strings.
- Learn to refer to letters by indices, parse strings, and slice them.

By a *string* we mean a text surrounded by double or single quotes, such as `"this is a string"` or `'this is a string as well'`. Strings are useful to make outputs more informative, but they have many other uses as well. For example, they can represent data in databases such as in a phone book. So we need to understand them well, as well as various operations that we can do with them.

8.2 Using quotes

Let us first understand how to use quotes in strings. The safest way is to use them with a backslash:

```
print "I said \"yes\"."
```

Output:

```
'I said "yes".'
```

Another example:

```
print "It doesn\'t matter."
```

Output:

```
"It doesn't matter."
```

8.3 Multiline strings and backslashes

If we want to use multiline strings, the best way is to enclose them in triple quotes:

```
edgar = """
Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
''Tis some visitor,' I muttered, 'tapping at my chamber door -
Only this, and nothing more.'
"""
print edgar
```

Output:

```
Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
''Tis some visitor,' I muttered, 'tapping at my chamber door -
Only this, and nothing more.'
```

Yes, there is one empty line on top and one on bottom. The reason is that Python inserted newline symbols there. If we want to avoid these empty lines, we have to include a backslash after the first triple quote, and also at the end of the last line before the closing triple quote. The backslash prevents Python from inserting a newline symbol into the string. Let us show one more example on backslashes. The string

```
"""\
This is the first line,
and this is the second line.\
"""
```

will render as

```
'This is the first line,\nand this is the second line.'
```

When a backslash is included at the end of the first line of text,

```
"""\
This is the first line,\
and this is the second line.\
"""
```

we obtain

```
'This is the first line,and this is the second line.'
```

Yes we need to watch for empty characters! Inserting one before the second backslash,

```
"""\
This is the first line, \
and this is the second line.\
"""
```

we finally obtain

```
'This is the first line, and this is the second line.'
```

8.4 Concatenation and repetition

Strings can be concatenated (glued together) with the '+' operator, and repeated with '*':

```
word = 'Help' + 'me!'
print "I yelled" + 3 * word
```

Output:

```
I yelledHelpme!Helpme!Helpme!
```

You can see that empty spaces matter. Let's try again:

```
word = '\"Help' + ' me!\" '
print "I yelled " + 3 * word
```

Output:

```
I yelled "Help me!" "Help me!" "Help me!"
```

8.5 Referring to letters by their indices

Individual letters forming a string can be accessed via indices. The indices start from zero. It is also handy to use the index `-1` for the last index, `-2` for the one-before-last etc.

```
word = "breakfast"
print "First character:", word[0]
print "Second character:", word[1]
print "Last character:", word[-1]
print "One-before-last character:", word[-2]
```

Output:

```
First character: b
Second character: r
Last character: t
One-before-last character: s
```

8.6 Parsing strings with the `for` loop

The length of a string is obtained using the function `len()`. Example:

```
word = "breakfast"
n = len(word)
print "Length of the string:", n
```

Output:

```
Length of the string: 9
```

In order to parse strings, it is practical to introduce the Python counting (`for`) loop. We do this ahead of time, more on the `for` loop will be said in Section 10. Let us begin with an example again: The following code parses the string `word` letter by letter and prints all the letters:

```
word = "breakfast"
n = len(word)
for i in range(n):
    print word[i]
```

Output:

```
b
r
e
a
k
f
a
s
t
```

This deserves some explanation. Calling the function `range(n)` generates a sequence of integers that goes from zero to `n-1`. Let's try this:

```
print range(3)
```

Output:

```
[0, 1, 2]
```

Another example:

```
print range(10)
```

Output:

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

When called with two arguments, `range(n1, n2)` will generate a sequence `[n1, n1+1, n1+2, ..., n2-1]`. Example:

```
print range(4, 10)
```

Output:

```
[4, 5, 6, 7, 8, 9]
```

The `for` loop goes through the sequence generated by the `range()` function, taking one element per cycle. So in the first cycle, the variable `i` has the value `0`, in the second cycle `i` is `1`, in the third cycle `i` is `2` etc., until the end of the sequence is reached.

Since we know from Subsection 8.5 how to index letters in a string via their indices, now the above code that decomposed the word `breakfast` into individual letters makes perfect sense!

8.7 Slicing strings

We can also access substrings, this is called *slicing*:

```
w1 = "bicycle"
w2 = w1[0:3]
print w2
```

Output:

```
bic
```

An omitted first index in a slice defaults to zero:

```
w3 = w1[0:2]
print w3
```

Output:

```
bi
```

An omitted second index defaults to the length of the string:

```
w4 = w1[2:]
print w4
```

Output:

```
cycle
```

This is enough for now but hold on – many wonderful operations one can do with text strings will be shown in Subsection 12.7

9 Tuples, Lists, and Dictionaries

9.1 Objectives

- Learn to store data in tuples, lists, and dictionaries.
- Learn the differences between these data types, and their best use cases.
- Learn useful built-in functions to operate with these data structures efficiently.

Tuples, lists, and dictionaries belong to the most important data structures in Python, and the language offers a wide range of functionality to work with them efficiently. They become especially useful in combination with the `for` loop that was already mentioned in Subsection 8.6, and that will be discussed in more detail in Section 10.

9.2 Tuples

Let us begin with tuples. Tuples are perfect for sequences of data that does not change in time - such as the names of weekdays, or names of months. To define a tuple, enclose comma-separated items in round brackets:

```
months = ('January', 'February', 'March', 'April', 'May', 'June',\
'July', 'August', 'September', 'October', 'November', 'December')
```

The items in the tuple `months` are all strings, but they do not have to be. A tuple can be very heterogeneous – it can contain strings, numbers, other tuples, etc. Although – in simplicity is beauty, we do not have to always take advantage of the extreme possibilities. It is important to remember that **a tuple, once defined, cannot be changed** in the sense that no new items can be added and existing items cannot be deleted.

