
Instruction Manual Introduction to Programming in Python

Jan Stienstra

First Edition

Published 2012

Contents

0 Syllabus	5
1 Editing, Compiling and Executing	7
Goals	7
Introduction to Eclipse	8
Installation and starting of Eclipse	8
Arranging files	9
Compiling and executing programs	10
Printing a program	10
Submitting assignments	10
Trial submission	11
2 If statements and loops	13
Goals	13
Instructions	13
Theory	14
Efficient programming	14
Constants	15
Identifiers	16
Conventions	18
Self test	18
If-statements	21
Assignments	22
1 VAT	22
2 Plumber 1	22
3 Plumber 2	22
4 Othello 1	23
5 Electronics	23
6 Othello 2	23
7 Manny	24
8 Alphabet	24
9 Collatz	24

10	SecondSmallest	25
3	Methods and functions	26
	Goals	26
	Instructions	26
	Theory	27
	Methods and functions	27
	Importing the IPy library	29
	Assignments	31
	1 NuclearPowerPlant	31
	2 Characters 1	31
	3 Characters 2	31
	4 Pyramid	31
	5 Pizza	32
	Graded assignment	33
	6 Replay 1	33
4	Parsing input	35
	Goals	35
	Theory	36
	Layout	36
	Comments	37
	Parsing input	38
	Assignments	41
	1 Replay 2	41
	2 Replay 3	41
	Graded assignment	42
	3 Statistics	42
5	Modules, classes and lists	43
	Goals	43
	Theory	44
	Modules and classes	44
	Assignments	48
	1 List	48
	2 BodyMassIndex	48
	3 BodyMassIndex2	48
	Graded assignment	50
	4 Pirate	50
6	Events and animations	51
	Goals	51
	Theory	52
	Events	52
	Animations	53
	Stepwise refinement	53
	Assignments	57
	1 Events	57
	2 Animation	57
	Graded assignment	58

3	Snake	58
7	Bonus	60
	Graded assignment	60
1	Life	60

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



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 The book that will be used during this course is *Learning with Python*, 2nd Edition, by Jeffrey Elkner, Allen B. Downey, and Chris Meyers. 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.

Chapters The course material is divided into six chapters. These chapters contain additional theory and assignments and will be used as an instruction manual during the lab sessions. Every week a single chapter will be treated. Theory treated during the lectures will be repeated as little as possible in the chapters. The chapters will however feature notes on programming style. Each week, a chapter will be made available on Blackboard. This course also features a bonus assignment. The bonus assignment can be found in chapter 7. If you pass the bonus assignment, half a point will be added to your lab grade.

Assignments Every chapter consists of theory and assignments. Only the last assignment of a chapter will be graded. This does not mean that the other assignments are less important. All the assignments in a chapter will train essential skills needed to successfully complete the graded assignment. *Every* assignment has to be approved by the teaching assistant 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.

Deadlines and assessment

Assignments need to be turned in on paper, so written feedback can be provided by the teaching assistant. Graded assignments have to be submitted to Blackboard each week as well.

The teaching assistant will only assess an assignment if all previous assignments have been approved. When the quality of an assignment is not sufficient, it has to be corrected according to the provided feedback and turned in again. Graded assignments can only be turned in once. 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 Starting with chapter 3, the last assignment of a chapter will be graded. **Deadlines will be posted on BlackBoard and will always be on friday 23:59.** Late submission will be accepted, but a point is deducted for every day the assignment is late. The grades are weighted in the following way:

Chapter	Graded Assignment	Weight
1	HelloWorld2	slipday
2	All	slipday
3	Replay1	1x
4	Statistics	1x
5	Pirate	2x
6	Snake	3x
7	Life	0,5 bonus

Slipdays The first two chapters do not include graded assignments. There are regular assignments to be made, so there is also a deadline in the first two weeks. You can earn a *slipday* if all the assignments have been approved before the deadline; you can earn at most 2 slipdays. Slipdays can be used in the following weeks. An assignment can be turned in a day late, without point deduction, using a slipday. Two slipdays can be used to submit a single assignment two days late, or two assignments one day late.

Final Grade You have passed the lab when:

1. all assignments have been approved
2. every graded assignment has been graded
3. the average grade of all the graded assignments is ($\geq 5, 5$).

You passed the Introduction to Programming in Python course if both the exam and lab are passed; both $\geq 5, 5$. The final grade for this course is calculated using the following formula: $F = MAX(E, (2 * E + P)/3)$, with F = Final Grade, E = Exam and P = Practical.

Editing, Compiling and Executing

Abstract

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

Goals

- Use Eclipse to open, edit, save and organize Python-files
- Execute Python files
- Print source code

Introduction to Eclipse

Installation and starting of Eclipse

VU Eclipse has already been installed on all the computers that will be used during the lab sessions. Using Windows, it can be started from the Start Menu. Using Linux, it can be started by typing *eclipse* in a terminal window.

At home, using Windows

To download Eclipse, browse to <http://www.eclipse.org/downloads/> and download “Eclipse IDE for Java Developers”.

After installing Eclipse, it needs to be configured to be able to use it for Python programming.

1. Install Python. Download the Windows x86 MSI Installer (2.7.3) from Python.org:
<http://www.python.org/getit/releases/2.7.3/>.
2. Install the Python Imaging Library (PIL) 1.1.7 for Python 2.7. It can be downloaded at <http://www.pythonware.com/products/pil/>.
3. Install the PyDev plugin for Eclipse.
 - (a) In Eclipse go to Help → Install new software...
 - (b) Click Add...
 - (c) Use PyDev as name and <http://pydev.org/updates/> as location.
 - (d) Select PyDev and click Next.
 - (e) On the next screen, click Finish.
4. Navigate to Window → Preferences → PyDev → Interpreter - Python.
5. Click on New... → Browse and select python.exe from the installation directory.
6. Click OK in the next two screens. Eclipse is now configured to be used with Python.

At home, using Linux

To install Eclipse, open a terminal window and type: *sudo apt-get install eclipse*.

After installing Eclipse, it needs to be configured to be able to use it for Python programming.

1. Install Python. Open a terminal window and type:
sudo apt-get install python2.7.
2. Install the Python Imaging Library (PIL). Open a terminal window and type: *sudo apt-get install python-imaging-tk*.
3. Install the PyDev plugin for Eclipse.
 - (a) In Eclipse go to Help → Install new software...
 - (b) Click Add...

