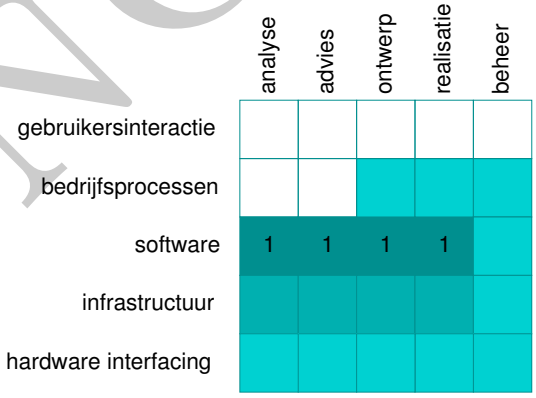
HOGESCHOOL ROTTERDAM / CMI

Development 2

INFDEV02-2

Number of study points:	4 ects
Course owners:	Dr. Giuseppe Maggiore, Tony Busker

Modulebeschrijving

Module name:	Development 2
Module code:	INFDEV02-2
Study points and hours of effort:	This module gives 4 ects, in correspondence with 112 hours: • 3 x 6 hours frontal lecture • 3 x 6 hours practicum • the rest is self-study
Examination:	Written examination and practicums (with oral check)
Course structure:	Lectures, self-study, and practicums
Prerequisite knowledge:	Basic imperative control structures and datatypes, as per INFDEV02-2.
Learning tools:	• Book: Think Python; author A. B. Downey (`http://www.greenteapress.com/thinkpython/`) • Presentations (in pdf): found on N@tschool and on the GitHub repository `https://github.com/hogeschool/INFDEV02-2` • Assignments, to be done at home and during practical lectures (pdf): found on N@tschool and on the GitHub repository `https://github.com/hogeschool/INFDEV02-2`
Connected to competences:	
Learning objectives:	At the end of the course, the student can: • **understand** the concept of abstraction through function definition `FUNABS` • **use** and **design** functions `FUNDEF` • **use** and **design** recursive functions `FUNREC` • **understand** the concept of abstraction through class definition `CLSABS` • **use** and **design** classes without inheritance or interfaces `CLSDEF` • **use** recursively defined lists `LISTS` • `RECDATA` • **use** standard predefined collections `STDDS`

Content:	• basic concepts of computation from a logical standpoint • basic concepts of computation from a concreter perspective in terms of storage and instructions • variables (in Python 2) • primitive datatypes and expressions (in Python 2) • conditional control-flow statements (in Python 2) • looping control-flow statements (in Python 2)
Course owners:	Dr. Giuseppe Maggiore, Tony Busker
Date:	1 december 2015

1 General description

Programming is one of the most ubiquitous activities within the field of ICT. Many business needs are centered around the gathering, elaboration, simulation, etc. of data through programs.

This course covers intermediate aspects of building abstractions (functions, data structures, and classes) in the Python programming language (version 3).

1.1 Relationship with other teaching units

Subsequent programming courses build upon the knowledge learned during this course.

The course also provides a semantic background in order to understand the implementation of SQL queries within the framework of higher order (list) functions.

Knowledge acquired through the programming courses is also useful for the projects. A word of warning though: projects and development courses are largely independent, so some things that a student learns during the development courses are not used in the projects, some things that a student learns during the development courses are indeed used in the projects, but some things done in the projects are learned within the context of the project and not within the development courses.

2 Course program

The course is structured into six lectures. The six lectures take place during the six weeks of the course, but are not necessarily in a one-to-one correspondance with the course weeks. For example, lectures one and two are fairly short and can take place during a single week.

2.1 Lecture 1 - data structures

The first lecture covers basic concepts of data structures as a means to avoid brittle representation of data by means of multiple basic variables:

Topics

- Mechanism of abstraction;
- The necessity for data structures;
- Data structures in Python (class);
- Semantics (Heap, Stack);
- Layers of abstraction.

