Instruction Manual Programming for Economists

Jasper Veltman

Third Edition

Published May 2017

VU UNIVERSITY AMSTERDAM

Contents

Origin of the manual

This manual is largely based on the third international edition of the Instruction Manual Introduction to Programming. A more detailed history of this manual can be found in the third international edition and the thirtieth edition of the Dutch *Practicumhandleiding Inleiding Programmeren*. This manual has been adapted to reflect the change of the programming language used during this practical. Previously, Java was the programming language used for all study programmes during practicals. Starting from the academic year 2012-2013, Java will be replaced by Python in the courses for *Information, Multimedia and Management* and *Lifestyle Informatics* students. This adaptation has been performed by Jan Stienstra in co-operation with a reviser; Bram Veenboer.

Jan Stienstra, July 2012

In the third edition of this manual, the MAC support for the ipy_lib is added. As well as some minor changes and fixes.

Marcel de Vries, October 2013

Some changes have been made to make the manual fit for the course 'Programming for Economists'.

Shanti Bruyn, May 2015

Statistics has been replaced by Administration. Futher the graded assignments got a change in weighting due to the 'Second Assignment'.

Shanti Bruyn, May 2016

0 Syllabus

Course format

This course features a series of lectures and parallel lab sessions. During the lectures, theory on programming using the Python programming language, is taught. During the lab sessions, programming is practiced by making assignments using the Python programming language. Assignments should be prepared in advance, at home. Students will be assigned to groups. Every group will have a teaching assistant, who will assist with the assignments and grade the deliverables.

Course documents and assignments

Book During this course the online book *Learning with Python*, 2nd Edition, by Jeffrey Elkner, Allen B. Downey, and Chris Meyerswill be used. It can be found here. Parts of this book will be treated during the lectures. During the lab sessions, you are supposed to take your lecture notes with you.

Modules The course material is devided into five modules. These modules contain additional theory and assignments and will be used as an instruction manual during the lab sessions. Theory treated during the lectures will be repeated as little as possible in the modules. The modules will however feature notes on programming style.

Assignments Every module consists of theory and assignments. For module 3-4 only the last assignment of a module will be graded, in module five both assignments will be graded. This does not mean that the other assignments in module 3-4 are less important. All the assignments in a module will train essential skills needed to succesfully complete the graded assignment. *Every* assignment has to be approved by the teaching assistent to pass the course. You should not start working on an assignment before completing all previous assignments. This way, you will not make possible mistakes twice. Teaching assistants will therefore not answer any questions on an assignment, if the previous assignments have not yet been completed.

Deadlines and assessment

Graded assignments have to be submitted to Practool. The teaching assistent will only provide help with an assignment if all previous ungraded assignments have been approved. When the quality of an ungraded assignment is not sufficient, it has to be corrected according to the provided feedback. Graded assignments cannot be resubmitted once a grade has been given. Feedback on these assignments can therefore not be used to improve a program in order to receive a higher grade. Nevertheless, it is advised to process the feedback received on graded assignments as well.

Deadlines For modules 3 and 4, only the last assignment will be graded. However for module 5 both assignments will be graded. **Deadlines will be posted on Canvas.** Late submission will be accepted, but a point is deducted for every day the assignment is late. Pay close attention: **no work will be accepted after the practicum is finished.** The grades are weighted in the following way:

Module	Graded Assignment	Weight
1	HelloWorld2	slipday
2	All	slipday
3	GeographyGrades	10%
4	Administration	15%
5	House Market	20%
	Second Assignment	55%

Slipdays The first two modules do not include graded assignments, however if you hand in these assignments correctly your TA can grant you a *slipday*. You can earn at most two slipdays. When a graded assignment is handed in late, one point will be deducted for each day. After the last module your TA will use your slipdays to reduce the 'late days' penalty on the most heavy weighted assignments. Both slipdays can be used for the same assignment.

Chapter	grade	days late
1	1 slipday	
2	1 slipday	
3	7	0
4	8	2
5	6	1
	7	-

Table 1: Example of how to calculate with slipdays

$$\text{Your final mark would've been: } \frac{7*10+(8-2+\mathbf{1})*15+(6-1+\mathbf{1})*20+7*55}{100}.$$

Want a higher grade? Once you've gotten a grade for a graded assignment and you would like a higher grade, you can only do this by handing in a Supplementary Exercise. These can be found on Canvas. You cannot hand in the

same assignment another time.
The last grade will count, even if this turn out to be a lower grade.

Final Grade You have passed the lab when the average grade of all the graded assignments is ($\geq 5,5$). Assignments that have not been submitted will be graded with a $1,0$.
You passed the "Programming for Economists" course if the lab is passed: $\geq 5,5$.

1 Editing, Compiling and Executing

Abstract

This chapter will introduce the IDE (Integrated Development Environment) PyCharm and explain how to organize Python files and execute programs.

Goals

- Use PyCharm to open, edit, save and organize Python-files
- Execute Python code

Introduction to Python and PyCharm

Installing Anaconda

VU Anaconda has already been installed on all the VU computers that will be used during the lab sessions.

At home
In this course we are going to program in **Python 2.7**, and use a library for graphical representations. This library makes use of the Python package "matplotlib". Since matplotlib has a lot of other dependencies, it is easiest to install a Python distribution that includes matplotlib. In this course we use the Python 2.7 distribution called "Anaconda".

- **64 bit Windows:** Download Anaconda from this link.
- **32 bit Windows:** Download Anaconda from this link.
- **MacOS:** Download Anaconda from this link.

Once you have downloaded the Anaconda installer, click the downloaded file and follow the instructions that appear on the screen.

Installation and starting PyCharm

VU PyCharm has already been installed and configured on all the VU computers that will be used during the lab sessions.

At home
To download PyCharm, browse to `https://www.jetbrains.com/pycharm/download/` and download **PyCharm Community Edition**. Follow the instructions in the installer.

Once PyCharm has been installed, follow these steps to setup the editor:

1. Download the Practical plugin from Canvas ("Practical.jar").
2. Start PyCharm.
3. If this is the first time PyCharm is started, it will take you through some setup options.
4. Once you have gone through the setup steps, a welcome screen will appear.
5. At the bottom right of the welcome screen is a button called **Configure**. Click this button.
6. In the dropdown menu that appears, click **Plugins**.
7. At the bottom right of the plugins screen is a button called **Install plugin from disk...**. Click this button.
8. Select the Practical plugin that was downloaded from Canvas on the computer, then click **OK**.
9. Click **OK** again. If PyCharm asks you if you want to restart it, click **Yes**.

