UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Karl-Aksel Puulmann

Python module for automatic testing of programming assignments

Bachelor's Thesis (6 ECTS)

Supervisor: Aivar Annamaa, BA
Supervisor: Margus Niitsoo, PhD

Tartu 2014

Python module for automatic testing of programming assignments

Abstract:

This thesis contains a description of a Python module for automatically assessing programming assignments in introductionary programming courses. Most notably, the module allows to test both input-output based tasks and functions at the same time.

In the first part, existing automatic assessment systems are analyzed. Then a guide is given on how to use the module for testing different task types, how to extend it and how to use it within other grading systems. Lastly the thesis deals with implementation decisions, on how to secure testing and usage experiences from two different courses.

Keywords:

Automated grading; Automated testing; Education; Python; Docker

Pythoni teek programmeerimisülesannete automaatseks testimiseks

Lühikokkuvõte:

Antud töö sisaldab kirjeldust Pythoni teegi kohta, mille abil saab automaatselt testida programmeerimisülesandeid sissejuhatavates programmeerimiskursustes. Antud teegi abil saab testida nii sisend-väljundipõhiseid ülesandeid ja ka funktsioone samal ajal.

Töö esimeses osas analüüsitakse olemasolevaid hindamissüsteeme. Lisaks tutvustatakse kuidas loodud teeki saab kasutada erinevate ülesannetetüüpide testimiseks, kuidas teeki laiendada ning kuidas teda kasutada olemasolevate hindamissüsteemide sees. Kirjeldatakse ka teegi arhitektuuri, kuidas turvata võõra koodi käivitamist ning tuuakse välja kogemused teegi kasutamisest eri kursustes.

Võtmesõnad:

Automaathindamine; Automaattestimine; Haridus; Python; Docker

Contents

Introduction

This thesis describes the design of a Python module created by the author that makes writing automated tests for introductory programming courses simpler.

Chapter 1 describes the background of the work by analyzing existing testing modules and programs and their shortcomings for testing typical homework tasks. The last section gathers the requirements and goals for the proposed solution.

Chapter 2 depicts the basic design decisions of the module and how to write tests using the module. It also introduces ways of extending the module to support new task types and how to use the web IDE created as a part of this thesis.

Chapter 3 shows how the module was implemented by analyzing each component in depth, also explaining reasons for some design decisions. The later part of the chapter depicts security problems with testing unsecured code and how sandboxing the module is done.

Chapter 4 discusses how the system has been used in various courses and outlook for future work.

Chapter 1

Problem statement

1.1 Background

The Python programming language is currently used in many universities, including the University of Tartu, as a basis for introductory programming courses. While there are many testing frameworks that support testing Python code, most of the testing of homeworks and programming tests in those courses is still done by hand.

In the author's opinion, the main reason for that is the large diversity of task types that need to be tested and the flexibility required in assessing beginners' programs. For example, the following task types were used in Computer Programming course homeworks in autumn 2013:

1. **Input-output (IO) based tasks** - in this task type, the program will ask for input from the user, do some simple calculations based on the input and output an answer in prescribed format. This might also involve reading and writing files.
    - **Interactive IO tasks** - IO based task where the user and the program are continually communicating. Examples: tic-tac-toe game against the computer, computer tries to guess a number 1-10000 selected by the user.
2. **Function-based tasks** - the student has to create a function (or several functions) which does something according to task statement. Usually involves returning some value from the function.
3. **Drawing tasks** - draw a picture onto the screen using turtle module inspired by logo turtle graphics [1].
4. **Limited tasks** - used in combination with previous task types. The student may

not use some construction in the Python language, e.g. loops for writing a function. This can be tested by doing static analysis such as parsing the tested program into a representation called Abstract Syntax Tree (AST) and looking for forbidden constructions there.

1.2 Analysis

To get a list of requirements for such a testing tool, existing solutions were analyzed for their strengths and weaknesses, taking into account what task types the solution needs to support.

Testing systems fall broadly into two types:

1. **Input-output based testing** where the program has a fixed input and the output of the program is checked character-by-character against the expected output, sometimes ignoring whitespace differences.
2. **Functional testing** where functions are tested by calling them with specific arguments and variables and classes are checked against expected values.