- (c) Use PyDev as name and <http://pydev.org/updates/> as location.
 - (d) Select PyDev and click Next.
 - (e) On the next screen, click Finish.
4. Navigate to Window → Preferences → PyDev → Interpreter - Python.
 5. Click on New... → Browse and select `\usr\bin\python2.7` from the installation directory.¹
 6. Click OK in the next two screens. Eclipse is now configured to be used with Python.

At home, Mac OS X **Not supported**

Selecting a workspace The *workspace* is a folder in which all created files are saved. The workspace only has to be defined once. When Eclipse is started for the first time, it will automatically prompt for a location for the workspace. If, for whatever reason, this does not happen, it can be set manually: File → Switch Workspace. The home directory is usually a good place for the workspace `/home/vunet-id/workspace`.

Arranging files

After selecting the workspace, a welcome screen will be shown. Click on Workbench. On the left of the current window, the Package Explorer is shown. This frame will hold all files, sorted on Project. Every Project consists of Source Folders, whilst Source Folders consist of Packages. This may sound complicated on first glance, but the next few steps will explain everything:

1. Create a new **PyDev Project**. To do this right-click in the Package Explorer and select New → Project. Select **PyDev Project** and click Next.
2. The name of this project will be **Introduction to Programming**. Check the radio button for “Don’t configure PYTHONPATH (to be done manually later on)” and click Finish.
3. Create a new **Source Folder**. Logically related programs will be combined into one **Source Folder**. In this course, every chapter will have its own **Source Folder**. Right-click on the project in the Package Explorer and select New → Source Folder. Name it **Chapter1** and click Finish.
4. Every assignment will have its own **Package**. Right-click on the **Source Folder** that was just created and select New → Package. The name of the **Package** will be the same as the name of the assignment.
5. An initial Python file will automatically be created.

¹It might be necessary to execute Eclipse with elevated privileges. To do this, start Eclipse by typing `sudo eclipse` in a terminal window.

Compiling and executing programs

The file `__init__.py` in its current form is called a *skeleton*; an empty program, that does nothing. As a start, copy the following example and rename the file to `hello_world`:

```
print "Hello World"
```

Programs are compiled automatically in Eclipse. Make sure that the file is saved (Ctrl+S). Right-click on the file to be executed and select Run As → Python Run. To execute the same program again, use Ctrl+F11. 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.

Printing a program

Printing using Windows To print a program, select File → Print or use Ctrl+P. Print two pages on one page.

Printing using Linux Printing using Eclipse is not possible whilst using Linux on VU machines. To print a program using Linux, a special Eclipse command needs to be set.

- Select Run → External Tools → External Tools Configuration
- Select Program and click on **New**
- Set Name to **Print**
- Set Main→Location to: `/usr/local/bin/ipr`
- Select Main→Variables... in the Working Directory and select **folder prompt**
- Set Main→Arguments to: `*.py`
- Click on Apply

To print a program, simply select Run → External Tools → Print. A folder prompt will appear. Navigate to Eclipse Workspace → Introduction to Programming. Select the correct chapter and select the folder with the same name as the program. Click on OK. The program will now be printed.

Submitting assignments

A graded assignment needs to be submitted to BlackBoard. 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 **Package** in the Package Explorer to be exported and select Export. Select General → Archive File and click Next. Click on Browse and navigate to the folder where the archive file should be stored. Make sure the name of the

file starts with the name of assignment and is followed by your VUnet-id separated by a hyphen. For example: pirate-rhg600.zip.

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

2. Submit the .zip-file to Blackboard. Login to Blackboard, browse to this course and select Submit. Click on the assignment you want to submit. On the screen that appears, select Attach File → Browse My Computer and select the .zip-file. If necessary, add comments and click on Submit. Wait for Blackboard to confirm the submission and click on Finish. To ensure that the submission has been received, go to the Grade Center and verify whether there is a green exclamation mark in the cell for the current assignment.

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	replay1-rhg600.zip
4	statistics-rhg600.zip
5	pirate-rhg600.zip
6	snake-rhg600.zip
7	life-rhg600.zip

Note that 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 module hello_world2 and copy hello_world to hello_world2.

Hint: Copying files using Eclipse can be done easily using the refactor functionality provided by Eclipse. For more information about refactoring, visit the [Eclipse website](#).

Edit the module 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? Print the program, hand-in the print to your Teaching Assistent and submit the program to Blackboard.

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 print and submit programs. These are essential skills required during the rest of this course.

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

- Study your lecture note 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 worthwhile. 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 addresses 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 re-compile 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 cannot only ease the maintenance of a program, but can increase the comprehensibility of the code as well. When an constant, like LABEL_HEIGHT, is used, it is immediately clear what this number represents instead of only knowing its numerical value. This property gives constants an added value. Therefore, it is advisable to use constants in programs, even if the value of the constant will never change.



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

user_input = 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 1**, **Plumber 2** 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 *numberOfPatients*. 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:

```
outwardTravelTime # in minutes
returnTravelTime  # 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 *quadraticCoefficient*, *linearCoefficient* and *constantTerm*. 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 = 1
for i in xrange(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 separated 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.

```
1  ''' Assignment: MileInKilometer
2      Created on 6 aug. 2012
3      @author: Jan Stienstra '''
4
5
6  PASS_MINIMUM = 5.5
7
8  grade = int(raw_input("Enter a grade: "))
9
10 if grade >= PASS_MINIMUM :
11     print "The grade, %0.2f, is a pass.\n" % grade
12 else :
13     print "The grade, %0.2f, is not a pass.\n" % 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** and **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 1

The employees at plumbers 'The Maverick Monkey' are notorious bad mathematicians. 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 always rounded. Plumbing laws fix the call-out cost at €16,00.

Write a program that calculates the cost of a repair. Take the hourly wages and number of billable hours as input for this program.

Example A plumber earning €31,50 an hour working for 5 hours should get the following output.

```
Enter the hourly wages: 31.50
Enter the number of billable hours: 5
The total cost of this repair is: 173.50 euro
```

3. Plumber 2

After careful assessment of the new program it turns out that the employees of 'The Maverick Monkey' are as bad in rounding numbers as they are in calculations. Therefore the boss decides to alter the program in such a way that the program only needs the number of hours an employee has actually worked. The program will determine the number of billable hours based on this input. Make a copy of the previous assignment and edit the code to implement this new feature. The number of billable hours is the number of hours worked rounded to an integer.

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
```

Hint: How can a number be rounded to the nearest integer? Do not use the `round()` function from the `math` module.

¹Examples will have input printed in italics.

4. Othello 1

During this course there will be multiple assignments concerning the game of Othello, also known as Reversi. More about Othello can be found at:

<http://www.yourturnmyturn.com/rules/reversi.php>.

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%
```

5. Electronics

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

The electronics 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.