Now PyCharm and Practical should be installed.

Creating a project The *project* is a folder in which all created files are saved. For this course, only one project needs to be created. When PyCharm is started and no projects have been made yet, it will automatically ask you to create a new project (the button is called **Create New Project**). Click this button and create a **Pure Python** project. The *Location* field decides where on your computer the files in this project will be saved. The *Interpreter* field decides which Python interpreter will be used for this project. Select the Anaconda interpreter. Click **Create**, and the project will be created.

Arranging files

After the project has been created, a new screen will show. On the left of the current window, the Package Explorer is shown. This is where you can see all files belonging to a project. Files can be organized in folders. The rest of this theory explains how to organize a module in PyCharm.

1. Create a new **Directory** (folder). To do this right-click in the Package Explorer and select New → Directory. The name of this directory will be **Module 1**.

2. Create a new **Directory** inside of the directory created in the previous step. To do this right-click the directory created in the previous step (called *Module 1* and select New → Directory. The name of this directory will be **Hello World 1**.

3. Every assignment will be in its own **Python File**. To do this, right-click on the directory created in the previous step (called *Hello World 1*) and select New → Python File. The name of the **Python File** will be the same as the name of the assignment.

Compiling and executing programs

When a new Python File is created, it is called a *skeleton*; an empty program, that does nothing. As a start, create a new Python File called hello_world.py, and copy the following code to this Python File:

```
print "Hello World"
```

Programs are compiled automatically in PyCharm. Right-click on the file to be executed and select **Run** (the button with a green arrow next to it). The output of the program will be printed in the Console, at the bottom of the screen. If the program expects input, it can be typed into the Console as well.
Programs can only be executed if they are syntactically correct. If there are any errors, these are underlined in red. Hover the mouse over the underlined words to show an error message.

Submitting assignments

A graded assignment needs to be submitted to PracTool (`phoenix.labs.vu.nl/practool`). The process of correctly submitting assignments is given below:

1. Export all the files of the assignment to a .zip-file. Right-click on the **Directory** in the Package Explorer to be exported and select Export as ZIP. Navigate to the folder where the ZIP file should be stored. Make sure the fila name is the name of the assignment and is followed by your VUnet-id seperated by a hyphen. For example: `pirate-rhg600.zip`.

   Click **Open** and the .zip-file has been created in the directory that has been selected.

2. Submit the .zip-file to PracTool. Login to PracTool (register an account if you do not already posses one, PracTool is NOT linked to VUnet or Canvas). Browse My Computer and select the .zip-file. Click on Submit. Wait for PracTool to confirm the submission and click on Finish.

If your VUnet-id is rhg600, the files you submit should be named in the following way:

Module	Name
1	hello_world2-rhg600.zip
2	chapter2-rhg600.zip
3	geographygrades-rhg600.zip
4	administration-rhg600.zip
5	houseMarket-rhg600.zip
	-

Note: chapter2-rhg600.zip contains all the assignments from chapter2.

Warning
Assignments can only be processed if they are submitted in the format described above. Do not submit files in any other format!

Trial submission

Create a new Python File hello_world2 and copy hello_world to hello_world2.

Edit the program in such way that it will ask for your name:

```
name = str(raw_input("Enter your name: "))

print "Hello world!! written by: %s" % name
```

Add a comment to the top of your code which includes the name of the assignment, the date of completion and your name. This ensures that your teaching assistant knows which assignment belongs to whom. For example:

```
''' Assignment: hello_world2
    Created on 25 aug. 2012
    @author: Jan Stienstra '''

name = str(raw_input("Enter your name: "))

print "Hello world!! written by: %s" % name
```

Test the program. Does it work as expected? If so, hand in the program by submitting it on Practool.

This program is not graded like the other assignments that have to be submitted. It is possible to earn a slipday if the program is submitted on time. The syllabus provides more details on slipdays. The goal of this assignment is to make sure that you can submit programs.

2
If statements and loops

Abstract

The first few programs in this chapters will read from *standard input* and write output to *standard output*. These programs will be very simple. The focus in the first part of this chapter will be on writing programs with a clear layout using well chosen names. The second part of this chapter will introduce if-statements and loops.

☛ **Warning**

This chapter contains nine assignments of variable size. Make sure to utilize the time given to you during the lab sessions. The lab sessions only provide sufficient time if you write your programs in advance. This way, any problems you encounter whilst writing your programs can be resolved during the lab sessions.

Goals

- The use of clear identifiers.
- Familiarize with if, else and elif statements and recognize situations in which to apply these.
- Familiarize with for and while loops and recognize situations in which to apply these.

Instructions

- Read the theory about **Efficient programming** and **Constants**. With this information in mind, make the assignments **VAT**, **Plumber** and **Othello 1**.
- Read the theory about **Identifiers** and **If-statements**. With this information in mind, make the assingments **Electronics** and **Othello 2**.

- Study your lecture notes on **Loops**. With this information in mind, make the assignments **Manny**, **Alphabet**, **Collatz** and **SecondSmallest**.

Theory

Efficient programming

Once upon a time, running a computer was so expensive that any running time that could be saved was worthwile. Programs had to contain as little lines of code as possible and programs were designed to run fast; clear code was not a priority. Such a programming style is called machine-friendly nowadays. Luckily, the situation has changed.
Programs that have been written in the past often need altering in one way or another. If a program was written in a machine-friendly, but incomprehensible programming style, it is almost impossible to edit it. After half a year, one easily forgets how the program works exactly. Imagine the problems that could occur, when the programmer that wrote the code no longer works for the company, that wants to edit it.
The logical implication of this programming style is that programs are not changed at all. Everyone has to work with the, then well-intentioned 'features', that are no longer maintainable.
Running programs is becoming increasingly less expensive. Programmers, on the other hand, are only getting more expensive. Efficient programming therefore does not mean:

> "writing programs that work as fast as possible."

but

> "writing programs that require as little effort and time possible to be
>
> - *comprehensible*
> - *reliable*
> - *easily maintained*."

This will be one of the major themes during this course. Assignments are not completed when the program does what the assignment asks them to do. Programs are only approved when they meet the standards described above.
Theory provided in this Instruction Manual is an addition to the lectures and book. The book will teach you the syntax and basic functionality, this instruction manual will teach you how to do this, taking the standards described above into account.