2.2 Lecture 2 - lists

The second lecture covers a well-known data structure that exemplifies good design and reasoning in terms of encapsulation and genericity:

Topics

- The need for a variable to contain an *unknown* number of values;
- Abstraction of list: `Node (Head), Tail, Empty`;
- Implementation of list (Python 3);
- Semantics of list: `Heap and Stack`

2.3 Lecture 3 - functions

The third lecture covers abstraction over (groups of) instructions and statements through functions:

Topics

- Abstraction operations (functions)
- The need for functions;
- Creating and using functions in Python;
- Formal and actual parameters and return;
- Brief introduction to: scope (local and global variables) and visibility;
- Syntax and semantics;
- Introduction to recursion;

2.4 Lecture 4 - higher order functions and SQL

The fourth lecture covers higher order functions (HOF's) and connects them with the world of databases by sketching the connection between list HOF's and SQL queries:

Topics

- What are HOF's and *why we do need them*?
- Functions as parameter;
- Lambda: λ-expressions (syntax and semantics);
- Fundamental operations on list: `transform, filter, fold`;
- Using HOF's;
- SQL vs list HOF's.

2.5 Lecture 5 - methods

The fifth lecture covers abstraction of data structures and functions within the single container of classes:

Topics

- Joining functions (methods) and data to classes;
- Designing a class;
- Concrete implementation of a class;
- Syntax and semantics;
- special method names;
- rebuilding the list data structure;
- Brief introduction immutability and mutability.

2.6 Lecture 6 - collections library

The sixth (and last) lecture covers the existing collections library of Python, and illustrates how it can be used instead of rebuilt by hand:

Topics

- lists;
- tuples;
- maps;
- sets.

3 Assessment

The course is tested with two exams: a series of practical assignments, a brief oral check of the practical assignments, and a theoretical exam. The final grade is determined as follows:

`if theoryGrade` $\geq$ `75% & practicumCheckOK then return practicumGrade else return insufficient`
This means that the theoretical knowledge is a strict requirement in order to get the actual grade from the practicums, but it does not reflect your level of skill and as such does not further influence your grade.

Motivation for grade A professional software developer is required to be able to program code which is, at the very least, *correct*.
In order to produce correct code, we expect students to show: *i*) a foundation of knowledge about how a programming language actually works in connection with a simplified concrete model of a computer; *ii*) fluency when actually writing the code.
The quality of the programmer is ultimately determined by his actual code-writing skills, therefore the final grade comes only from the practicums. The quick oral check ensures that each student is able to show that his work is his own and that he has adequate understanding of its mechanisms. The theoretical exam tests that the required foundation of knowledge is also present to avoid away of programming that is exclusively based on intuition, but which is also grounded in concrete and precise knowledge about what each instruction does.

3.1 Theoretical examination INFDEV02-2

The general shape of a theoretical exam for `INFDEV02-2` is made up of a series of highly structured open questions. In each exam the content of the questions will change, but the structure of the questions will remain the same. For the structure (and an example) of the theoretical exam, see the appendix.

3.2 Practical examination INFDEV02-2

Each week there is a mandatory assignment. The assignments of week 4, 5 and 6 will be graded. Each assignment is due the following week. The sum of the grades will be the *practicumGrade*. If the course is over and *practicumGrade* is lower than 5,5 then you can retry (herkansing) the practicum with one assignment which will test all learning goals and will replace the whole *practicumGrade*. The following rules apply to the assignment:

- The assignments are to be uploaded to N@tschool in the required space (Inlevermap);
- Only basic operations are allowed for the assignment unless explicit permitted otherwise;

The oral check (preferred during the practicums) is done on work uploaded to N@tschool:

- two (2) questions per assignment about *What does this line (these lines) do?*
- the exercise runs correctly

3.3 Oral check INFDEV02-2

During the oral check, the teacher will verify ownership and competence with the code that was handed in during the practicum. **This is not a replacement for handing in solutions to the practicum assignments**. This procedure is the one that will effectively determine the grade of the practicum. The procedure works as follows:

1. During the last practicum lecture, the teacher will give students an URL;
2. At this URL, students can find a series of incomplete solutions for the practicum assignments;
3. During the three hours of practicum, students will restore the missing functionality;
4. Succesful restoring of the functionality will give the points for that assignment; failure in restoring the functionality will result in zero points for that practicum;

The teachers still reserve the right to check the practicums handed in by each student, and to use it for further evaluation.

Theoretical examination INFDEV02-2

The general shape of a theoretical exam for `DEV II` is made up of a series of highly structured open questions.