These systems will be evaluated using the following criteria:

1. What task types are easy to test? Are any impossible? Do we need to change existing tasks to use the system?
2. How easy is it to write tests? Is there a need to write extra code each time or some other extra requirements?
3. Can the system be extended to support new task types?
4. How helpful is the feedback provided to the student? Can the feedback be used to fix issues with the solution?
5. Can we safely test untrusted code? What extra steps do we need to take for securing the system?

1.2.1 Input-output based testing frameworks

Using IO-based testing is very common in programming competitions, due to being very robust and language-agnostic since the system only needs to know how to compile and run the student's program. There exist many online judge systems such as Moodle virtual programming lab [2], Mooshak contest system [3] and Sphere online judge [4]. These systems have also been used for testing homeworks in algorithmics courses [5].

Tests for these systems are specified by an input and expected output file. Testing is done by first compiling the program being tested and then piping the input data to the program. The output of the program is compared against the expected output character by character or by sometimes ignoring whitespace.

Doing fixed IO tests is most natural in these systems. Some systems like Jutge.org [6] and Sphere online judge also provide support for testing interactive tasks - the task creator can provide scripts which interact with the program. Grading limited tasks is also sometimes possible if the system supports custom build scripts which can do static analysis. Outside of this, the system is quite rigid and generally not extendable.

Test input data for these systems are usually prepared by hand for smaller inputs and programmatically for larger inputs by writing custom generation scripts. Expected output data is generated by running a sample solution on the input data. This method of preparing tests usually has one unfortunate side-effect: since test data generators are not used by the testing system, they are usually not published. This makes it harder to change existing tasks or create new similar tasks which have slightly different requirements since the test generators often need to be rewritten.

Feedback in systems mentioned before is mostly given in the form of a few standard messages such as **Accepted**, **Wrong answer**, **Compile error** and **Runtime error**. Some systems also expose the input data and expected answer to students. This sort of feedback is not very helpful, especially on larger input data sets where it is hard to understand what the correct answer should be without first visualizing the underlying structure of the input data.

These systems are usually well-secured against malicious users, leveraging such tools as chroot() jails, rlimits or by using separate virtual machines for testing [7].

1.2.2 Functional testing framework - unittest

Using Python `unittest` module [8], testing is done by writing functions which check the behaviour of the tested program. Notifying of errors works through raising exceptions, usually through assert-like functions[1] or by propagating exceptions caused by the tested code.

[1]For example such a function might check if its two arguments are equal. If not, an exception is raised, with the error message containing both argument values.

This means that it is most convenient for testing function-based tasks, but the approach is also extendable for testing limited tasks. However, by default all tests for a program are run in the same process and in the same thread which means that if the tested code has some significant side-effects (such as modifying global variables or files it relies on), the next tests may fail even if the code behaves correctly under normal conditions.

The module is extendable by allowing the user to write their own subclasses for testing, making it possible to reuse code for similar tasks. The negative side of this is that tests written with this module are quite verbose.

Testing tasks which do IO is not impossible but simply hard with this module, requiring writing code which starts the tested program and communicates with it, feeding it new input data and processing the output. Implementing this would be a large and non-trivial task which would require about as much effort as writing a new testing engine.

Feedback for testing is given in the form of a traceback, which contains information about the functions being executed when an error was raised and the error message. In case the tested code itself raised an error (e.g. a command references a variable which does not exist), the traceback helps the programmer to find the offending line easily. In case the testing code notifies of an error the error message should display information about what went wrong if assertion methods bundled with `unittest` are used. For example, if a function returned the wrong answer, the error message traceback contains both the expected value and returned value.

It should be noted that while tracebacks are very helpful for an experienced programmer, new students often have trouble understanding them.

The module itself is not secured by default, as it was built under the assumption that only trusted code is ever tested. This means that proper sandboxing is necessary and also care must be taken so that tested code could not override methods the testing framework relies on. Securing the module becomes easier if the framework was extended in a way that each test runs in a separate process.