Rule of Thumb
Try to refrain from writing lines of code longer than the screen width. If it is impossible to write code on a single line, a \ can be used to continue on a new line.

Constants

Although Python does not support constants in the context of unchangeable variables, like those in the Java and C programming languages, the principles of using constants are upheld in Python. In other words, although Python does not support constants, variables can be used as if they were constants. Imagine a program that reads a number of addresses from a file and prints them on labels, thirty characters wide, six lines high. All of the sudden, the wholesale company changes the size of the labels to thirty-six characters wide and five lines high.
Fortunately, the program looks like this:

```
''' Assignment: Labels
    Created on 6 aug. 2012
    @author: Jan Stienstra '''


# This program reads adresses from input,
# and prints them in a specific format.

LABEL_WIDTH = 30 # characters
LABEL_HEIGHT = 6 # lines

# etc...
```

The only thing that needs to be done, is to change the two constants and recompile the program.

Errors that can occur when a program does not incorporate constants are:

- The code contains a `6` on 12 different places and is only replaced on 11 places by a `5`
- Derived values like `5` (= `LABEL_HEIGHT - 1`) are not changed to `4` (= `LABEL_HEIGHT - 1`)!

Constants do not only ease the maintenance of a program, but can increase the comprehensibility of the code as well. When a constant, like `LABEL_HEIGHT`, is used, it is imediately clear what this number represents instead of only knowing its numerical value. This property gives constants an added value. Therefore, it is advisable to always use constants in your programs.

✎ **Rule of Thumb**
All numbers used in a program are constants, except 0 and 1.

Example The following example program will read a number of miles from the standard input and prints the equivalent number of kilometers on the output. Take special notice to the use of identifiers, constants and layout.

```
''' Assignment: MileInKilometers
    Created on 6 aug. 2012
    @author: Jan Stienstra '''


MILE_IN_KILOMETERS = 1.609344
```

```
number_of_miles = int(raw_input("Enter the number of miles: "))

number_of_kilometers = number_of_miles * MILE_IN_KILOMETERS

print "%f miles equals %f kilometer" % \
        (number_of_miles, number_of_kilometers)
```

☞ Make the assignments **VAT**, **Plumber** and **Othello 1**.

Identifiers

All constants, types, variables, methods and classes have to be assigned a name. This name is called the identifier. This identifier has to be unique within the class that it is defined in. This might seem easier than it is. In this practical you will learn to choose the right identifier for the right object.

The importance of the right name The identifier that is assigned to an object should reflect the information it contains. When a variable to maintain a record of the number of patients in a hospital is needed, *n* would not suffice as a identifier for this variable. The identifier *n* does not specify the information the variable contains. When the identifier *number* is chosen, the problem seems to be resolved. However, it is still unclear to which number the identifier refers. Is it the number of doctors? Is it the number of beds? No, it is the number of patients. That is why this variable should be called *number_of_patients*. It might take some time to find an appropriate identifier in some cases, but it is certainly worth the effort. This ensures that everyone will understand your program, including the teaching assistant.

Example A long, long time ago, the maximum length of identifiers in some programming languages was limited. All information about the contents of the variable had to be contained in six or seven characters. This meant that it was very difficult to find clear and understandable identifiers. As a result, programs were often hard to read. A program that had to find travel times in a timetable would contain identifiers like:

```
ott # outward travel time, in minutes
rtt # return travel time, in minutes
```

The introduction of programming languages like Pascal significantly improved the readability of code by removing the restriction on identifier lengths. Like Pascal, Python does not limit the length of identifiers. Therefore the identifiers in the example can be rewritten:

```
outward_travel_time # in minutes
return_travel_time # in minutes
```

Abbreviated identifiers Uncommon abbreviations should not be used as identifiers, as the example above illustrates. Identifiers do not necessarily have to be long to be understandable. In mathematics for example, characters are often used to denote variables in equations. Let's have a look at the quadratic equation:

$$ax^2 + bx + c = 0$$

A quadratic equation has at most two solutions if the discriminant is larger than zero:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A program to solve a quadratic equation would contain the following code:

```
discriminant = (b * b) - (4.0 * a * c)
if discriminant >= 0:
    x1 = (-b + math.sqrt(discriminant)) / (2.0 * a)
    x2 = (-b - math.sqrt(discriminant)) / (2.0 * a)
```

Note that this implementation uses the identifiers *a*, *b* and *c* in the same way as the mathematical definition. Readability would not improve if these identifiers would be replaced by *quadratic_coefficient*, *linear_coefficient* and *constant_term*. It is clear that using *a*, *b* and *c* is the better choice. The identifier *discriminant* is used as no specific mathematical character is defined for it. The module `math` identifies the method to calculate a square root with `sqrt()`. The module `math` identifies a number of other methods with equally well known abbreviations. For example: `cos()`, `log()` and `factorial()`.

Exceptions There are some conventions for identifiers. An example for calculating the factorial of $n > 0$:

```
factorial = n
for i in range(1, n):
    factorial *= i
```

The identifier for the variable *n* is not changed into *argument*. Numerical arguments are by convention often identified as *n*. Variables that are used for iterations are similarly not identified as *counter*, but as *i*. When more than one iterator is used, it is common practice to use *j* and *k* as identifiers for the next iterators.
Let us look at another example. When programming a game of chess, the pieces on the board can be identified by *ki* (king), *qu* (queen), *ro* (rook), *bi* (bishop), *kn* (knight) and *pa* (pawn). Everyone with a elemental knowledge of chess will surely understand these abbreviations, one might think. However, if someone else reads this program *kn* might be interpreted as king and *ki* as knight. This example shows the need of *'psychological distance'* between two identifiers. The psychological distance between identifiers cannot be measured exactly. Psychological distance is roughly defined as large when the chance of confusion between identifiers is small. On the contrary, the psychological distance is small when confusion between identifiers is almost inevitable. Two identifiers with a very small psychological distance are the identifiers in the first timetable example.

Rule of Thumb
Identifiers which are used a lot in the same context, need to have a large psychological distance.

Conventions

One important restriction for choosing identifiers is that they cannot contain whitespace. It is common practice to write identifiers consisting of multiple words by capitalising each word, except the first. In this practical the following guidelines are in place:

- Names of variables, methods and functions are written in lowercase, with words seperated by underscores.

  Example: `number_of_students`
  Example: `read_line() { ... }`
  Example: `calculate_sum_of_profit(x)`

- Identifiers identifying constants are written in upper case. If an identifier for a constant consists of multiple words they are separated by underscores.