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
```


6. Othello 2

During a game of Othello the time a player spends thinking about his moves is recorded. Write a program that takes the total 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 does not require more than a thousand milliseconds to contemplate its moves.

Example

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

7. Manny

 The following four assignments use loops. Use the right loop for the right assignment; use the for statement once and the while statement thrice.

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.
```

8. 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
```

9. 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 $n/2$
- if 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 mathematicians 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.

10. 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
```

To denote the end of the input, press enter followed by Ctrl-Z (Windows) or Ctrl-D (Linux).

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 necessary 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 knowledge in mind, make the assignments **NuclearPowerPlant**, **Characters 1**, **Characters 2**, **Pyramid** and **Pizza**.
- Make sure to import the library `libUI` into Eclipse as described in **Importing the IPy library**. Now make the graded assignment **Replay 1**.

Theory

Methods and functions

The theory about how methods work, what they are used for and how to call them has been explained in the lectures. A method 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 method, possibly calling other methods, has to be executed. The great thing about using methods is that at the moment that a method is called, it does not matter how the method is executed. The only thing that matters is *what* the method does, and not *how* the method 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:
    dutchSentence = readSentence(line)
    englishSentence = translateSentence(dutchSentence)
    writeSentence(englishSentence)

# 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 methods `readSentence()`, `translateSentence()` and `writeSentence()` exist. The way that these methods work, does not matter. Without knowing how these methods work, it can be concluded that this piece of code is correct.

The method `readSentence()` is not that difficult to write. A sketch of this method:

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

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

    return sentence
```

Methods are used to split the program in smaller parts, that have a clear and defined function. This can all be done without knowing how other methods 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 method, 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 method does exactly what it is supposed to do according to the method 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 method 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 Replay 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 characterized 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.

```
1  import sys
2
3
4  ''' Assignment: Hatzelklatzer
5      Created on 29 sep. 1997
6      @author: Heinz Humpelstrumpf '''
7
8
9  STARTING_YEAR = 1950
10 FINAL_YEAR = 2050
11
12 def print_percentage_of_cases(percentage) :
13     print "The percentage of illnesses that match " + \
14         "the hypothesis is: %.2f\n", percentage
15
16 # Reads a number from the input string. If the number is not
17 # in range the program will print an error message and
18 # terminates. Otherwise, the number is returned.
19 def read_in_range(input_string, start, end) :
20     result = int(input_string)
21     if result < start or result > end :
22         print "ERROR: %d is not in range (%d, %d)\n" % \
23             (result, start, end)
24         sys.exit(1)
25
26     return result
27
28
29 def odd_month(input_string) :
30     date = input_string.split()
31
32     # the day is read, but not saved
33     read_in_range(date[0], 1, 31)
34
35     # the month is read and saved in the variable "month"
36     month = read_in_range(date[1], 1, 12)
37
38     # the year is read, but not saved
```


```

39     read_in_range(date[2], STARTING_YEAR, FINAL_YEAR)
40
41     return month % 2 != 0
42
43
44     '''Start Program'''
45     total_number_of_seizures = 0
46     number_in_odd_months = 0
47
48     lines = sys.stdin.readlines()
49     for line in lines :
50         if odd_month(line) :
51             number_in_odd_months += 1
52
53         total_number_of_seizures += 1
54
55     percentage = (float(number_in_odd_months) /
56                  total_number_of_seizures) * 100.0
57
58     print_percentage_of_cases(percentage)

```

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

Importing the IPy library

 The **Replay 1** assignment is the first assignment that requires the `ipy_lib`. Complete all previous assignments before starting on this assignment.

Some assignments in the remainder of this manual require the IPy library. The IPy library is a collection of classes and methods that enables the use of a graphical interface instead of the console as input and output. Such a collection of classes, providing additional features, is called a library.

To use this library, the Python compiler and the Python interpreter have to know where to find the UserInterface-library. Right-click on “Introduction to Programming” in the Package Explorer and select Properties → PyDev - PYTHON-PATH. Under External Libraries click on Add source folder. Select the folder `/home/ip/python1`. Click OK.

Import-statements can now enable the use of specific elements of the UI-library. For example, if the program requires the `SnakeUserInterface` from the `ipy_lib`, add the following line to the top of the program:

```
from ipy_lib import SnakeUserInterface
```

Selecting input using the IPy library Using the `ipy_lib` library, Eclipse can use files as input instead of the *standard input*, the keyboard. To select a file as input, add the following statement before any other statement relating to the input:

```
file_input()
```

When a program is executed that includes the above mentioned statement, the program will open a browser to select the input file. Browse to the location

¹On Windows or non-VU machines, the `ipy_lib` needs to be downloaded from Blackboard.

of the file and press Enter. The entire contents of the file will be returned in a string.

For more information on the possibilities of the IPy library, visit www.few.vu.nl/~ip/pythondoc/.

Using the IPy library on laptops Using the IPy library on a laptop with a small screen resolution may cause a part of the user interface to disappear off screen. To prevent this, supply an additional third argument to the constructor:

```
ui = SnakeUserInterface(width, height, scale)
```

for example:

```
WIDTH  = 40  
HEIGHT = 40  
SCALE  = 0.5
```

```
ui = SnakeUserInterface(WIDTH, HEIGHT, SCALE)
```

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. Write a program that prints this message three times using a single method. Do not use duplicate code. Separate every warning message by a blank line.

2. Characters 1

Write a program that will print the the following string:

```
abcdefghijklmnopqrstuvwxyzxwvutsrqponmlkjihgfedcba
```

Hint: This line consists of three parts:

- a to y
- z
- y to a

3. Characters 2

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

- read a letter from standard input
- print the string from Characters 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 methods 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
abdcba
abcdedcba
abcdefedcba
abcdefgfedcba
abcdefghgfedcba
abcdefghihgfedcba
abcdefghijihgfedcba
abcdefghijkljihgfedcba
abcdefghijklkljihgfedcba
abcdefghijklmlkljihgfedcba
abcdefghijklmmlkljihgfedcba
abcdefghijklmnonmlkljihgfedcba

```

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. Do not use the function factorial 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. Replay 1

Starting with this assignment, a lot of assignments will be using the library `ipy_lib` and read input from a file. Instruction on how to do this in Eclipse can be found in [Importing the IPy library](#).

The goal of this assignment is to write a program that can replay a game of Othello. The input file contains all the changes that have been made to the board in every turn. All the changes that are made during a turn are to be visualized on screen.

Read and process one line of the input file at a time, letting the changes take place on the board. After each turn, the game halts as long as it has been instructed to by the input file. When the end of the file is reached, the game has finished.