1.3 Summary

As seen from analysis, neither of the two common testing system types completely satisfies all of our requirements. This makes it hard to test homeworks in introductory programming courses taught at University of Tartu without making heavy modifications to the course contents or significantly modifying the framework being used.

However such a system can be built by putting aside some of the aspects of the previous two systems (such as independence of programming language) and combining other ideas.

Chapter 2

Python module `grader`

For the purpose of automatically testing homework assignments, a new Python module named `grader` was created. This chapter discusses the main design decisions and how to write tests using the module by working through various examples.

2.1 Scope

As seen from the analysis in Chapter 1, existing testing engines vary much in scope, starting from small modules such as `unittest` to full educational systems which in addition to testing support and do course content management and grading such as Jutge.org and Sphere online judge.

While larger systems are very much needed, this thesis focuses only on creating a module which:

- allows teachers to specify tests for all task types in the same language and programming style.
- is expressive and can be extended to support new task types (Section 2.6).
- gives valuable feedback to the student which helps them debug their code.
- can be used within existing grading systems.
- can be securely used for testing homeworks (Section 3.4)

2.2 Design decisions

The API design of the `grader` module is largely inspired both by `Flask` web framework [9] and also Python `unittest` [8] and `nose` [10] modules.

The main design decisions made were as follows:

1. All tests are encapsulated as functions. This yields shorter and more readable tests.

2. Each test should have a unique human-readable description, which describes the test data and the expected result (e.g. description `"function add(5, 2) should return 7"`).

3. Test functions have access to the output, variables and functions of the tested solution, and can also insert strings into the solution's input stream.

4. Python function decorators[1] are used to manipulate test configuration - setting time limits, adding extra parameters to test function or creating temporary files. Another usage of decorators is wrapping tests, for example to create several tests with same test code but different arguments.

5. Each test function runs in parallel with the tested code. This allows the framework to do IO testing and functional testing at the same time. This is explained in detail in Chapter 3.

6. Notifying of failed tests happens through exceptions. As the error messages are

[1]Python decorators are functions which take a function as an argument and return another function which then replaces the original function. They are often used to make functions cache their results or to log return values and exceptions (example in Listing 2.1).

```python
def log_this(function):
    def replaced_function(argument):
        result = function(argument)
        print("Call with argument {1} returned {2}"
                .format(argument, result))
        return result
    return replaced_function

@log_this
def f(x):
    return 2*x
```

Listing 2.1: Example of a decorator declaration in Python. Each time function `f` is called the result of calling the function is written to the output.

shown to the student they should contain relevant information about what went wrong. Raising exceptions is usually done using the assert statement, but the tester also notifies the student if their code raised an error (e.g. division by zero).

7. Tests are run in isolation, each in a separate process within a separate directory for each test. The need for this is explained in Section 3.1.2.
8. Output of running the tests is a JSON-serializable dictionary that can be further processed by other programs.

2.3 Example - interactive search

To help understand how to use the module, a simple task with tester code is presented below as an example.

2.3.1 Task statement

Write a program which tries to guess an integer between 1 and 10000 that the user picks with as few guesses as possible. More specifically, each time the program outputs a number, the user will answer that the number is either "too large", "too small" or "correct".

Example session (program output in bold):

```
5
too small
7
too large
6
correct
```

2.3.2 Analysis

This is a textbook example of an interactive program. Sample solution using binary search can be seen in Listing 2.2.

Using binary search finding any number should take no more than $log_2(10000) = 14$ guesses.

```
# search using binary search.
bottom = 1
top = 10000

while True:
    guess = (top + bottom) // 2
    print(guess)

    # ask the user how the number compares
    answer = input()

    # update search boundaries
    if answer == "too large":
        top = guess - 1
    elif answer == "too small":
        bottom = guess + 1
    elif answer == "correct":
        break
```

Listing 2.2: Sample solution

2.3.3 Tester code

Listing 2.3 shows tester code for this task. In it the **functions as tests** principle comes into play within the `search_tester` function. Since the function is decorated with `test_cases` and given 12 different arguments the module registers 12 different tests, each of which calls `search_tester` with a different number to search for.