  Example: `MAXIMUM_NUMBER_OF_STUDENTS = ...`

- Identifiers identifying a module are written in the same way as variables.

- Identifiers identifying a class are written in lowercase, with the first letter of all the words capitalized.

  Example: `Library`
  Example: `AgeRow`

Self test

Expressions 1

The following questions are on expressions. These questions do not need to be turned in. Do make sure you are able to answer all the questions posed below, as this knowledge is vital in order to make the exam in good fashion. For all questions write the generated output, or indicate an error. In addition write down every expression in a question and denote the type of the resulting value of the expression.

Question 1

```
result = 2 + 3
```

Question 2

```
result = 1.2 * 2 + 3
```

Question 3

```
result = "ab" + "cd"
```

Question 4

```
result = ord('c') - ord('a') + ord('A')
result = chr(result)
```

Question 5

```
result = True or False
```

Question 6

```
result = 17 / 4
```

Question 7

```
result = 17 % 4
```

Question 8

```
if True :
    print "not not true"
```

Question 9

```
if False :
    print "really not true"
```

Question 10

```
if 2 < 3 :
    print "2 is not larger or equal to 3"
```

Question 11

```
if (3 < 2 and 4 < 2 and (5 == 6 or 6 != 5)) or True :
    print "too much work"
```

Question 12

```
number = '7'
print "%c" % number
```

Question 13

```
if False and (3 > 2 or 7 < 14 or (5 != 6)) :
    print "finished quickly"
```

Expressions 2

The following questions are on expressions. For all questions write the generated output, or indicate an error. In addition write down every expression in a question and denote the type of the resulting value of the expression.

Question 1

```
def function() :
    number = 2
    return number / 3

result = function() * 3
```

Question 2

```
def world_upside_down() :
    numbers_upside_down = 2 > 3
    booleans_upside_down = True == False

    return numbers_upside_down and booleans_upside_down

if world_upside_down() :
    print "The world is upside down!"
else :
    print "The world is not upside down."
```

Question 3

```
def awkward_number() :
    character = 'y'
    return 'z' - character


print "The result is awkward " + "result: \%s" %
            awkward_number()"
```

Question 4

```
if 'a' < 'b' :
    print "smaller"
```

Question 5

```
if 'a' > 'B' :
    print "hmmm"
```

Question 6

```
number = '7'
print "%d" % number - 1
```

If-statements

☞ Study your lecture notes on if-statements. Section **3.1** in the book will provide additional information on if-statements.

Example This example program will read an exam grade and prints whether this student has passed.

```
''' Assignment: MileInKilometer
    Created on 6 aug. 2012
    @author: Jan Stienstra '''


PASS_MINIMUM = 5.5

grade = float(raw_input("Enter a grade: ")

if grade >= PASS_MINIMUM :
    print "The grade, %0.2f, is a pass." % grade
else :
    print "The grade, %0.2f, is not a pass." % grade
```

This example can also be implemented using a ternary operator as described in the section Layout.

☞ Make the assignments **Electronics** and **Othello 2**.

Assignments

1. VAT

☞ Before starting with this assignment, read the theory about **Efficient programming** en **Constants**.

Write a program that takes the price of an article including VAT and prints the price of the article without VAT. The VAT is currently 21.00%.

Example Using an input of 121 the output will be:[1]

```
Enter the price of an article including VAT: 121
This article will cost 100.00 euro without 21.00% VAT.
```

2. Plumber

The employees at plumbery 'The Maverick Monkey' are notorious bad mathmaticians. Therefore the boss has decided to use a computer program to calculate the cost of a repair. The cost of a repair can be calculated in the following way: the hourly wages multiplied by the number of billable hours plus the call-out cost. The number of billable hours is the number of hours worked rounded to the nearest integer. Plumbing laws fix the call-out cost at €16,00.

Example A plumber earning €31.50 an hour, working for 4.5 hours should get the following output.

```
Enter the hourly wages: 31.50
Enter the number of hours worked: 4.5
The total cost of this repair is: 173.50 euro
```

3. Othello 1

The goal of this assignment is to give some information about the outcome once a game has finished. This information is obtained by two measurements:

- The percentage of black pieces of all the pieces on the board.
- The percentage of the board covered in black pieces.

The Othello board measures eight squares by eight squares, making the total number of squares sixty-four.
Write a program that takes the number of white pieces followed by the number of black pieces as input. Print the two percentages as output.

Example

```
Enter the number of white pieces on the board: 34
Enter the number of black pieces on the board: 23
The percentage of black pieces on the board is: 35.94%
The percentage of black pieces of all the pieces on the board is: 40.35%
```

In case you have gotten interested in the game of Othello, you can find more information about it here. You will not need this information for this course.

[1] Examples will have input printed in italics.

4. Electronics

☞ Before starting this assignment, read the theory about **Identifiers** and **If-statements**.

The electrics company 'The Battered Battery' is nearly bankrupt. To avoid total disaster, the marketing branch has come up with a special sale to attract more customers. Whenever a customer buys three products, he or she receives a 15% discount on the most expensive product. Write a program that takes the prices of three products as input and prints the discount and final price as output. Remember that the goal of making the assignments in this chapter is to practice the use of if statements. Therefore, do not use the built-in function max in this assignment.

Example Determine the reduction and final price if the three products cost €200, €50 and €25 respectively.

```
Enter the price of the first article: 200
Enter the price of the second article: 50
Enter the price of the third article: 25
Discount: 30.00
Total: 245.00
```

5. Othello 2

During a game of Othello the time a player spends thinking about his moves is recorded. Write a program that takes the time that two players have thought, one human, one computer, in millisecond as input. The program determines which of the two players is human and prints the thinking time of the human in the following format: *hh*:*mm*:*ss*. It may be assumed that a computer always has less thinking time than a human.

Example

```
Enter the time the black player thought: 21363
Enter the time the white player thought: 36
The time the human player has spent thinking is: 00:00:21.
```

6. Manny

☞ The following four assignments use loops. Use the right loop for the right assignment; use both the for statement and the while statement twice.

Mobster Manny thinks he has found the perfect way to part money from their rightfull owners, using a computer program. Mobster Manny secretly installs the program on someone's computer and remains hidden in a corner, waiting for the program to finish. The program will ask the user how much he or she wants to donate to charity, thirsty toads in the Sahara (Manny's Wallet). If the unsuspecting victim wants to donate less than €50, the program will ask again. The program will continue to ask for an amount until the user has agreed to donate €50 or more, after which Mobster Manny will show up to collect the money.
Write this malicious program, but make sure it does not fall in the wrong hands.