The game start with four stones in the center of the screen. Two white stones at 'd 4' and 'e 5' and two black stones at 'd 5' and 'e 4'. Make sure that these stones are present before processing any turns.

Input specification A number of games of Othello can be found on Black-board. A piece of input file looks like this:

```
black 1035      move b 3
black 0         move c 3
black 0         move d 3
white 4058      move h 1
white 0         move g 2
black 24234     pass
white 8494      move a 6
etc...
```

The line starts with indicating which player played this turn. The next number indicates the thinking time in milliseconds. After this there is either the word "move" or "pass". Stones are only being placed when a turn is not passed. The coordinate following the word "move" is the stone placed by the player currently playing. All other coordinates by the same player are captured stones.

Specification OthelloReplayUserInterface This assignment uses the `OthelloReplayUserInterface` to show the othello board. After declaring and initializing it, like any other variable, it is ready to use:

```
1 ui = OthelloReplayUserInterface()
```

If all is well, the `OthelloReplayUserInterface` should pop up.

The `OthelloReplayUserInterface` has a number of methods available to the user. All the methods required for this assignment are:

- `wait(milliseconds)`
- `place(x, y, color)`
- `show()`

The class contains the following constants:

- `NUMBER_OF_COLUMNS = 8`

- `NUMBER_OF_ROWS = 8`
- `BLACK = 2`
- `WHITE = 1`
- `EMPTY = 0`

Example The following example shows a piece of code placing a white stone in the top-left corner, a black stone in the bottom-right corner, waiting 5 seconds, changing the white stone in the top-left corner to black and finally deleting the black stone in the bottom-right corner.

```
1 from ipy_lib import OthelloReplayUserInterface
2
3
4 WAITING_TIME = 5000 # in milliseconds
5
6 ui = OthelloReplayUserInterface()
7
8 ui.place(0, 0, ui.WHITE)
9 ui.place(ui.NUMBER_OF_COLUMNS - 1, ui.NUMBER_OF_ROWS - 1, ui.BLACK)
10 ui.show()
11
12 ui.wait(WAITING_TIME)
13
14 ui.place(0, 0, ui.BLACK)
15 ui.place(ui.NUMBER_OF_COLUMNS - 1, ui.NUMBER_OF_ROWS - 1, ui.EMPTY)
16 ui.show()
```

Submitting This assignment needs to be submitted on Blackboard. Instructions on how to do this can be found in module 1.

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

Methods are separated by at least one blank line. The declaration 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 is ofcourse a truism. Comments may never replace clear programming.

Comments should not be written wherever possible, but on those occasions where they are necessary. An example of unnecessary 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 method (for example, the previous if-statements). But usually comments are placed at the top of the method. These comments are usually placed to explain a complex method, describing:

- what the method does

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

A well written program contains a lot of methods without any comments. Usually, the name of a method 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 method

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

Parsing input

Using functions 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 functions much more sophisticated input can be read.

Reading strings As seen before, using the function `readline()` an entire line can be read. Using the function `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 function `readlines()` on `sys.stdin` 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.

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()` function 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()` function. 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 method 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 made up of an unknown number of students and is read with `sys.stdin.readlines()`. 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\n" %(street, house_number,
                             postal_code, city)

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

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

    print_address(address)

'''Start Program'''
students = sys.stdin.readlines()

for student in students :
    printStudent(student)

```

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

- `start()` 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)` reads the name and address separately and forwards the address to the `printAddress` method.

```

Melissa White-Admiral Nelsonway;12;2345 AP;Seaty

```

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

```

Admiral Nelsonway;12;2345AP;Seaty

```

Assignments

1. Replay 2

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

The input of Replay1 was structured like this:

```
black 1035      move b 3
black 0         move c 3
black 0         move d 3
etc...
```

This causes a lot of unnecessary information. Every turn is being divided among multiple lines. There is currently no distinction between a turn and a move. The input has been re-arranged, and now looks like this:

```
black 1035      move b 3 c 3 d 3
white 4058      move h 1 g 2 f 3 e 4
black 24423     pass
white 8494      move a 6
etc...
```

In this arrangement, every line consists of a turn, and every turn consists of multiple coordinates. Make a copy of Replay1 and edit the code such that it can handle this new arrangement. In addition, print the number of pieces that have been conquered during each turn. Use the statusbar of the OthelloReplayUserInterface to print the textual output of the program.

Example

```
black: conquered 2 pieces
white: conquered 3 pieces
black: passed
white: conquered 0 pieces
```

2. Replay 3

For this assignment, multiple games of Othello have been combined into a single file, separated by a '='. This file can be found on Blackboard. Make a copy of Replay2 and edit the code in such a way that it can handle this input file.

Games should be played one after another, waiting five seconds between the end of one game and the start of another. Before starting a new game, make sure to erase the board and status bar.

Graded assignment

3. Statistics

University ‘The Blossoming Brain’ is home to an annual Othello tournament between artificial intelligent programs, made by bèta students. As one can imagine, it is important for each student to know how well his program fared against others. The goal of this assignment is to write a program that provides this information.

A number of files with statistics about the matches can be found on Blackboard. These files are structured in the following way:

```
2000
I
Achraf Belmokadem           2355  22
Richard van Heuven van Staereling 1355  95
Shan Shan                   2315 100
=W
Wartie Zeggen               4311  19
Melissa de Wit              2041  99
Enes Goktas                 1195  74
-
2001
I
etc...
```

The file holds information on how well a program fared against other programs of other students of different academic years. Each academic year is separated by a ‘-’. Each year starts off with a number, declaring which year the statistics concern. After this, the file shows how well the program fared against students of different studies. Each study is separated by a ‘=’. Next, each individual match is declared. First, the name of the student is shown followed by a TAB (‘\t’). The next number shows how long the program had to think against this player. The last number is the percentage of pieces that the program has conquered.

Write a program that makes a bar chart showing how often the program has ended with 0-9%, 10-19%, ..., 90-100% of the pieces.

This assignment uses the `BarChartUserInterface`. The declaration and initialization of the `BarChartUserInterface` is quite similar to declaration and initialization of the `OthelloReplayUserInterface`:

```
numberOfBars = 10
barChart = BarChartUserInterface(numberOfBars)
```

Start with a small program in which the use of the user interface can be tested. For example, make a bar chart of 4 bars and try to raise a number of bars. Next, expand the program in such a way that it can parse the input.

Bonus The current program is working fine for 10 bars. The number of bars, “10”, is a constant. Make sure the program still works well when this constant is changed. Test your program with different numbers of bars, such as 7 and 99.

Modules, classes and lists

Abstract

Classes are used to encapsulate a number of relating methods and variables. In the previous chapters methods and functions were used to structure a program. Using classes, a program can be structured even further. For instance, it is possible to represent real-world objects using classes. This module shows how to use classes.

Goals

- Using lists to save a (un)known number of values.
- Using more classes and modules and recognising when to use extra classes and/or modules.
- Recognizing the right class/module for the right method, variable or constant.

Theory

Modules and classes

A python source file is called a module. Modules can contain a program, methods, a class, or all of these. A class defines an object from the real world. Complex programs almost always require the use of two or more modules or classes. To decide whether a new module should contain a class consider the following:

1. Modules contain several logically connected methods do not become classes. An example of such a module is the math class. This class contains various methods used to perform mathematical operations. All resulting programs from the previous assignments also belong to this set of modules as well.
2. Classes are modules that define an object from the real world. An example of such a class is the class Person. This class could contain data like: name, address, date of birth, etc. In addition, this class could contain a function `age()`, which returns the age of this person by using the date of birth and the current date. Another example of such a class is Circle. This class contains a center and a radius. Methods that are logically connected to these class are for example `surface_area()` and `circumference()`.

In the first four modules, only modules of the first category were required. Assignment from this module onwards will also use classes. An example will show how to make and use such a class. It is good practice to use separate files for different classes.

Example This example will feature a program used by airline 'FlyLo' to calculate the profit of a flight to London. The airline uses four different fares:

1. Toddlers, aged 0 to 4 years old are charged 10% of the regular fare.
2. Children, aged 5 to 12 years old are charged half the regular fare.
3. Adults are charged the regular fare.
4. Elderly people, aged 65 years or more pay an extra 10% on top of the regular fare, as they have more money anyway.

The regular fare for a single ticket to London is €99. A Boeing 747, the largest plane in the airline's fleet, will accommodate 400 passengers.

The airline wants to know what consequences a change in the maximum age of a toddler may have. The airline wants to know what the new profit of a flight will be, and if there is an increase or decrease in the profit.

The program uses a two-line input. The first line contains all the passenger's ages. The second line contains the new maximum toddler age. The program will print the two profits and the difference between the two. It is useful to save all the ages in a row. This way, the row can simply be 'asked' how many passengers will fit each category. This data will allow the program to calculate

the total profit. A row, such as the one proposed above is a good example of a class. The class holds data, the ages of all the passengers and the number of passengers. Apart from the constructor, the class has a method to add an age to the row and a method to calculate how many passengers fit into a certain category.

This row can be implemented using a list. The method `add()` will add a new age to the back of the row. The function `number_in_range(from, to)` calculates the number of passengers that fit this range. The methods `read_age_row()` and `calculate_total_profit()` are self-explanatory.

```
class AgeRow (object):

    def __init__(self) :
        self.age_row = []

    def add(self, age) :
        self.age_row.append(age)

    def number_in_range(self, start, end) :
        # Calculates how many passengers are in the range
        # bounded by start and end

        result = 0

        for age in self.age_row :
            if start <= age and age <= end :
                result += 1

        return result
```

```

import sys
from age_row import AgeRow

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

MAX_TODDLER_AGE = 4    # year
MAX_CHILD_AGE   = 12   # year
MAX_ADULT_AGE   = 64   # year
MAX_AGE         = 135  # year

ADULT_FARE      = 99.0      # euro
TODDLER_FARE    = ADULT_FARE * 0.1 # euro
CHILD_FARE      = ADULT_FARE * 0.5 # euro
ELDERLY_FARE    = ADULT_FARE * 1.1 # euro

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 read_age(input_string) :
    return read_in_range(input_string, 0, MAX_AGE)

def read_age_row(passengers) :
    result = AgeRow()

    for passenger in passengers :
        age = int(passenger)
        result.add(age)

    return result

def calculate_profit(start, end, fare, ageRow) :
    return ageRow.number_in_range(start, end) * fare

def calculate_total_profit(max_toddler_age, age_row) :
    toddler_profit = calculate_profit(0, max_toddler_age, TODDLER_FARE, \
        age_row)

    children_profit = calculate_profit(max_toddler_age + 1, \
        MAX_CHILD_AGE, CHILD_FARE, age_row)

    adult_profit = calculate_profit(MAX_CHILD_AGE + 1, \
        MAX_ADULT_AGE, ADULT_FARE, age_row)

    elderly_profit = calculate_profit(MAX_ADULT_AGE + 1, \

```

```
        MAX_AGE, ELDERLY_FARE, age_row)

    return toddler_profit + children_profit + \
        adult_profit + elderly_profit

def print_change_in_profit(old_profit, new_profit) :
    print "When using the new age limits for " + \
        "toddlers and children \n the profit" + \
        "changes from EUR %8.2f to EUR %.2f." % \
        (old_profit, new_profit)
    print "The difference is EUR %8.2f." % \
        (new_profit - old_profit)

'''Start Program'''

passengers = sys.stdin.readline().split()
ages = sys.stdin.readline()

age_row = read_age_row(passengers)
new_max_toddler_age = read_age(ages)

normalProfit = calculate_total_profit(MAX_TODDLER_AGE, age_row)
newProfit = calculate_total_profit(new_max_toddler_age, age_row)

print_change_in_profit(normal_profit, new_profit)
```


Assignments

1. List

Read exactly twenty numbers from standard input and print them in reversed order. Do not use a second class yet.

2. BodyMassIndex

Professor Hatzelklatzer has researched the extremely rare Hatzelklatzer-syndrome. There appear to be less cases of the syndrome in odd months than in even months. Further research should reveal if the syndrome affects people more often if they are too heavy.

A way of determining whether someone is too heavy is the *body-mass index* (BMI). This is a measure of a person's weight taking into account their height. The BMI is defined as $weight/length^2$. The World Health Organization (WHO) considers a BMI between 18,5 and 25 as ideal and considers people with such a BMI healthy.

The program receives input consisting of two persons with their name, sex, length and weight.

Dean Johnson	M	1.78	83
Sophia Miller	V	1.69	60

Process this input into structured data. To achieve this, use a useful extra class with useful methods to enhance the structure of the program. Use this structured data to print for each person: an appropriate style of address, surname, the BMI and a statement whether this is considered healthy or not.

Example