Inside the function, the argument **m** is an object which contains the standard output and input streams of the solution. Using these the tester communicates as if it were a user using the program.

Note that the contents of `search_tester` are quite similar to the contents of a program that makes the user guess a number between 1 and 10000. In other words, the tester code is the inverse of the solution and simulates how the user is communicating with the tested program. Everything else surrounding the function is configuration - information about how the tests are run and what the description shown to the user should be.

```python
# import functions and decorators from the tested module
from grader import *

# This creates 12 tests, each for searching a number
@test_cases(
    # list all the 12 numbers to search for
    # each one gets passed in to the function as the
    # second argument - searched_number
    [1, 10000, 5000, 3, 9990, 7265, 8724, 2861, 2117, 811, 6538,
        4874],
    # The description of the test, shown to the user.
    # {0} is replaced by the searched number
    description="Searching for number {0}"
)
def search_tester(m, searched_number):
    # test function - First argument (always given) is a container
    # for the users program and for the stdin/stdout.
    # Second is the searched number (see above).
    found = False
    guesses = []

    while len(guesses) < 15 and not found:
        # Get what the user guessed since last time we asked.
        guess = int(m.stdout.new())
        guesses.append(guess)

        # let the program know if the guess was
        # correct, too large or too small.
        if guess < searched_number:
            m.stdin.put("too small")
        elif guess > searched_number:
            m.stdin.put("too large")
        elif guess == searched_number:
            m.stdin.put("correct")
            found = True

    # If program didn't find the solution fast enough,
    # notify that the program made too many guesses.
    # This raises an AssertionError if found is False.
    assert found, (
        "Program made too many guesses.\n" +
        "Guesses were: {}".format(guesses))
```

Listing 2.3: Tester for this task

2.3.4 Running the tests

The tester can be run via the command line with the following command:

```
python -m grader path/to/tester.py path/to/solution.py
```

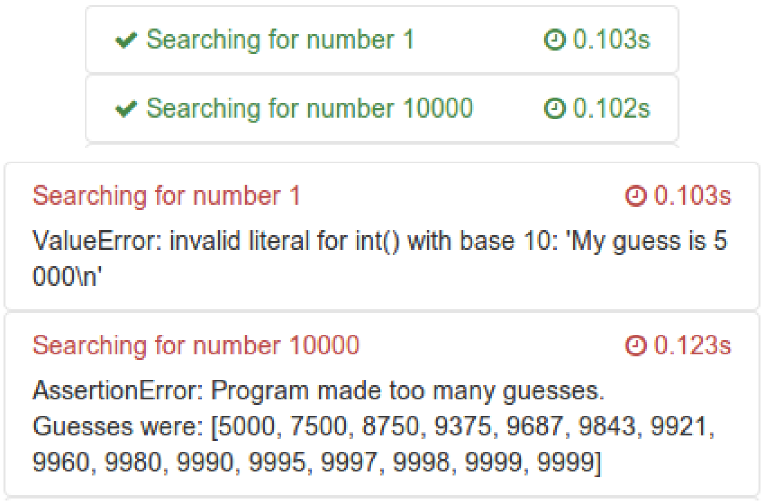
The command outputs JSON containing the feedback for testing *solution.py* with *tester.py*. Another way of running the tester is by importing the module and calling a specific function within the module. See API documentation [11] for method `test_module`.

2.3.5 Feedback results

As mentioned previously, the result of running the tests is a JSON dictionary. The dictionary contains the test description, a boolean representing whether the result was successful, execution time and error message (if any).

While this representation is useful when developing tests on local machine or when doing computer-computer communication, some values such as tracebacks or long error messages are not readable in this format.

When presenting feedback to the user, styled html or a picture should be shown. See Figure 2.1 for an example how feedback for this task is shown in the web IDE (Section 4.3). As shown by P. Ihantola in [12], the feedback can also be in the form of an image - for example in a chess task, the feedback might be an animation of chess moves on a board.