Example An example of a correct execution of the program is shown below:

```
Enter the amount you want to donate:
0
Enter the amount you want to donate:
10
Enter the amount you want to donate:
52
Thank you very much for your contribution of 52.00 euro.
```

7. Alphabet

Write a program that prints the alphabet on a single line. Print every character seperated by a space. Do not use the ascii_lowercase constant, or other constants from the string module.

Hint: use the ord() and chr() functions. To make sure Python does not print a newline after each print statement use a comma at the end of the print statement.
You should get the following output:

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

8. Collatz

One of the most renowned unsolved problem is known as the Collatz conjecture. The problem is stated as follows:

> Start out with a random number n.
>
> - if n is even, the next number is $n/2$
> - is n is odd, the next number is $3n + 1$.
>
> This next number is treated exactly as the first. This process is repeated. An example starting with 11: 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 4 2 1 4 2 1 . . .
> Once the sequence has reached 1, the values repeat indefinitely. The conjecture is that every sequence ends with 4 2 1 4 2 1 . . .

This conjecture is probably correct. Using computers all numbers up to $10 * 2^{58}$ have been found to end with this sequence. This problem might seem very simple, but no one has proved the conjecture since Collatz stated it in 1937. There have even been mathmaticians that have spent years of continued study on the conjecture, without success. Fortunately, writing a program that generates the Collatz sequence is a lot less challenging.
Write a program that takes any positive integer and prints the corresponding Collatz sequence. End the sequence when it reaches one.

Hint Use the % (modulo) operator to test whether a number is even or odd.

9. SecondSmallest

Take an unknown number of positive integers as input. Assume that the first number is always smaller than the second, all numbers are unique and the input consists of at least three integers. Print the second smallest integer.

Example

```
10 12 2 5 15
The second smallest number is:5
```

Do not use the method sort(), min() or max().

3 Methods and functions

Abstract

The programs written in the previous module use if-statements and loops. Writing complicated programs with these statements will quickly result in confusing code. Introducing methods and functions to the code can solve this problem. This chapter will provide the neccesary knowledge on how to use these constructs, but more important on how introduce structure to code using these constructs.

Goals

- Familiarize with methods, functions and parameters.
- Use methods and functions to structure programs.

Instructions

- Read the theory about **Methods and functions**. With this knowlegde in mind, make the assignments **NuclearPowerPlant**, **Palindrome 1**, **Palindrome 2**, **Pyramid** and **Pizza**.
- Now make the graded assignment **Geography Grades 1**.

Theory

Methods and functions

The theory about how functions work, what they are used for and how to call them has been explained in the lectures. But as a small reminder: functions and methods are both "self-made commands", however functions are commands in your program and methods are commands in an object. Since objects are not covered in this course, you'll only be making functions. Even though you won't be making methods you will be using methods. For example from the String object you'll be using the method "split()".
A function call is just another statement. The execution of this statement is slightly more complicated than the execution of a normal Python statement; instead of executing a single statement, a whole function, possibly calling other functions, has to be executed. The great thing about using functions is that at the moment that a function is called, it does not matter how the function is executed. The only thing that matters is *what* the function does, and not *how* the function does this.
An example. A program that translates Dutch text into flawless English will most likely feature a piece of code like this:

```
for line in text:
    dutch_sentence = read_sentence(line)
    english_sentence = translate_sentence(dutch_sentence)
    write_sentence(english_sentence)

    # etc
```

It is very unlikely that someone will doubt the correct execution of this piece of code. Whilst writing a part of the program, it is assumed that the functions read_sentence(), translate_sentence() and write_sentence() exist. The way that these functions work, does not matter. Without knowing how these functions work, it cán be concluded that this piece of code is correct.
The function readSentence() is not that difficult to write. A sketch of this function:

```
def read_sentence(line):
    # Returns a Dutch sentence.

    sentence = ''
    for word in line:
        sentence += read_word(word) + " "

    return sentence
```

Functions are used to split the program into smaller parts, that have a clear and defined use. This can all be done without knowing how other functions do what they are supposed to do. When writing a part of the program, it is important not to be distracted by a detailed implementation somewhere else in the program. This also works the other way around. When writing a function, it is not important what it is used for in the piece of program that calls it. The only thing that matters, is that the function does exactly what it is supposed to do according to the function's name.
Small pieces of code can easily be understood and can be checked easily whether they do what they are supposed to do in the right way.

Rule of Thumb
A function consists of no more than 15 lines.

An elaborate example is provided below. Study the structured way of parsing the input. This technique will be extensively used in the GeographyGrades assignments.

Example The world-renowned Swiss astrologer Professor Hatzelklatzer has discovered a new, very rare disease. This disease will be known to the world as the Hatzelklatzer-syndrom. The disease is charactarized by seizures lasting for approximately one hour.
Professor Hatzelklatzer suspects that these seizures happen more often in odd months. He has asked his assistant to write a program that will test this hypothesis. Professor Hatzelklatzer has observed a group of test subjects. A file contains all the reported seizures. Each line indicates the date on which one of the test subjects suffered from a seizure. The input is structured in the following way:

```
12 01 2005
28 01 2005
etc...
```

The following example program will parse this input.

```
import sys

''' Assignment: Hatzelklatzer
    Created on 29 sep. 1997
    @author: Heinz Humpelstrumpf '''

STARTING_YEAR = 1950
FINAL_YEAR = 2050

def print_percentage_of_cases(percentage) :
    print "The percentage of illnesses that match " + \
            "the hypothesis is: %.2f" % percentage

# Reads a number from the input string. If the number is not
# in range the program will print an error message and
# terminates. Otherwise, the number is returned.
def read_in_range(input_string, start, end) :
    result = int(input_string)
    if result < start or result > end :
        print "ERROR: %d is not in range (%d, %d)" % \
                (result, start, end)
        sys.exit(1)

    return result


def odd_month(input_string) :
        date = input_string.split()

        # the day is read, but not saved
        read_in_range(date[0], 1, 31)

        # the month is read and saved in the variable "month"
        month = read_in_range(date[1], 1, 12)

        # the year is read, but not saved
        read_in_range(date[2], STARTING_YEAR, FINAL_YEAR)

        return month % 2 != 0


'''Start Program'''
total_number_of_seizures = 0
number_in_odd_months = 0

lines = open('input.txt').readlines()
for line in lines :
    if odd_month(line) :
        number_in_odd_months += 1

    total_number_of_seizures += 1

percentage = (float(number_in_odd_months) /
                total_number_of_seizures) * 100.0

print_percentage_of_cases(percentage)
```

☞ You should now have sufficient knowledge to make the assignments **NuclearPowerPlant**, **Palindrome 1**, **Palindrome 2**, **Pyramid** and **Pizza**.