Items in a tuple are referenced by indices, in the same way as individual characters are referenced in a text string. Remember that indices always start from zero:

```
print "First month:", months[0]
print "Second month:", months[1]
print "Third month:", months[2]
```

Output:

```
First month: January
Second month: February
Third month: March
```

Index `-1` refers to the last item, `-2` to the one before last, etc.:

```
print "One before last month:", months[-2]
print "Last month:", months[-1]
```

Output:

```
One before last month: November
Last month: December
```

We can *slice* tuples similarly to how we sliced strings:

```
months[2:5]
```

Output:

```
('March', 'April', 'May')
```

The length of a tuple is obtained using the function `len()`:

```
len(months)
```

Output:

```
12
```

9.3 Lists

We first encountered lists in Karel (Part I, Section 11). What we learned there will hold true in Python, but Python lists have many additional useful functions that we will learn here. Let us start by reviewing the basics.

Lists are similar to tuples, and all indexing and slicing works in the same way. Empty list `L` is defined as follows:

```
L = []
```

The biggest difference between a tuple and a list is that **items in a list can be changed**. So, we use lists for items that can change in time. Let's say that there are 10 people in a school class `cls`:

```
cls = ['John', 'Pam', 'Emily', 'Jessie', 'Brian', \
'Sam', 'Jim', 'Tom', 'Jerry', 'Alex']
```

Then however, Emily moves to a different city:

```
del cls[2]
print cls
```

Output

```
['John', 'Pam', 'Jessie', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex']
```

After some time, a new student Jack moves in:

```
cls.append('Jack')
print cls
```

Output

```
['John', 'Pam', 'Jessie', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex', 'Jack']
```

Useful can be the function `pop()` that deletes an item and returns it for further use (as opposed to `del` which just deletes the item):

```
name = cls.pop(2)
print name
print cls
```

Output

```
Jessie
['John', 'Pam', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex', 'Jack']
```

New item can be inserted at an arbitrary position using the function `insert()`:

```
cls.insert(3, 'Daniel')
print cls
```

Output:

```
['John', 'Pam', 'Brian', 'Daniel', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex', 'Jack']
```

A list can be sorted via the function `sort()`:

```
cls.sort()
print cls
```

Output:

```
['Alex', 'Brian', 'Daniel', 'Jack',
'Jerry', 'Jim', 'John', 'Pam', 'Sam', 'Tom']
```

The function `reverse()` reverses a list:

```
cls.reverse()
print cls
```

Output:

```
['Tom', 'Sam', 'Pam', 'John', 'Jim',
'Jerry', 'Jack', 'Daniel', 'Brian', 'Alex']
```

The function `count()` counts the number of occurences of an item in the list:

```
cls.count('Jerry')
```

Output:

```
1
```

The function `index()` returns the index of the first occurence of an item. If the item is not found, error is thrown:

```
cls.index('Jerry')
```

Output:

```
5
```

Another useful operation with lists is their *zipping*. The `zip` command takes two lists as arguments and creates a new list of pairs. Example:

```
A = [1, 2, 3]
B = ['x', 'y', 'z']
print zip(A, B)
```

Output:

```
[(1, 'x'), (2, 'y'), (3, 'z')]
```

If the lists are not equally-long, the superfluous items in the longer list are skipped:

```
A = [1, 2, 3, 4, 5]
B = ['x', 'y', 'z']
print zip(A, B)
```

Output:

```
[(1, 'x'), (2, 'y'), (3, 'z')]
```

9.4 Dictionaries

Sometimes we need to store additional information for people (or general items) in a list. Such information may be their phone number, age, weight, address, family status and/or many other things. In this context, the people's names are said to be *keys* and the additional information are the corresponding *values*. Of course this is not only about people, we can have a list of cars and the additional information to each car may be its price, mileage, or gas consumption. And we should not forget to mention a real dictionary, where the keys can be English words and the values the corresponding Spanish translations.

The data structure containing keys and values is called a *dictionary*. An empty dictionary `D` is defined as follows:

```
D = {}
```

An example of a phone book would be:

```
phonebook = {'Peter Parson': 8806336, 'Emily Everett': 6784346, \
'Lewis Lame': 1122345}
```

Note the curly brackets. Once we have created a new phone book, we may want to add a new person to it:

```
phonebook['Silly Sam'] = 1234567
print phonebook
```

Output:

```
{'Peter Parson': 8806336, 'Emily Everett': 6784346,
'Lewis Lame': 1122345, 'Silly Sam': 1234567}
```

And we can also delete a person from the phonebook:

```
del phonebook['Peter Parson']
print phonebook
```

Output:

```
{'Emily Everett': 6784346, 'Lewis Lame': 1122345,
'Silly Sam': 1234567}
```

We can check if a key is in the dictionary:

```
if phonebook.has_key('Silly Sam'):
    print "Silly Sam's phone number is", phonebook['Silly Sam']
else:
    print "Silly Sam is not in the phonebook."
```

Output:

```
Silly Sam's phone number is 1234567
```

To get the list of all keys, we type:

```
phonebook.keys()
```

Output:

```
['Emily Everett', 'Lewis Lame', 'Silly Sam']
```

Similarly, we can also get the list of all values:

```
phonebook.values()
```

Output:

```
[6784346, 1122345, 1234567]
```

It is important to remember that **dictionaries are not ordered** in any special way. While a list can be sorted, a dictionary can not. They are only designed to get you the value for a given key. The length of a dictionary can be obtained using the function `len()`:

```
len(phonebook)
```

Output:

```
3
```

There are additional functions that one can use with dictionaries. However, what we mentioned so far is enough for an introductory course and we refer to the Python tutorial[1] for more details.