3.3.0.1 Question I: abstracting patterns with functions

General shape of the question: *Given a problem description, define one or more functions in order to solve the original problem.*

Concrete example of question: *Define a recursive* `range` *function to create a custom list (only use* `Empty` *and* `Node`*, see Appendix) with all the elements between two given numbers.*

Concrete example of answer: *The resulting code is:*

```
def range(l, u):
  if l > u:
    return Empty()
  else:
    return Node(l, range(l+1,u))
```

Points: *25%.*

Grading: *All points for correct function, minor mistakes (wrong check, some elements might be missing, etc.) half points, wrong function (infinite recursion, iterative version, etc.) zero points.*

Associated learning goals: `FUNABS`, `FUNDEF`, `FUNREC`, `RECDATA`.

3.3.0.2 Question II: runtime behaviour of functions

General shape of the question: *Given a function definition and a sample call, show stack and heap at all steps of the computation.*

Concrete example of question: *Given the following function definition and a sample call, show stack and heap at all steps of the computation.*

```
def f(n):
  if n <= 1:
    return n
  else:
    return n * f(n-1)

f(3)
```

Concrete example of answer: *The last call of the stack is :*

```
S: PC   f   PC   n   f   PC   n   f   PC   n
    7  nil   2   3  nil   2   2  nil   2   1
H: always empty
```

The stack will then unwind as follows:

```
S: PC   f   PC   n   f   PC   n   f   PC   n
    7  nil   2   3  nil   2   2   1   3   1
```

```
S: PC   f   PC   n   f   PC   n
    7  nil   2   3  2*1   4   2
```

```
S: PC   f   PC   n
    7  3*2   4   3
```

Points: *25%.*

Grading: *All points for all stack frames and values, half points for at least half correct stack frames and values, otherwise zero points.*

Associated learning goals: FUNABS, FUNDEF, FUNREC, RECDATA.

3.3.0.3 Question III: classes

General shape of the question: *Given a description, give the implementation of a class and its methods in Python.*

Concrete example of question: *Define a* `Counter` *class with a single method,* `Tick`*, which increments the internal* `cnt` *of the class. Also provide an implementation of* `__str__`*)*

Concrete example of answer: *The resulting code is:*

```
class Counter:
  def __init__(self):
    self.cnt = 0
  def Tick(self):
    self.cnt = self.cnt + 1
  def __str__(self):
    return "Ticked " + str(self.cnt) + " times"
```

Points: *25%.*
Grading: *All points for correct answer, half points for at least correct implementation of methods __init__ and Tick, otherwise zero points.*

Associated learning goals: CLSABS, CLSDEF.

3.3.0.4 Question IV: standard libraries

General shape of the question: *Define a loop that performs some simple operation on a standard data structure.*

Concrete example of question: *Define a loop that sums all positive elements of a Python list* `l` *which contains only integers. Finally, print the sum.*

Concrete example of answer: *The resulting code is:*

```
sum = 0
for x in l:
  if x > 0:
    sum = sum + x
print(sum)
```

Points: *25%.*

Grading: *All points for correct answer, otherwise zero points.*

Associated learning goals: ARR.

Exam sample

What follows is a concrete example of the exam.

3.3.0.5 Question I: abstracting patterns with functions

Define a `map` function to transform all elements of the input list (defined with `Empty` and `Node`, see Appendix) according to a given function `f`.

Answer: *The resulting code is:*

```
def map(l,f):
  if l.IsEmpty():
    return Empty()
  else:
    return Node(f(l.Head()), map(l.Tail(), f))
```

Points: *25%.*

Grading: *All points for correct function, minor mistakes (wrong check, some elements might be missing, etc.) half points, wrong function (infinite recursion, iterative version, etc.) zero points.*

3.3.0.6 Question II: runtime behaviour of functions

Given the following function definition and a sample call, show stack and heap at all steps of the computation.

```
def filter(l,p):
  if p(l.head):
    return Node(l.head, filter(l.tail, p))
  else:
    return filter(l.tail, p)

filter(Node(1,Node(2,Empty())), lambda x: x >= 2)
```

Answer: *Each call of the stack should contain a value for `l` and one for `p`. The heap should contain a value for each node of the list, and for the lambda function of `p`.*

```
S: PC   filter   PC     l        p
    7    nil      2   ref(2)   ref(3)

H: 0              1                          2                          3
  [ ]   [head->1;tail->ref(0)]   [head->2;tail->ref(1)]   lambda x: x >= 2
```