Parsing input

Using methods from the file and string modules, structured input can be read. Until now, all input was quite simple; a number was read by reading an entire line and converting it to an integer afterwards. Using the aforementioned methods much more sophisticated input can be read.

Reading strings As seen before, using the method readline() an entire line can be read. Using the method read() the entire input can be read. The input in the rest of this practical will mostly consist of multiple lines of the same input. Calling the method readlines() will read all lines on the standard input and return them as a list. This list can subsequently be iterated through with a for-statement.

Strings can be read from a file by using the open() function. This function takes as argument the path of the file that will be opened. For example, a file called example.txt can be opened by calling open('example.txt') if the file is in the same directory as where the code is ran from. In PyCharm this can be done by placing the input file in the same folder as the Python file.

Reading from a string When a string contains more than one word or number, it is often required to parse it further. This can be done using the split() method from the string module.

Using split() without an argument will split the string on any *whitespace* string. These include spaces, tabs and newlines. For example:

```
string = "a,b,c,d 2#4#6#8"
strings = string.split()

letters = strings[0]
```

```
numbers = strings[1]
```

Reading using delimiters Strings can not only be split on whitespace, but on any string. To do this, an argument has to be supplied to the split() method. Splitting the string "2#4#6#8" with split() would just return a list containing a single element, "2#4#6#8". To split this string into four separate strings, use split("#"). For example:

```
string = "2#4#6#8"
numbers = string.split("#")

# read all the numbers, and print the sum of all the numbers.
result = 0
for number in numbers :
    result += int(number)

print result
```

Hint: Split has an optional argument maxsplit which limits the amount of splits to this number. This is useful if one only wants to remove the first line of a file for example.

Example Input is often structured, this means that the input is made up of different parts, often themselves divided in separate parts. Such input can be read in a structured way by first reading the large parts and forwarding these parts to a different function that will read the sub-parts.

The example uses the following structured input:

```
Melissa White-Admiral Nelsonway;12;2345 AP;Seaty
Richard of Hughes-Green Lawn;1;2342 SS;Seaty
Godwyn Large-Calferstreet;101;2341 NG;Seaty
Petronella Diesel-The Mall;1102;2342 MW;Seaty
etc...
```

The input is read from a file called input.txt by calling open('input.txt').readlines() and is made up of an unknown number of students. Every line states the name and address of a single student. The name is separated from the address by a '-'. The address consists of a street, house number, postal code and city. The components are separated by a ';'.

One of the most important skills is to recognize such structures. Luckily this is not that hard. Study the example and the explanation provided below. The program will read the input defined above and print the addresses in format suitable for letters.

```
import sys


''' Assignment: Addresses
    Created on 6 aug. 2012
    @author: Jan Stienstra '''


def print_address(input_address) :
    address = input_address.split(";")

    street = address[0]
    house_number = int(address[1])
    postal_code = address[2]
    city = address[3]

    print "%s %d\n%s %s" %(street, house_number,
                   postal_code, city)

def print_student(student) :
    student_details = student.split("-")

    full_name = student_details[0]
    address = student_details[1]

    print_address(address)


'''Start Program'''
students = open('input.txt').readlines()

for student in students :
    print_student(student)
```

The program is very comprehensible, even without comments. The program has three functions, each reading a different aspect of the input:

- The start of the program splits the input into separate students:

  ```
  Melissa White-Admiral Nelsonway;12;2345 AP;Seaty
  Richard of Hughes-Green Lawn;1;2342 SS;Seaty
  Godwyn Large-Calferstreet;101;2341 NG;Seaty
  Petronella Diesel-The Mall;1102;2342 MW;Seaty
  ```
  etc...

- print_student(student) then reads the name and address separately and forwards the address to the printAddress function.

  $\boxed{\texttt{Melissa White}}$-$\boxed{\texttt{Admiral Nelsonway;12;2345 AP;Seaty}}$

- print_address(input_address) then reads every component, and prints them in a desired format:

  $\boxed{\texttt{Admiral Nelsonway}}$;$\boxed{\texttt{12}}$;$\boxed{\texttt{2345AP}}$;$\boxed{\texttt{Seaty}}$

Assignments

1. NuclearPowerPlant

☞ Before starting this assignment, read the theory about **Methods and functions**.

The nuclear powerplant at Threeyedfish will automatically run a program to print a warning message when the reactor core becomes unstable. The warning message reads:

```
NUCLEAR CORE UNSTABLE!!!
Quarantine is in effect.
Surrounding hamlets will be evacuated.
Anti-radiationsuits and iodine pills are mandatory.
```

Since the message contains crucial information, it should be printed three times. To do this, write a function that prints this message. This function has to be used three times.

2. Palindrome 1

Write a program that will print the the following string:

```
    abcdefghijklmnopqrstuvwxyzyxwvutsrqponmlkjihgfedcba
```

It is not allowed to do this hardcoded.

Hint: This line consists of three parts:

- a to y
- z
- y to a

3. Palindrome 2

This assignment takes of where Palindrome 1 has finished. Make a copy of Palindrome 1 and edit the code so that the program will:

- read a letter from standard input
- print the string from Palindrome 1 up to this letter

For example, if the letter was c, the output would be:

```
    abcba
```

4. Pyramid

Write a program that prints a pyramid made of letters in the middle of the screen. Use functions with parameters for this assignment. The example shows the expected output, a pyramid of 15 levels. It can be assumed that the screen width is 80 characters.