10 Counting Loop

10.1 Objectives

- Review the `range()` function and understand that it creates a Python list.
- Understand when the Python `for` loop should be used.
- Learn that the `for` loop can go over any list or tuple.

Before we bring up the `for` loop, let us revisit the `range()` function that was first mentioned in Section 8. After learning about tuples, lists and dictionaries in Section 9, the reader will not be surprized when we state that this function produces a Python list of integer numbers. For example:

```
range(10)
```

produces the following output:

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

It is possible to start from a different number:

```
range(3, 10)
```

Output:

```
[3, 4, 5, 6, 7, 8, 9]
```

The `for` loop in Python is a *counting loop* analogous to the `repeat` loop in Karel the Robot. Thus it should be used when we know how many repetitions there will be. However, in Python, the counting loop is not restricted to integer numbers only. It is much more general – *it goes over the items of aa arbitrary list or tuple*.

[1] http://docs.python.org/tutorial

10.2 Combined with the `range()` function

When used in combination with the `range()` function, it goes over integer indices. Example:

```
for i in range(10):
    print i
```

Output:

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

As usual, note indentation of the body of the loop, which has the same purpose as indentation in the `while` loop.

10.3 Combined with a general list or tuple

Remember our tuple of months? We can use the `for` loop to print them as well:

```
for m in months:
    print m
```

Output:

```
January
February
March
April
May
June
July
August
September
October
November
December
```

The `for` loop works in the same way with lists – it does not distinguish between these two data types at all.

10.4 Reverting a dictionary

Remember that it is easy to extract keys and values from a dictionary using the `keys()` and `values()` functions? In either case, the result of this operation is a *Python list*, which in turn can be used with the `for` loop. That was easy. Let us do something more interesting – perhaps convert an English-Spanish dictionary into a Spanish-English one?

```
ES = {'city': 'ciudad', 'fire': 'fuego', 'sun': 'sol'}
SE = {}
for english_word in D:
    spanish_word = D[english_word]
    SE[spanish_word] = english_word

print "Spanish-English dictionary:"
print SE
```

The output of this program is:

```
Spanish-English dictionary:
{'ciudad': 'city', 'sol': 'sun', 'fuego': 'fire'}
```

Isn't Python beautifully simple?

11 Exceptions

11.1 Objectives

- Understand what exceptions are, when they can occur, and how to catch them.
- Learn to never underestimate the user.
- Understand `ZeroDivisionError` and other exceptions.
- Learn to use the `assert()` function and catch the `AssertionError` exception.
- Learn how to raise exceptions using the `raise` statement.

Imagine that you have to write a Python function to evaluate

$$f(x) = \frac{1}{9 - x^2}.$$

Of course, this is simple!

```
def f(x):
    return 1. / (9 - x**2)
```

But we can almost guarantee that the first user will call the function with $x = 3$ or $x = -3$. What does he see?

```
Traceback (most recent call last):
  File "<nclab>", line 1, in <module>
ZeroDivisionError: float division by zero
```

11.2 Never underestimate the user

We need to be very careful with users! If we are not, they always manage to crash our code. By always we mean *always*. And most often they do even if we are as careful as possible. Sometimes they do it for fun, to show us that we are not all that smart. But most of the time they do it without bad intentions – they are just learning to use our program, not knowing what they are not supposed to do, stepping over boundaries that we would never cross. For example, the author of the function

```
from numpy import sqrt
def invsqrt(x):
    return 1./sqrt(x)
```

would hardly call it with `x = 0`, or even `x = -1`. *The function was not designed for that!* But the user did not know it.

Here is an alternative version of the above function `f(x)` that will not crash:

```
def f(x):
    if x != 3 and x != -3:
        return 1. / (9 - x**2)
    else:
        print "Division by zero detected, returning 0."
        return 0
```

However, it is useful to slow down here and look at what we have. The fact that the warning sentence is printed is not very useful, because it is printed as output for the human eye only. What if there is nobody to read it? Even if division by zero happens, the function returns zero (a valid number) and the program continues on. Not good. Fortunately, Python (and other languages) have an elegant way to take care of exceptional situations like this. This is called *exceptions handling*.

11.3 `ZeroDivisionError` Exception

In contrast to *predicting* what possibly could go wrong and including conditions to detect wrong input, we do not check the input and let it simply happen. Does it sound careless? Let us return to the original simple form of the function `f(x)`:

```
def f(x):
    return 1. / (9 - x**2)
```

This is the function that we will use. However, we will call it in such a way that an exceptional situation (here division by zero) is captured. For this we use the `try` command:

```
try:
    result = f(a)
except ZeroDivisionError:
    print "Division by zero detected, performing emergency scenario!"
else:
    print "Evaluation of f(a) was successful, going on normally."
```

Before continuing, we recommend that you type this code into the code cell, including the definition of the function `f(x)`, and run it with values that crash the function `f(x)` and with values that do not. If you are a newcomer to exceptions, this is an interesting experience. Feels a bit like letting the airplane crash, but save all passengers afterwards.