After each step, the stack grows but the heap does not:

```
S: PC   filter   PC     l        p     filter   PC     l        p
    7    nil      5   ref(2)   ref(3)  nil      2   ref(1)   ref(3)

H: 0              1                          2                          3
  [ ]   [head->1;tail->ref(0)]   [head->2;tail->ref(1)]   lambda x: x >= 2
```

The rest follows similarly. Watch out for reconstruction of recursive result with correct elements wrt returned value of `p(l.head)`.
Points: *25%.*

Grading: *All points for all stack frames and values, half points for at least half correct stack frames and values, otherwise zero points.*

3.3.0.7 Question III: classes

Concrete example of question: *Define a* `Train` *class with attributes:*

- `Position` *of the ship in the map (a 2D vector, see Appendix)*
- *amount of* `Passengers`
- *amount of* `Containers`

and methods:

- `TravelTo` *that receives a* `Station`[1] *as a destination and changes*
  - *the* `Position` *of the train to the* `Position` *of the station*
  - *the* `Passengers` *of the train (and the* `WaitingPassengers` *of the station)*
  - *the* `Containers` *of the train (and the* `WaitingContainers` *of the station)*

Points: *25%.*

Grading: *All points for all attributes and methods, half points for at least half correct attributes and methods, otherwise zero points.*

Answer: *The implemented class is:*

```
class Station:
  def __init__(self, p, wp, wc):
    self.Position = p
    self.WaitingPassengers = wp
    self.WaitingContainers = wc

class Ship:
  def __init__(self, p):
    self.Position = p
    self.Passengers = 0
    self.Containers = 0
  def NavigateTo(self, port):
    self.Position = port.Position
    self.Passengers = port.WaitingPassengers
    port.WaitingPassengers = 0
    self.Containers = port.WaitingContainers
    port.WaitingContainers = 0
```

3.3.0.8 Question IV: standard libraries

Concrete example of question: *Define a loop that adds all odd elements in a given Python list* `l` *which contains only integers. If the list is empty the result should be* `0`*. Finally, print the result.*

Points: *25%.*

[1] The `Station` class has at least the attributes `Position`, `WaitingPassengers`, `WaitingContainers`

Grading: *All points for correct iteration and sum, half points for wrong use of indices or wrong iteration, zero points otherwise.*

Answer: *The implemented class is:*

```
l = [1,2,3,4]
res = 0
for x in l:
  if x % 2 == 1:
    res = res + x
print(res)
```

3.4 Exam appendix

3.4.0.9 List implementation

```
class Empty:
  def IsEmpty(): return True
Empty = Empty()

class Node:
  def IsEmpty(): return False
  def Head(self): return self.Head
  def Tail(self): return self.Tail
  def __init__(self, x, xs):
    self.head = x
    self.tail = xs
```

3.4.0.10 Vector2 implementation

```
class Vector2:
  def __init__(self, x, y):
    self.X = x
    self.Y = y
  def Length(self):
    return math.sqrt(self.X * self.X + self.Y * self.Y)
  def __neg__(self):
    return Vector2(-self.X, -self.Y)
  def __add__(self, other):
    return Vector2(self.X + other.X, self.Y + other.Y)
  def __sub__(self, other):
    return self + (-other);
  def __mul__(self, k):
    return Vector2(self.X * k, self.Y * k)
  def __str__(self):
    return "(" + str(self.X) + "," + str(self.Y) + ")"
  def Zero():
    return Vector2(0.0, 0.0)
  def UnitX():
    return Vector2(1.0, 0.0)
  def UnitY():
    return Vector2(0.0, 1.0)
```

Bijlage 1: Toetsmatrijs

Learning goals	Dublin descriptors
FUNABS	1, 4
FUNDEF	1, 2, 4
FUNREC	1, 2, 4
CLASABS	1, 4
CLSDEF	1, 2, 4
LISTS	1, 2, 4
RECDATA	1, 2, 4
STDDS	1, 2, 4

Dublin-descriptors:

1. Knowledge and understanding
2. Applying knowledge and understanding
3. Making judgments
4. Communication
5. Learning skills