Example

```
              a
             aba
            abcba
           abcdcba
          abcdedcba
         abcdefedbca
        abcdefgfedcba
       abcdefghgfedcba
      abcdefghihgfedcba
     abcdefghijihgfedcba
    abcdefghijkjihgfedcba
   abcdefghijklkjihgfedcba
  abcdefghijklmlkjihgfedcba
 abcdefghijklmnmlkjihgfedcba
abcdefghijklmnonmlkjihgfedcba
```

5. Pizza

Mario owns a pizzeria. Mario makes all of his pizzas from 10 different ingredients, using 3 ingredients on each pizza. Mario's cousin Luigi owns a pizzeria as well. Luigi makes all his pizzas from 9 ingredients, using 4 ingredients on each pizza. Mario and Luigi have made a bet: Mario believes that customers can order a larger selection of pizzas in his pizzeria than they can order in Luigi's pizzeria.
Write a program that calculates the number of pizzas Mario and Luigi can make. Use functions for this assignment. Make your own implementation of the factorial() function from the math module. The outcome should look like this:

```
Mario can make 120 pizzas.
Luigi can make 126 pizzas.
Luigi has won the bet.
```

Hint When choosing k items from n possible items, the number of possibilities can be obtained using the following formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Graded assignment

6. Geography Grades 1

In the input file, grades are listed for the geography tests of group 2b. There have been three tests of which the grades will be included in the half-yearly report that is given to the students before the Christmas break.

On each line of the input you can find the name of the student, followed by one or more under scores ('_'). These are succeeded by the grades for the tests, for example:

```
Anne Adema____________6.5 5.5 4.5
Bea de Bruin__________6.7 7.2 7.7
Chris Cohen___________6.8 7.8 7.3
Dirk Dirksen__________1.0 5.0 7.7
```

The lowest grade possible is a 1, the highest a 10. If somebody missed a test, the grade in the list is a 1.

Your assignment is to make the report for the geography course of group 2b, which should look like this:

```
Report for group 2b
Anne Adema has an average grade of 5.5
Bea de Bruin has an average grade of 7.2
Chris Cohen has an average grade of 7.3
Dirk Dirksen has an average grade of 4.6
End of report
```

4 Parsing input

Abstract

A lot of programs depend on some sort of input. In previous chapters only simple input was used. This module will introduce the notion of *structured reading* of *structured input* to parse complex input and write structured programs.

Goals

- Understand the notion of structured reading.
- Write well structured code that reflects the way the input is parsed.

Theory

Layout

A good layout is essential to make comprehensible programs. There are a lot of different layouts that will result in clear programs. There is no single best layout, but it is important to maintain the same layout throughout the whole program. Examples of a good layout can be found in all the examples in the book and in this instruction manual. A couple of rules of thumb:

Rule of Thumb

In for, while, if or elif statements, all code in the body is indented by four spaces, usually the width of one tab.

Rule of Thumb

Functions are separated by at least one blank line. The initialization of variables and assignments are also separated by a blank line. White lines can be added anywhere, if this increases clarity.

Novice programmers often lack enough space in their programs. If the code does reach the end of the page, by indenting twelve times, the code is probably too complicated. It will have to be simplified by introducing new methods and functions. The TAB key is useful for indenting pieces of code.
The layout of an if-statement is one of the most difficult statements to define. The layout is greatly influenced by the code following the statement. These examples show possible layouts:

```
if boolean expression:
    statement
```

```
if boolean expression:
    statement
else:
    statement
```

```
if boolean expression: short statement
```

```
if boolean expression: short statement
else : short else-statement
```

```
if boolean expression:
    a lot of statements
    ...
else: # opposite of the boolean expression
    statements
    ...
```

```
if boolean expression 1:
    statement 1
elif boolean expression 2:
    statement 2
elif boolean expression 3:
    statement 3
else: # explanation on the remaining cases
    statement 4
```

The ternary operator

When a choice between two cases can be made based on a short expression, the following statement can be used:

value, if expression is true `if` *expression* `else` *value, if expression is false*

This way, the following piece of code:

```
if a < b :
    minimum = a
else :
    minimum = b
```

can be shortened to:

```
minimum = a if a < b else b
```

Comments

"Comments make sure that a program is readable. Everyone knows this, a truism. In the past, when no one could program, someone would sometimes write a program without comments. This is considered by many to be old-fashioned, offensive even. Comments are the programmer's cure-all. When a comment is added to all cleverly thought over pieces of code, nothing can go wrong."

Wrong!

Comments are not meant to explain dodgy programs to a reader. A program that can be understood without comments, is better than a program that cannot be understood without comments. This ís ofcourse a truism. Comments may never replace clear programming.
Comments should not be written wherever possible, but on those occasions where they are neccesary. An example of unneccesary commenting:

```
# the sum of all values is assigned to result.
result = 0
for input in inputs :
    result += input
```

It can be assumed the reader can understand Python. A clear piece of code does not need additional explanation.
There are some cases in which it is advisable to add comments in the middle of a function (for example, the previous if-statements). But usually comments are placed at the top of the function. These comments are usually placed to explain a complex function, describing:

- what the function does
- (if neccessary) how it does this
- (if neccessary) how the function changes external values. If, for example, a global variable is changed within the function, it might be useful to write this in a comment.

A well written program contains a lot of functions without any comments. Usually, the name of a function will indicate precisely what will happen and the code will be readable. For example:

```
def print_row(row) :
```

does not require explanation telling the reader that a row is printed. However, the function

```
# Sorts the list using "rapidsort";
# see Instruction Manual.

def sort(row) :
```

does require this kind of explanation. The execution of a sorting algorithm is not trivial. This can be solved in two ways: explaining the algorithm within the program, or reference another document describing the precise execution of this piece of code. In the latter case, the code has to exactly match the description of course.

✎ **Rule of Thumb**

If the *name* of a function explains *what* it does and it is trivial *how* it does this, no comments are neccessary.
If one or both of the prerequisites are not met, a comment is needed.
There are very little or no comments within a function.

Assignments

1. Geography Grades 2

☞ Before starting this assignment, read the theory about **Parsing input**.

Make a copy of your program for the problem Geography Grades 1 and change the code in such a way that your program no longer prints the average grade, but the final grade.

The final grade is calculated by rounding the average grade to the nearest multiple of a half. So, for example, a 7.2 becomes a 7.0 and 7.3 becomes a 7.5. If this calculation results in a 5.5, the final grade becomes a 6.0

Your assignment is to make the report for the geography course of group 2b, that, with the same example input as for the problem Geography Grades 1, should look like this:

```
Report for group 2b
Anne Adema has a final grade of 6.0
Bea de Bruin has a final grade of 7.0
Chris Cohen has a final grade of 7.5
Dirk Dirksen has a final grade of 4.5
End of report
```

2. Geography Grades 3

Make a copy of your program for the problem Geography Grades 2 and change the code in such a way that your program can process multiple groups.