```
Mr. Johnson's BMI is 26.2 and is unhealthy.
Mrs. Miller's BMI is 21.0 and is healthy.
```

3. BodyMassIndex2

Professor Hatzelklatzer has realized that although the program written in the BodyMassIndex assignment provides some usefull information on the effects of the BMI on the Hatzelklatzer syndrome, its output is quite difficult to process. To make the output easier to understand, the program should be adapted to analyze a group of test subjects.

The program should read the input, determine the average BMI and count the number of cases of the Hatzelklatzer syndrome amongst people with a lower than average BMI and amongst people with a higher than average BMI. To this end, Professor Hatzelklatzer has provided you with the information of his diagnoses of all the test subjects.

The input of the program now looks like this:

Dean Johnson	M	1.78	83	Yes
Sophia Miller	V	1.69	60	No
...				

Just as in the previous BMI exercise, the input consists of people with their name, sex, length and weight. In addition, the word "Yes" has been added if the person suffers from the Hatzelklatzer syndrome, whilst "No" is added if the person did not suffer from the Hatzelklatzer syndrome. Instead of only two people, the input now consists of an unknown number of people, who all need to be analyzed. The input file can be found on BlackBoard.

The output should look like this:

```
The average BMI of the test subjects is  $x$ .  
There are  $y$  cases of the syndrome amongst people with a BMI  $\geq x$ .  
There are  $z$  cases of the syndrome amongst people with a BMI  $< x$ .
```

Graded assignment

4. Pirate

During his completely miserable life, pirate Abraham Blaufelt has been in search of the lost treasure of Atlantis. On a very fortunate day in the year of the Lord 1642 he enters an abandoned cathedral of a long gone sect in the ancient forests of Poland. Inside he finds a mysterious ancient parchment. The parchment reads:

*Traveler, if you want to reach thine goal,
trace the path through seas by making the broken, whole.*

*5,4 4,5 8,7
Add behind 6,3 3,2 9,6 4,3
Add in front 7,6
Add behind 9,8
Add in front 5,5 7,8 6,5 6,4*

Abraham Blaufelt immediately knew what he was dealing with. A puzzle of which the result is a safe route to the treasure. This route was essential, the sea was crawling with monsters in those days. Since this most fortunate day, almost four hundred years ago, the european tectonic plate has shifted significantly. As a result all coordinates have to be shifted by (1, 0).

Write a program that solves this puzzle. This has to be done in the following way: Start with the coordinates on the first row, add the coordinates of the second row at the back, then add the coordinates of the third row in front etc. Make a new `Coordinate` and `CoordinateRow` class for this assignment. The latter class has methods to add a `CoordinateRow` in front or at the back and methods to add a single `Coordinate` at the front or at the back.

The coordinates of the puzzle are in a file on Blackboard. Every `CoordinateRow` is seperated from another by an '='. Every coordinate in a row is seperated by a space. The x and y values of a coordinate are seperated by a comma.

After all data has been read, the program will print the treasure route on the standard output. The correct route can also be found on Blackboard.

Events and animations

Abstract

This chapter introduces events and animations. These notions are essential for programming interactive programs. The graded assignment of this module will involve programming the game *Snake*. A game such as *Snake* is quite complex and therefore requires a careful approach. Using *stepwise refinement* one begins with a rough sketch of a program, which is developed with increasing detail. Before starting with programming the graded assignment it is compulsory to make such a sketch of the program, which has to be approved before being allowed to continue.

Goals

- Familiarize with events.
- Use events to program an animated program.
- Use events to program an interactive program.
- Use *stepwise refinement* to create complex programs.

Theory

Events

The Replay assignment introduced the Graphical User Interface (GUI). This program was not interactive, it did not react to input provided by the user. To make a program interact with the user, this course uses *events*. An event is for example a mouse-click, a keystroke or even the fact that it is 2 o'clock. Using the following functions from the GUI will allow the program to work with events.

```
get_event()
```

Calling this function will make the program halt and wait for an event to arise. After an event has risen, the function returns an Event-object containing information on the event. The Event class looks like this:

```
class Event:
    def __init__(self, name, data):
        self.name = name
        self.data = data
```

Pressing the letter 'a' will generate an Event-object containing the name "letter" and the data "a". Clicking field 4,3 will return an Event-object containing the name "click" and the data "4 3".

An interactive program, a program that reacts to input generated by the user, works as follows:

1. wait for an event to arise
2. process event
3. repeat.

In Python this can be implemented like this:

```
while True : # infinite loop
    event = ui.get_event()
    process_event(event)
```

The method processEvent needs to determine what has happened, and what should happen as a result. Such a method, that calls different methods according to a specific condition is called a *dispatch*-method.

```
def processEvent(event) :
    if event.name == "click" :
        process_click(event.data)
    elif event1 :
        process_event1(event.data)
    elif (event2) :
        process_event2(event.data)
    else :
        ....
```

 Make the assignment **Events**.


Animations

The way events have been used so far, only allows the program to react to input given by the user. It is not possible for the program to change anything on the screen on its own account. An animated program has to be able to do something without requiring input from the user.

Computer games like snake are called interactive animations. In these programs, the user can influence an animation. In the Replay assignment, the `wait()` method was used to regulate the speed of the animation. This approach is not useable for interactive animation, as the whole program halts when the `wait()` method is executed. The program cannot react to events, not even those generated by the user. Waiting for ten seconds, will cause a mouse-click for example to be processed with a ten second delay. To solve this issue, the `SnakeUserInterface` contains the following method:

```
set_animation_speed(frames_per_second)
```

When this method is called with, for example 24.0 frames per second, the program will generate 24 events per second. These event all have the name "alarm" and data "refresh". The program can now be made to react to these events by refreshing the screen. This way, the `wait()` method does not have to be used, and events generated by the user can be processed instantly.

 Make the assignment **Animation**.

Stepwise refinement

An important part of writing structured programs is *stepwise refinement*. This can be defined roughly in the following way:

- Write down exactly what the program should do, in English or in another natural language.
- Next, step by step elaborate on the description of the program. Again, write in a natural language, or, when it is trivial, directly in Python.
- This process is repeated until the whole program is written in Python.

When the algorithm is correctly executed, the result will be a flawless structured program.

Example Write a program that reads a date in the format: day month year, separated by spaces. The program prints whether the date is correct. This program will be made using stepwise refinement. Important to note is that all programs are a version of precisely the same assignment. The only difference is the amount of English replaced by Python.

Program 1

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

```
'''Start Program'''  
    # read the date  
    # check the date  
    # print output
```

Program 2

```
import sys  
  
''' Assignment: DateCheck  
    Created on 6 aug. 2012  
    @author: Jan Stienstra '''  
  
def is_correct(day, month, year) :  
    # return true if date is correct,  
    # return false otherwise  
  
def read_in_range(input_string, start, end) :  
    # read an integer and return this int if  
    # it is between start and end. Print  
    # an error message otherwise and  
    # terminate.  
  