11.4 The `assert()` function

The `assert()` function raises an exception whenever the Boolean statement used as its argument is `False`. This is a powerful tool to raise our own exceptions. These exceptions can be caught using the `try` command as all other exceptions:

```
def f(x):
    assert(x != 3 and x != -3)
    return 1. / (9 - x**2)

try:
    result = f(a)
except AssertionError:
    print "Assertion failed, performing emergency scenario!"
else:
    print "Assertion passed, going on normally."
```

When called with `a` equal to `1`, this code gives:

```
Assertion passed, going on normally.
```

When called with `a` equal to `3`, this is what happens:

```
Assertion failed, performing emergency scenario!
```

11.5 The `raise` statement

Python does not have a command to stop program execution. Sometimes this is need though, such as when the user calls our function with incorrect data. To illustrate this, let us return to the function `invsqrt()` that was defined above, but now we include a `raise` command in it:

```
from numpy import sqrt
def invsqrt(x):
    if x <= 0:
      raise ValueError("Please do not call the invsqrt() \
function when x is zero or negative.")
    return 1./sqrt(x)
```

Now the output of `invsqrt(-5)` is:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "<nclab>", line 4, in invsqrt
ValueError: Please do not call the invsqrt()
function when x is zero or negative.
```

Incorrect values of user input are the most common cause for raising exceptions on our own. Any other exception type can be raised as well if needed.

11.6 Other types of exceptions

There are many different exceptions including `MemoryError` (program runs out of memory), `NameError` (local or global name is not found), `OverflowError` (result of an arithmetic operation is too large to be represented), `SyntaxError` (parser encounters a syntax error), `IndentationError` (parser finds incorrect indentation), `UnboundLocalError` (reference is made to a local variable in a function or method, but no value has been bound to that variable), and others. All of them can be caught using the `try` command. If you are interested in learning more, visit http://docs.python.org/library/exceptions.html.

12 Object-Oriented Programming

12.1 Objectives

- Understand the philosophy of object-oriented (OO) programming.
- Understand the difference between OO programming and procedural programming.
- Learn basic concepts – classes, objects, and methods.
- Learn the syntax of defining and instantiating classes in Python.

Since the very beginning of this course, we have put emphasis on designing simple algorithms and writing clean, transparent code. This included:

- Creating custom functions for functionality that can be reused.
- Keeping all variables as local as possible

Object-oriented programming brings these good programming practices to perfection – it will help us to write crystal clear, elegant, and very safe code.

12.2 From procedural to object-oriented programming

The programming style we have been using so far is called *procedural programming*. The name, of course, comes from using *procedures* to solve various tasks that lead to the solution of the problem at hand. Procedure is an activity, a sequence of operations done with data that is supplied to it. *Procedures do not own the data they operate with.*

This, however, means that the data must be stored somewhere else. In other words, the range of validity of the data extends beyong the borders of the function where it is processed. Can you see an analogy to using local and global variables? It is great to keep variables local, and in the same way it would be great to make data local to procedures that operate with them. But wait, this is called *object-oriented programming!*

12.3 Example: Moving to a different city

Here is a real-life example demonstrating the difference between procedural and object-oriented programming. Imagine that you are moving and all your things need to be packed, boxed, loaded on a truck, hauled a 1000 miles, unloaded, and brought into your new home.

Procedural approach: You (the procedure) go rent a truck – this is the data that the procedure does not own. Then you pack everything, load it on the truck, drive to the other city, unload your things, and carry them into your new home. Then you go return the truck (that's the return statement at the end of the procedure).

Object-oriented approach: You call a moving company "C". This means that you just created an object C. Note that the object C owns the truck. They will come to your home and use methods C.pack(), C.box(), C.load(), C.haul(), C.unload(), C.carry() and perhaps even C.unpack() to solve your problem. Notice the following benefits of the object-oriented approach:

– The work is being done by the company, you do not have to worry (delegation of work).
– You do not have to be able to drive a truck (better code structure).
– You do not need to know how the company does the moving (reduced flow of data).
– You can concentrate on solving problems in your area of expertise (greater efficiency).
– You do not risk crashing the truck (safer programming).

12.4 Classes, objects, and methods

Let us stay with the moving example for a little while longer. The moving company that you heard of, or that you saw in an advertisement, is a *class* or in other words a *concept*. Formally, we say that *class* is an entity comprising data and functionality that uses the data. But a concept will not get your things moved. For that you need a concrete team of movers. This is called an *object* or an *instance of the class*.

When the movers appear at your doorstep, that's when an instance of the class is created. When they pack, box, and load your things, the object is using *methods* of the class. Technically, methods belong to the class because that's where they are defined, but often we say that "an objects uses its methods to ...". Once again, methods are *defined in a class* but *used by instances of the class*. Methods of a class can operate on data owned by the class (the truck) as well as on data that does not belong to the class (your things that are being moved.

Similarly with variables – variables are defined in a class, but they do not have concrete values there. In the class definition, they are part of a concept. Only after an instance of the class is created, the variables are filled with concrete values.

12.5 An example from geometry

Let's see how all this is implemented in Python. We will say goodbye to the movers and play with some simple geometrical concepts instead. Let us create a class `circle`. The data owned by this class will be:

– Radius `R`,
– center with coordinates `Cx, Cy`,
– two arrays `ptsx`, `ptsy` to plot the circle.

The methods of this class will include

– `__init__(self, r, cx, cy)` ... constructor,
– `area(self)` ... calculate and return its area,
– `perimeter(self)` ... calculate and return its perimeter,
– `draw(self)` ... draw itself.

Do not be disturbed by the funny way the constructor `__init__` is defined in Python, or by the argument `self` that is present in all methods. Constructors are used in all object-oriented languages to pass external parameters (such as the radius and the center point in this case) into newly created instances. In other words, each instance of the class `circle` can have a different radius and a different center.

The keyword `self` must be included in each method of the class and this is a reference to the concrete instance through which all data and methods of the instance are accessed. Such a reference is present in other object-oriented languages as well - including C++ where the programmer does not have to mention it explicitly like in Python. It is added there by the compiler.

12.6 Defining and using class `circle`

This is how the class `circle` is defined in Python:

```
from numpy import pi, cos, sin
from pylab import clf, plot, legend, axis

class circle:
    # Constructor:
    def __init__(self, r, cx, cy, n = 100):
        self.R = r
        self.Cx = cx
        self.Cy = cy
        self.n = n
        # And now the points for plotting:
        self.ptsx = []
        self.ptsy = []
        da = 2*pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * cos(i * da))
            self.ptsy.append(self.Cy + self.R * sin(i * da))
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)

    # Method to return circle area:
    def area(self):
        return pi * self.R**2

    # Method to return perimeter:
    def perimeter(self):
```

```
        return 2 * pi * self.R

    # Method to plot the circle:
    def draw(self, label):
        axis('equal')
        plot(self.ptsx, self.ptsy, label = label)
        legend()
```

Notice how the variables owned by the class are always used with the prefix `self`.

With such a nice class in hand, it is easy to draw various circles, and calculate their areas and perimeters:

```
# Create an instance:
C1 = circle(1, 0, 0)
print "Are and perimeter of circle 1:", \
C1.area(), C1.perimeter()
C2 = circle(0.5, 1, 0)
print "Are and perimeter of circle 2:", \
C2.area(), C2.perimeter()
C3 = circle(0.25, 1.5, 0)
print "Are and perimeter of circle 3:", \
C3.area(), C3.perimeter()
clf()
C1.draw("First circle")
C2.draw("Second circle")
C3.draw("Third circle")
lab.show()
```

The output (text and Fig. 33) is shown below.

```
Area and perimeter of circle 1: 3.14159265359 6.28318530718
Area and perimeter of circle 2: 0.785398163397 3.14159265359
Area and perimeter of circle 3: 0.196349540849 1.57079632679
```

12.7 Text string as a class

We first met text strings in Section 8. Now that we know how classes are defined and used, we can reveal that text string is a class! In fact it has many great methods that really make working with texts easy and fun. Let us show some of them. Below, arguments enclosed in brackets are optional, and if they are used, then without the brackets:

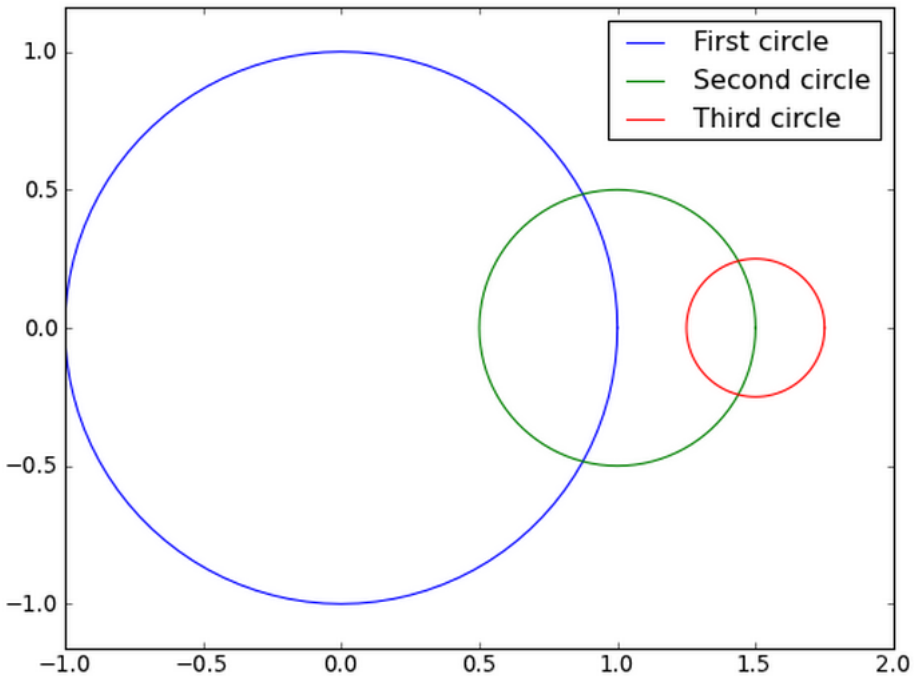


Fig. 33: Three instances of the class `circle`.

Method `capitalize()`

Return a copy of the string with only its first character capitalized.

Example:

```
str = "sentences should start with a capital letter."
print str.capitalize()
```

Output:

```
Sentences should start with a capital letter.
```

Method `count(sub[, start[, end]])`

Return the number of occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Example:

```
str = "John Stevenson, John Lennon, and John Wayne."
print str.count("John")
```

Output:

```
3
```

Method `find(sub[, start[, end]])`

Return the lowest index in the string where substring `sub` is found, such that sub is contained in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Return `-1` if `sub` is not found.

Example:

```
str = "This happened during the summer."
print str.find("sum")
```

Output:

```
25
```

Method `index(sub[, start[, end]])`

Like find(), but raise ValueError when the substring is not found.

Method `isalnum()`

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

Example:

```
str1 = "mypassword123"
if str1.isalnum():
    print str1, "is alphanumeric."
else:
    print str1, "is not alphanumeric."
str2 = "mypassword123+"
if str2.isalnum():
    print str2, "is alphanumeric."
else:
    print str2, "is not alphanumeric."
```

Output:

```
mypassword123 is alphanumeric.
mypassword123+ is not alphanumeric.
```

Method `isalpha()`

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

Example:

```
str1 = "John"
if str1.isalpha():
    print str1, "is alphabetic."
else:
    print str1, "is not alphabetic."
str2 = "My name is John"
if str2.isalpha():
    print str2, "is alphabetic."
else:
    print str2, "is not alphabetic."
```

Output:

```
John is alphabetic.
My name is John is not alphabetic.
```

Method `isdigit()`

Return true if all characters in the string are digits and there is at least one character, false otherwise.

Example:

```
str1 = "2012"
if str1.isdigit():
    print str1, "is a number."
else:
    print str1, "is not a number."
str2 = "Year 2012"
```

```
if str2.isdigit():
    print str2, "is a number."
else:
    print str2, "is not a number."
```

Output:

```
2012 is a number.
Year 2012 is a number.
```

Method `join(seq)`

Return a string which is the concatenation of the strings in the sequence seq. The base string (onject whose method is used as separator).

Example:

```
str = "..."
str1 = "This"
str2 = "is"
str3 = "the"
str4 = "movie."
print str.join([str1, str2, str3, str4])
```

Output:

```
This...is...the...movie.
```

Method `lower()`

Return a copy of the string converted to lowercase.

Example:

```
str = "She lives in New Orleans."
print str.lower()
```

Output:

```
she lives in new orleans.
```

Method `replace(old, new[, count])`

Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

Example:

```
str = "First word, second word, and third word."
print str.replace("word", "thing")
```

Output:

```
First thing, second thing, and third thing.
```

Method `swapcase()`

Return a copy of the string with uppercase characters converted to lowercase and vice versa.

Example:

```
str = "The abbreviation is MENSA"
print str.swapcase()
```

Output:

```
tHE ABBREVIATION IS mensa
```

Method `title()`

Return a titlecased version of the string: Words start with uppercase characters, all remaining cased characters are lowercase.

Example:

```
str = "this is the title of my new book"
print str.title()
```

Output:

```
This Is The Title Of My New Book
```

Method `upper()`
Return a copy of the string converted to uppercase.

Example:

```
str = "this is the title of my new book"
print str.upper()
```

Output:

```
'THIS IS THE TITLE OF MY NEW BOOK'
```

For additional useful text string methods see the Python documentation at `http://docs.python.org/release/2.5.2/lib/string-methods.html`.

Before we leave this section, notice that lists, tuples, dictionaries and many other data structures in Python are classes as well.

13 Class Inheritance

13.1 Objectives

- Understand the benefits of inheritance in object-oriented programming.
- Learn the syntax of creating descendant classes in Python.
- Learn how methods of parent class are called from the descendant.

To illustrate the inheritance process in object-oriented programming, we will build a small library of geometrical objects that contains triangles, quadrilaterals, rectangles, squares, and circles. We want all these classes to have the same functionality as the class `circle` from Subsection 12.6. This means, they should be able to:

- Calculate their area,
- calculate their perimeter,
- draw their geometry.

Notice one interesting thing: The drawing functionality will be the same for all these classes. Specifically, all of them will have the coordinate arrays `ptsx`, `ptsy` and the plotting method `draw()`. Other than that their functionality will differ. This is where *class inheritance* comes handy.

13.2 Creating base class `geometry`

Instead of repeating the same drawing functionality in all classes, we create a base class `geometry` as follows:

```
from pylab import clf, plot, legend, axis

# Base class:
class geometry:
    # Constructor:
    def __init__(self):
        self.ptsx = []
        self.ptsy = []

    # Plotting method:
    def draw(self, label):
        axis('equal')
        plot(self.ptsx, self.ptsy, label = label)
        legend()
```

This base class really cannot do much, but this is fine. We will use it to derive several new (descendant) classes. The benefit of inheritance is that if we decide in the future to change the plotting mechanism, we will just have to change it at one place – in the base class `geometry` – and the change will propagate automatically into all its descendants. Remember that humans are extremely prone to mistakes. Having the same code at several different places in the program is about the worst thing a programmer can ever do.

13.3 Deriving `polygon` from `geometry`

There is more to the story. Triangles, quadrilaterals, rectangles and squares are all *polygons*. Hence let us create the class `polygon` to be a descendant of the base class `geometry`. This class will represent general non-convex counter-clockwise (CCW) oriented polygons. Notice the syntax of the inheritance – one types `class polygon(geometry):` instead of just `class polygon`:

```
# Class polygon derived from class geometry,
# represents general, possibly non-convex
# polygon. Boundary needs to be CCW oriented.
class polygon(geometry):
    # Constructor (here L is list of vertices):
    def __init__(self, L):
        self.ptsx = []
        self.ptsy = []
        # Convert L into plotting arrays:
        for pt in L:
```

```
            self.ptsx.append(pt[0])
            self.ptsy.append(pt[1])
        # To close the loop in plotting:
        self.ptsx.append(L[0][0])
        self.ptsy.append(L[0][1])

    # Method to calculate area of a general
    # oriented polygon:
    def area(self):
        # Calculate minimum y-coordinate:
        ymin = min(self.ptsy)
        # The area of an oriented polygon is the sum of
        # areas of oriented trapezoids under its edges:
        m = len(self.ptsx) - 1    # Number of trapezoids.
        s = 0
        for i in range(m):
            s += (self.ptsx[i+1] - self.ptsx[i]) \
                 * (self.ptsy[i+1] + self.ptsy[i] \
                 - 2*ymin) / 2.
        return -s

    # Method to calculate perimeter of a general
    # oriented polygon:
    def perimeter(self):
        l = 0
        m = len(self.ptsx) - 1
        for i in range(m):
            l += sqrt((self.ptsx[i+1] - self.ptsx[i])**2
                 + (self.ptsy[i+1] - self.ptsy[i])**2)
        return l
```

Notice that we did not have to define the method `draw()` here.

13.4 Deriving `circle` from `geometry`

Next let us derive the class `circle` from the class `geometry`. Obviously it cannot be derived from `polygon` because circles are not polygonal geometries:

```
# Class circle derived from class geometry:
class circle(geometry):
```

```
    # Constructor (here r is radius, (cx, cy) the center point,
    # and n plotting subdivision):
    def __init__(self, r, cx, cy, n = 100):
        self.R = r
        self.Cx = cx
        self.Cy = cy
        self.n = n
        # Create array of points for plotting:
        self.ptsx = []
        self.ptsy = []
        da = 2*pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * cos(i * da))
            self.ptsy.append(self.Cy + self.R * sin(i * da))
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)

    # Method to calculate circle area:
    def area(self):
        return pi * self.R**2

    # Method to calculate perimeter:
    def perimeter(self):
        return 2 * pi * self.R
```

Again, notice that we did not have to define the method `draw()`.

13.5 Deriving `triangle` and `quad` from `polygon`

The `polygon` class defined above is very general. Of course we could use it directly to render triangles, quads, rectangles, and squares. But these geometries are simpler and thus we also want their instantiation to be simpler. Let us begin with the class `triangle`.

This class is very simple – it takes three points as parameters, forms a list containing these three points, and then the list is passed into the constructor of the parent class `polygon`. Notice the way the constructor of the `polygon` class is called from its descendant `triangle` via `polygon.__init__(self, L)`:

```
# Class triangle derived from class polygon:
class triangle(polygon):
    # Constructor (here a, b, c are the vertices):
```

```
    def __init__(self, a, b, c):
        L = [a, b, c]
        polygon.__init__(self, L)
```

Class `quad` is created analogously:

```
# Class quad derived from class polygon:
class quad(polygon):
    # Constructor (here a, b, c, d are the vertices
    # in CCW orientation):
    def __init__(self, a, b, c, d):
        L = [a, b, c, d]
        polygon.__init__(self, L)
```

13.6 Deriving `rectangle` from `quad`

Next, rectangle is a special case of quad, so let us derive the `rectangle` class from the `quad` class:

```
# Class rectangle derived from class quad.
# Represents rectangle (0, a) x (0, b):
class rectangle(quad):
    # Constructor:
    def __init__(self, a, b):
        quad.__init__(self, [0, 0], [a, 0], [a, b], [0, b])
```

13.7 Deriving `square` from `rectangle`

Taking this one step further, square is a special case of rectangle. Therefore we can derive the `square` class from the `rectangle` class:

```
# Class square derived from class rectangle.
# Represents square (0, a) x (0, a):
class square(rectangle):
    # Constructor. Here a is the edge length:
    def __init__(self, a):
        rectangle.__init__(self, a, a)
```

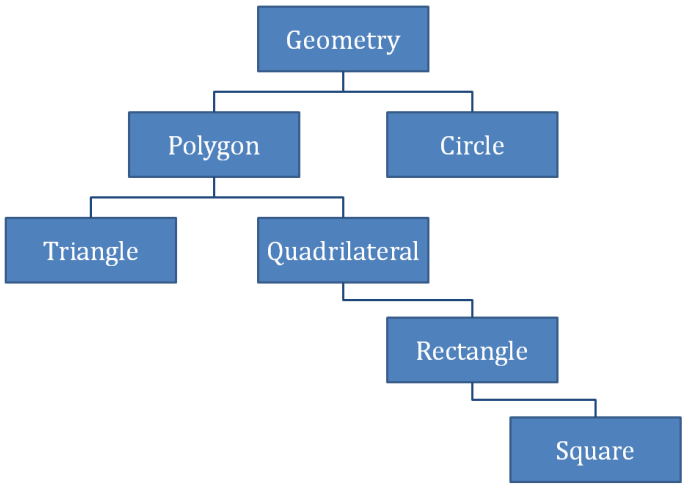


Fig. 34: Graphical representation of the structure of our classes.

13.8 Diagram of class structure

The inheritance structure of our classes is depicted in Fig. 34.

13.9 Creating sample instances

Finally let us create sample instances of the classes defined above, inquire about their areas and perimeters, and let them plot themselves:

```
# Create a triangle:
T = triangle([-2, -0.5], [0, -0.5], [-1, 2])
print "Area and perimeter of the triangle:", \
T.area(), T.perimeter()

# Create a quad:
Q = quad([-3, -1], [0, -1], [-1, 1], [-2, 1])
print "Area and perimeter of the quad:", \
Q.area(), Q.perimeter()

# Create a rectangle:
R = rectangle(3, 1)
print "Area and perimeter of the rectangle:", \
R.area(), R.perimeter()
```

```
# Create a square:
S = square(1.5)
print "Area and perimeter of the square:", \
S.area(), S.perimeter()

# Create a circle:
C = circle(2.5, 0, 0.5)
print "Area and perimeter of the circle:", \
C.area(), C.perimeter()

# Plot the geometries:
clf()
T.draw("Triangle")
Q.draw("Quad")
R.draw("Rectangle")
S.draw("Square")
C.draw("Circle")
ylim(-3, 4)
legend()
lab.show()
```

First, here is the textual output:

```
Area and perimeter of the triangle: 2.5 7.38516480713
Area and perimeter of the quad: 4.0 8.472135955
Area and perimeter of the rectangle: 3.0 8.0
Area and perimeter of the square: 2.25 6.0
Area and perimeter of the circle: 19.6349540849 15.7079632679
```

The graphical output is shown in Fig. 35.

Bézier curves:

Bézier curves are the most widely used approach to define curves in engineering design. Let us begin with the linear case. Linear Bézier curve is a straight line that connects two control points P_1 and P_2. All Bézier curves are parameterized from the interval $[0, 1]$, so in the linear case the exact mathematical definition is

$$B(t) = P_1 + t(P_2 - P_1)$$

You can easily check that the curve starts at P_1 (substitute $t = 0$ to the above formula), and that it ends at P_2 (substitute there $t = 1$).

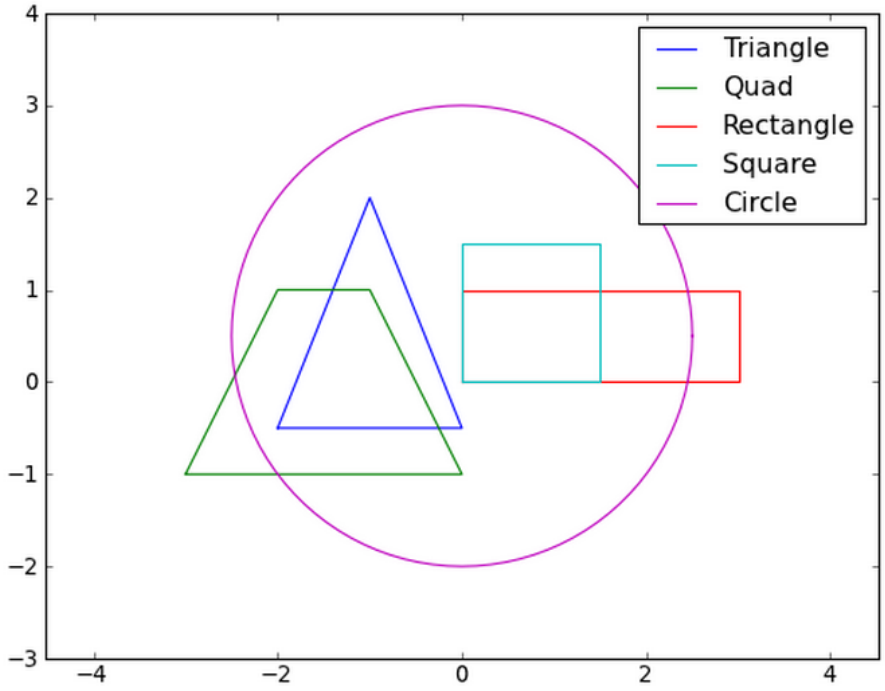


Fig. 35: Visualization of the instances created above.

Quadratic Bézier curves are defined using three control points P_1, P_2 and P_3. Again they are parameterized from $[0, 1]$, and their definition is

$$B(t) = (1-t)^2 P_1 + 2(1-t)tP_2 + t^2 P_3.$$

Cubic Bézier curves are defined using four control points P_1, P_2, P_3 and P_4. They are parameterized from $[0, 1]$ as usual, and their definition is

$$B(t) = (1-t)^3 P_1 + 3(1-t)^2 tP_2 + 3(1-t)t^2 P_3 + t^3 P_4.$$

14 Python Libraries

14.1 Objectives

- Understand that using libraries enhances your problem solving skills.
- Learn basic facts about Scipy, Numpy, Pylab, Matplotlib, and Sympy.
- Learn where you can obtain more information about these libraries.
- Learn that Internet is the best resource for Python programmers.

Using libraries is an indivisible part of Python programming. As opposed to commercial products, Python libraries are free. We have used some of them already, such as Pylab and Matplotlib in Section 4 and Numpy in Section 7.

14.2 Python Standard Library

The Python Standard Library has vast functionality related to built-in functions, built-in types, built-in constants, built-in exceptions, string services, numerics and mathematics modules, files, data compression, cryptographic services, and many others. Just introducing a table of contents would take several pages. To learn more, to visit `http://docs.python.org/library/`. Also, a long list of packages for Python programmers can be found at `http://pypi.python.org/pypi/`.

14.3 Pylab and Matplotlib

The Pylab library can be used for numerical computation but as there are other and stronger tools, we recommend to use it mainly for plotting, along with the Matplotlib library. For more details on Pylab visit `http://www.scipy.org/PyLab`. This page is not dedicated to plotting though, more on concrete plotting functionality can be found on the home page of Matplotlib, `http://matplotlib.sourceforge.net`. This page contains an exhaustive overview of plotting-related functionality, many examples, gallery, and documentation.

14.4 Scipy and Numpy

Scipy is a general library for scientific computing with Python. Numpy can be viewed as its "heart" containing numerical algorithms and methods. These methods comprise interpolation, integration, optimization, Fourier transforms, signal processing, linear algebra, eigenvalue problems, graph theory routines, statistics, ordinary and partial differential equations, image processing, etc. To learn more, visit `http://www.scipy.org` and `http://numpy.scipy.org/`.

14.5 Sympy

Sympy is an excellent library for symbolic mathematics covering high-school algebra, calculus, matrix algebra and differential equations. To learn more about Sympy, visit its home page (`http://www.sympy.org`).

15 Other recommended topics in Python

There are many topics that we have decided to skip in the first reading of the Python language. There is more to learn to almost every command, function aned concept described in this document. If you like Python, your nexr resource should be the original Python tutorial at `http://docs.python.org`.

16 What next?

Congratulations, you made it! We hope that you enjoyed the textbook and the exercises. If you can think of any way to improve the application Karel the Robot or this tutorial, we would be very happy to hear from you. If you have an interesting new game or exercises for Karel, please let us know as well.

Although you may feel like an Almighty Programmer right now, we would recommend staying humble. Even the most experienced programmers are learning new things all the time. There is much more to Python that we managed to cover in this introductory textbook. You already know about the Internet resources where you can learn more.

Alternatively, you may dive into a next programming language! We would recommend Javascript since this is the most popular language for web development. Of course there are many more languages to explore, including C/C++, Java, Perl, Ruby, Lua and others.

In any case, our team wishes you good luck, and keep us in your favorite bookmarks!

Your NCLab Team