These groups are on the input separated by '=\n'. Every group starts with a first line that contains the name of the group and the lines after contain the information about the students in the same way as is specified for the problem Geography Grades 1.

With the input

```
1b
Erik Eriksen__________4.3 4.9 6.7
Frans Franssen________5.8 6.9 8.0
=
2b
Anne Adema____________6.5 5.5 4.5
Bea de Bruin__________6.7 7.2 7.7
Chris Cohen___________6.8 7.8 7.3
Dirk Dirksen__________1.0 5.0 7.7
```

The output should be:

```
Report for group 1b
Erik Eriksen has a final grade of 6.0
Frans Franssen has a final grade of 7.0
End of report

Report for group 2b
Anne Adema has a final grade of 6.0
Bea de Bruin has a final grade of 7.0
Chris Cohen has a final grade of 7.5
Dirk Dirksen has a final grade of 4.5
End of report
```

Graded assignment

3. Administration

For the end of year administration of Programming for History of Arts students you are to write a program that can do 2 things:

1. calculate a final grade
2. print a small graph of similarity scores and, if applicable, list the students under investigation

The input is structured as follows:

```
Piet van Gogh___5 6 7 4 5 6
5=20=22=10=2=0=0=1=0=1;Vincent Appel,Johannes Mondriaan
Karel van Rijn___7 8 6 6
2=30=15=8=4=3=2=0=0=0;
```

The first line should be interpreted as follows:

```
<Name of the student><one or more underscores><one or more grades separated by spaces>
```

You have to calculate the final grade of the student. All grades have the same weight. The final grade is rounded as follows:

- a grade that is >= 5.5 AND < 6 should be noted as a "6-"
- otherwise a grade will be rounded to the nearest half

The second line should be interpreted as follows:

```
<10 numbers separated by ’=’>;<zero or more names separated by ’,’>
```

The first 10 numbers are the similarity scores. These scores represent the number of programs matching a certain percentage of the current program in steps of 10%. This means the first numbers indicates the matches from 1%-10% and the last number indicates the matches from 91%-100%.

Since this is not very readable, the professor would like a simple graph according to these rules:

- if there are zero matches, display an underscore: _
- if there are less than 20 matches, display a minus sign: $-$
- if there are 20 or more matches, display a caret: $\wedge$

The names of the students after the semicolon are the names of the students with matches in the final 3 categories. The names of these students should be printed under the graph. If there are no matches, the program should print "No matches found".

The output for the aforementioned input should be:

```
Piet van Gogh has an average of 6-
        -^^--__-_-
        Vincent Appel
        Johannes Mondriaan
Karel van Rijn has an average of 7.0
        -^-----___
        No matches found
```

5 Two Final Graded Assignments

Graded assignment

1. House market

Big Data is the new hype. You are given two datasets. The dataset "houses_sold" contains information about houses that were sold in the past year. The dataset "houses_for_sale" contains information about all houses that are for sale at the moment.

Both datasets contain an unknown number of lines. Each line looks as follows:

<size>;<price><end of line>

The size of the house is a float value (in m^2). The price of the house is an int value (in euro's).

Albeit a bit naive, we assume that there is a linear correlation between the size of the house and the price. This will enable you to use the dataset "houses_sold", to build a linear model of the house market, with independent variable 'size' and dependent variable 'price'.
Information about how to calculate the coefficients of the line fitting the data with the least square error can be found in `http://en.wikipedia.org/wiki/Simple_linear_regression`.

Write a program that will:

1. Read the information about all the houses that were sold in the last year. This information can be read from the file "houses_sold".
2. Calculate the coefficients of the straight line that fits the read data best by performing simple linear regression using the method of least squares of the course Quantitative Research Methods I.

3. Give a graphical representation of the read data and the calculated line.

    - Plot the line + a dot for every house sold.

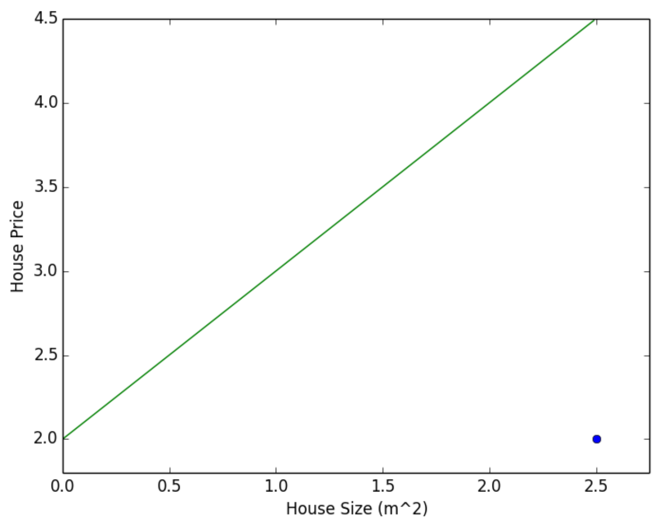
      The library ipy_lib contains a `HouseMarketUserInterface()` method that return a user interface that can be used to do the plotting using the methods `plot_dot()`, `plot_line()` and `show()`.

      Look in the documentation of the library ipy_lib to get information about `plot_dot()`, `plot_line()` and `show()`.

      Example:
      Plot the dot (2.5, 2.0) and the line y=ax+b when the coefficients have values a=2 and b=1.

```
import ipy_lib
ui = ipy_lib.HouseMarketUserInterface()

a = 2
b = 1
ui.plot_dot(2.5, 2.0, 'b')
ui.plot_line(a, b)
ui.show()
```



4. Read the information about all the houses for sale at the moment. This information can be read from the file " houses_for_sale".

5. Put for each house for sale a dot in the graphical representation mentioned in point 3. Use a different colour than for the dots that represent sold houses.

6. Print a line of text that tells whether the houses for sale are expensive or affordable. The houses for sale are expensive if the majority of them are above the line in the graphical representation of the data. The houses for sale are affordable if the majority is below that same line.

2. Second Assignment

The last friday of the practical you will do the Second Assignment. From 9:00 till 13:00 you will be making small programs on the level of module 2 and 3. You will have to do this on your own and can't receive help from the TA's or your peers. Internet won't be accessible but the one site used for the assignments. In the end you will hand in all these programms with the hand-in button.
You will receive the grade for these programs automatically the same Friday afternoon.

There will be an 'exercise round' one week prior to the assignment itself.

If need be, there is a re-take one week after the Second Assignment has been made.