'''Start Program'''  
date = sys.stdin.readline().split()  
  
day   = read_in_range(date[0], 1, DAYS_IN_MONTH);  
month = read_in_range(date[1], 1, MONTHS_IN_YEAR);  
year  = read_in_range(date[2], 0, YEAR_LIMIT);  
  
if is_correct(day, month, year) :  
    print "The date is correct."  
else :  
    print "The date is not correct."
```

Program 3

```
import sys  
  
''' Assignment: DateCheck  
    Created on 6 aug. 2012  
    @author: Jan Stienstra '''  
  
DAYS_IN_MONTH   = 31  
MONTHS_IN_YEAR  = 12  
YEAR_LIMIT      = 2500  
  
NUMBER_OF_DAYS_IN_A_MONTH = [  
    # list of thirteen values, for each month
```

```

        # the maximum number of days, and a random
        # value at index 0
    ]

def is_leap_year(year) :
    # return true if the year is a leap year
    # return false, otherwise

def is_correct(day, month, year) :
    if day == 29 and month == 2 :
        return is_leap_year(year);
    else :
        return day <= NUMBER_OF_DAYS_IN_A_MONTH[month]

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

'''Start Program'''
date = sys.stdin.readline().split()

day   = read_in_range(date[0], 1, DAYS_IN_MONTH);
month = read_in_range(date[1], 1, MONTHS_IN_YEAR);
year  = read_in_range(date[2], 0, YEAR_LIMIT);

if is_correct(day, month, year) :
    print "The date is correct."
else :
    print "The date is not correct."

```

Program 4

```

import sys

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

DAYS_IN_MONTH   = 31
MONTHS_IN_YEAR  = 12
YEAR_LIMIT      = 2500

NUMBER_OF_DAYS_IN_A_MONTH = [0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

def is_leap_year(year) :
    return year % 400 == 0 or \
        (year % 4 == 0 and year % 100 != 0)

```



```
def is_correct(day, month, year) :
    if day == 29 and month == 2 :
        return is_leap_year(year);
    else :
        return day <= NUMBER_OF_DAYS_IN_A_MONTH[month]

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

'''Start Program'''
date = sys.stdin.readline().split()

day   = read_in_range(date[0], 1, DAYS_IN_MONTH);
month = read_in_range(date[1], 1, MONTHS_IN_YEAR);
year  = read_in_range(date[2], 0, YEAR_LIMIT);

if is_correct(day, month, year) :
    print "The date is correct."
else :
    print "The date is not correct."
```

Assignments

1. Events

Before starting this assignment, read the theory about **Events**.

Write a program using a SnakeUserInterface of 40×30 which has the following features:

- Clicking on a square results in a piece of wall to be placed on that square.
- Pressing the space bar erases all the walls.
- The program prints the name and data of all events that occur.

2. Animation

Before starting this assignment, read the theory about **Animations**.

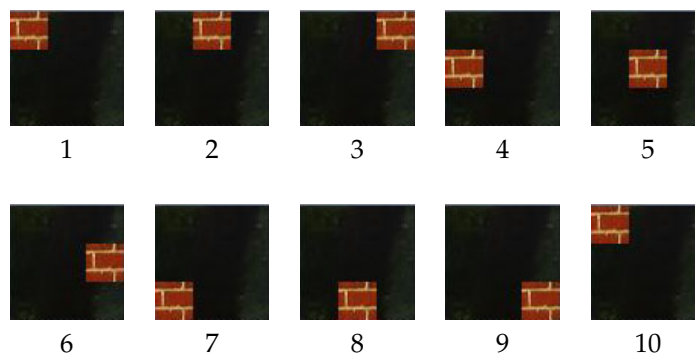
The goal of this assignment is to make an animated program in which a piece of wall moves across the screen. The piece of wall starts out on (0,0) and moves right a square at a time. Upon reaching the end of a row, the piece of wall will move to the first square of the next row. When the piece of wall reaches the end of the last row, it is transferred back to the initial (0,0) position.

On top of this make sure the program implements the following features:

- The animation should slow down 0.5 frames per second when \leftarrow (left arrow) is pressed.
- The animation should speed up 0.5 frames per second when \rightarrow (right arrow) is pressed.
- The piece of wall should change into a green sphere (a part of a snake) when g is pressed. Pressing g again will revert the change.

Use the SnakeUserInterface for this assignment.

Example



Graded assignment

3. Snake

Before starting this assignment, read the theory about **Stepwise refinement**. In addition, before starting with programming, a draft of the program has to be approved.

A logical step forward from interactive animated programs is games. The goal of this assignment is to program the classic computer game, *Snake*.

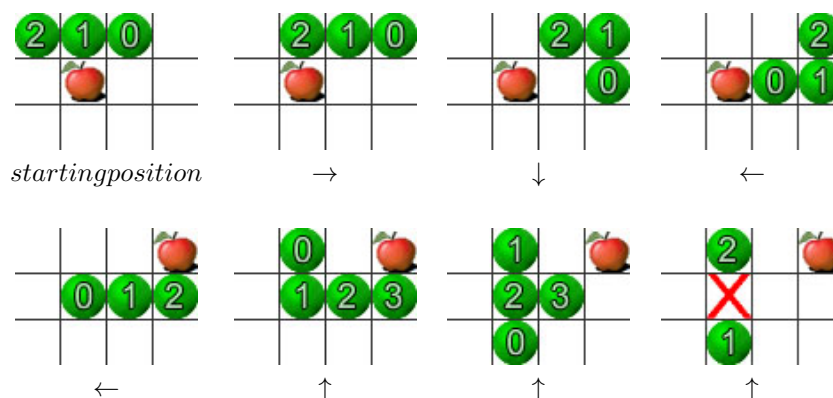
The goal of Snake is to create a snake as long as possible. This is achieved by guiding the snake to apples, lying about on the field. The snake cannot stop moving, and dies whenever it hits something. Because the snake is growing longer and longer as the game progresses, it is increasingly difficult to avoid collisions with the snake itself.

At the start of the game, the snake consists of two pieces at the coordinates (0,0) and (0,1). As said before, the snake is always moving. At the start of the game, it moves to the right. When the user presses one of the arrow keys, the snake changes direction.

At every moment in the game, there is always an apple somewhere in the field. If the snake hits an apple, the snake becomes one piece longer in the next screen refresh. A new apple is placed on a random location, excluding all places covered by the snake.

When the snake reaches the end of the screen, it will re-emerge at the other end.

Example The example below shows a short game of snake, played on a 4x3 field. The game to be designed in this assignment will have a field measuring 32x24. The arrow indicates in which direction the snake is travelling. The numbers on the snake indicate its position in the row.



This assignment uses the SnakeUserInterface.

Bonus Edit the program in such a way that it accept a level as input. A level defines a number of walls, which the player has to avoid. Levels can be found on Blackboard. The structure of these files is as follows: first the coordinates at which the snake is initialized are given followed by an =. Next, the initial direction of the snake is given, again followed by an =. Finally, all the coordinates at which to place walls are given.

Coordinates are formatted in the following way: one coordinate per line, in the format: $x < \text{space} > y$. The initial direction is one of four strings: "L" (Left), "R" (Right), "U" (Up) or "D" (Down).

An example of a piece of such a file:

```
1 0
0 0=R=3 3
4 3
5 3
6 3
7 3
8 3
etc...
```

Bonus**Abstract**

This chapter contains the bonus assignment. Students who complete the bonus assignment will be given half a point on top of their final grade.

**Warning**

As this is a bonus assignment, there is no time reserved for this assignment during the lab sessions. This assignment is solely meant for those students who have finished all other assignments. Students are only allowed to start on this assignments if all other assignments have been submitted.

Graded assignment**1. Life**

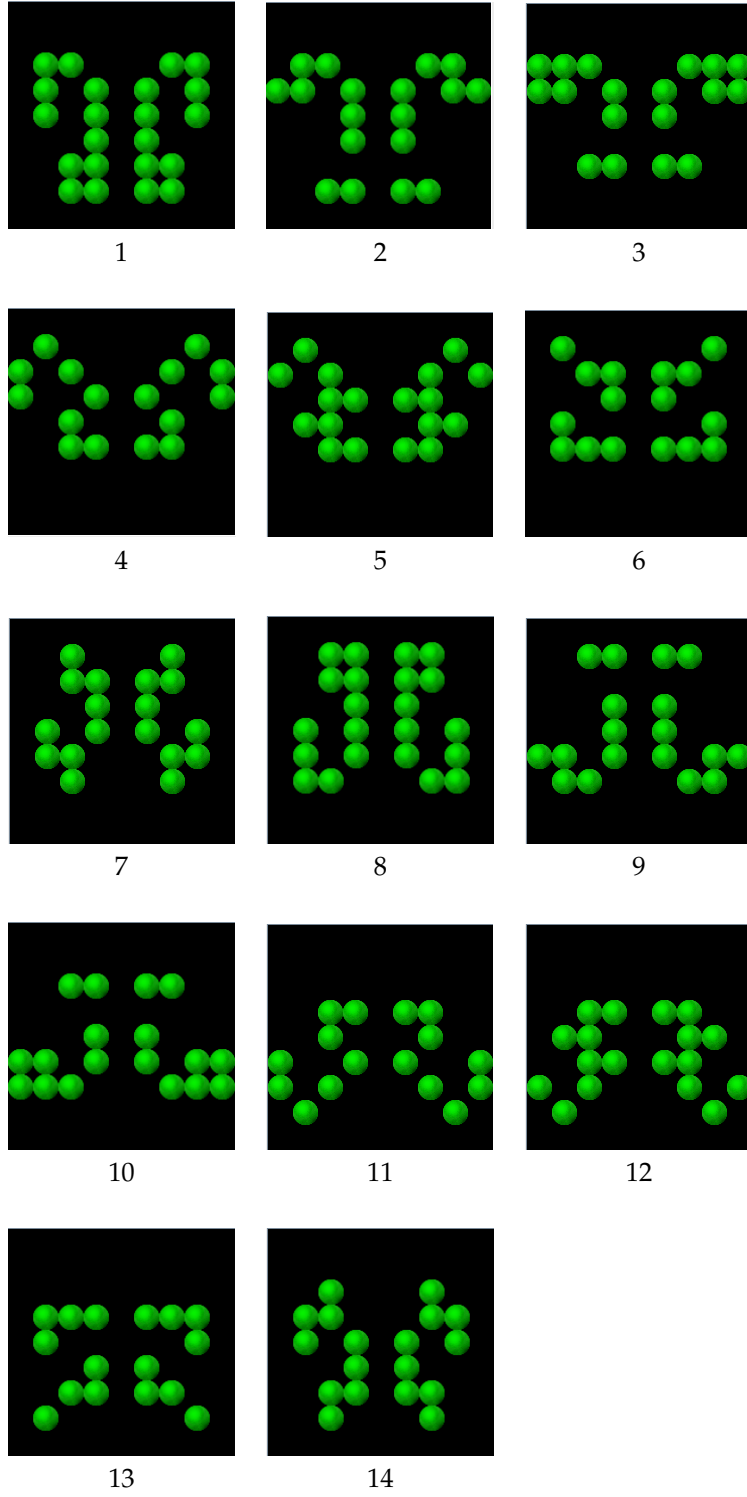
The Game of Life was invented by J.H. Conway. Two publication in the “Scientific American” by Martin Gardner saw the game introduced to the public. Life is played on a board of $n \times n$ squares, representing a population of dead and living cells. A living cell can either die or continue to live, based on a set of rules. A dead cell can either become alive again, or remain dead. Every cell has eight neighbours, except the cells on the edge of the board:

1	2	3
4	*	5
6	7	8

The set of rules determining the fate of a cell:

1. X is currently dead: If X has exactly three living neighbours, X becomes alive again. In all other cases, X remains dead.
2. X is currently alive: If X has zero or one neighbour(s), X dies of loneliness. If X has two or three living neighbours, X remains alive. In all other cases X dies of a shortage of living space.

An example using a 9 x 9 board:



It is possible for a figure to die (an empty board) or become an oscillator, i.e. generation $n = \text{generation } n + p$, for any n above a certain value. If the period equals 1 ($p=1$), it is called a still figure.

Write a program that takes a starting configuration from a file and generates generations as long as the figure has not died, become an oscillator with a certain p , or exceeds the maximum number of generations. When the program terminates, print a message stating why the program has terminated and if the figure has become an oscillator, its period. If the period of the oscillator is 1, the message should read "Still figure" instead of "Oscillator". The input files on Blackboard have the following structure:

- On the first line, the maximum number of generations, ranging from 1 to 100.
- On the second line, the largest period for which the figure should be tested to oscillate, ranging from 2 to 15.
- After this, a starting configuration for a 9×9 board, made up from 9 line of 9 characters. A living cell is represented by an 'x', a dead cell is represented by ' '

This assignment uses the LifeUserInterface.