✓ Searching for number 1 0.103s

✓ Searching for number 10000 0.102s

Searching for number 1 0.103s

ValueError: invalid literal for int() with base 10: 'My guess is 5 000\n'

Searching for number 10000 0.123s

AssertionError: Program made too many guesses. Guesses were: [5000, 7500, 8750, 9375, 9687, 9843, 9921, 9960, 9980, 9990, 9995, 9997, 9998, 9999, 9999]

Figure 2.1: Test results with feedback. Error messages of failed tests are also displayed.

2.4 Example - testing functions

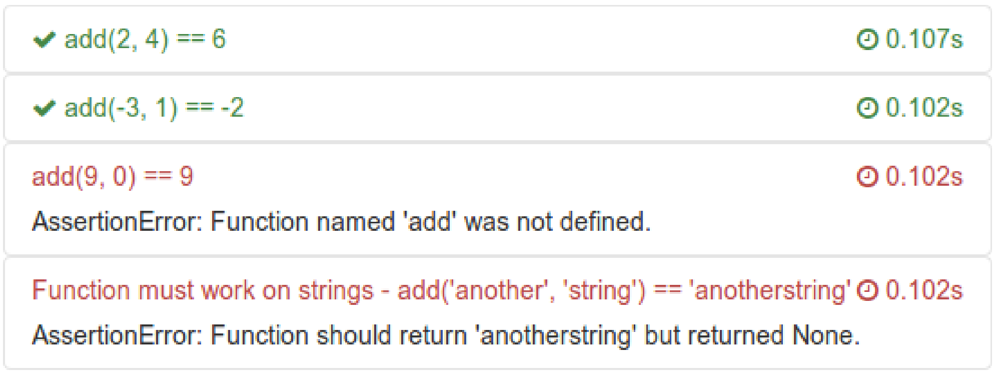
Unlike IO-based tasks, requirements for testing functions are usually very clear and almost always identical - the program must first finish executing, then the tested function is called with some arguments and the returned result is checked against the value returned by a sample solution.

Because of that, the `grader` module provides a **test generator** for these sorts of tasks. Test generators are functions, which dynamically create tests when they are called.

In the example in Listing 2.4, `check_function` generator is used to create 4 tests. Each of those tests first checks if the function named `add` exists and then if it returns the same result as the sample solution on the arguments it was called with. Figure 2.2 shows possible feedback for this task.

```
from grader import *

def add(x, y):
    return x + y

check_function(add, [2, 4]) # test for if add(2, 4) == 6
check_function(add, [-3, 1])
check_function(add, [9, 0])
# for edge cases, a description explaining the case can be made
check_function(add, ["another", "string"],
    description="Function must work on strings")
```

Listing 2.4: Tester for testing function add.



```
✓ add(2, 4) == 6                                                    0.107s
✓ add(-3, 1) == -2                                                  0.102s
add(9, 0) == 9                                                      0.102s
AssertionError: Function named 'add' was not defined.
Function must work on strings - add('another', 'string') == 'anotherstring' 0.102s
AssertionError: Function should return 'anotherstring' but returned None.
```

Figure 2.2: Example feedback from tester in Listing 2.4

2.5 Example - file-based task

The next example deals with testing a program which reads a matrix from a file and writes the transpose of the said matrix to another file.

There are two main options on how to make files available for the tester:

1. It is possible to add extra files in addition to the solution and tester that will be available. This method could be used to attach data files and would allow to use `test_cases` decorator that was used in Section 2.3.3. This option has two negative aspects: it makes invoking the tester harder and the solution must always ask for the file names that are to be processed.
2. Generate the files to be processed just before the solution program is started. This way all information related to the tests would be in one file and it would be easy to modify the task.

As option 2 is more natural to use and more flexible, it will be used for this example.

The module utilizes pre-test hooks (denoted by a decorator `before_test`) which are called right before the solution code and tester code are started. We can use this tool to prepare files to test. Example of this is provided in Listing 2.5.

```
@test # registers test some_test
@before_test(create_file('sampledata.txt', 'hello'))
def some_test(m):
    # sampledata.txt containing 'hello' is available within the test
    ...
```

Listing 2.5: Example of a test using pre-hooks to create files.

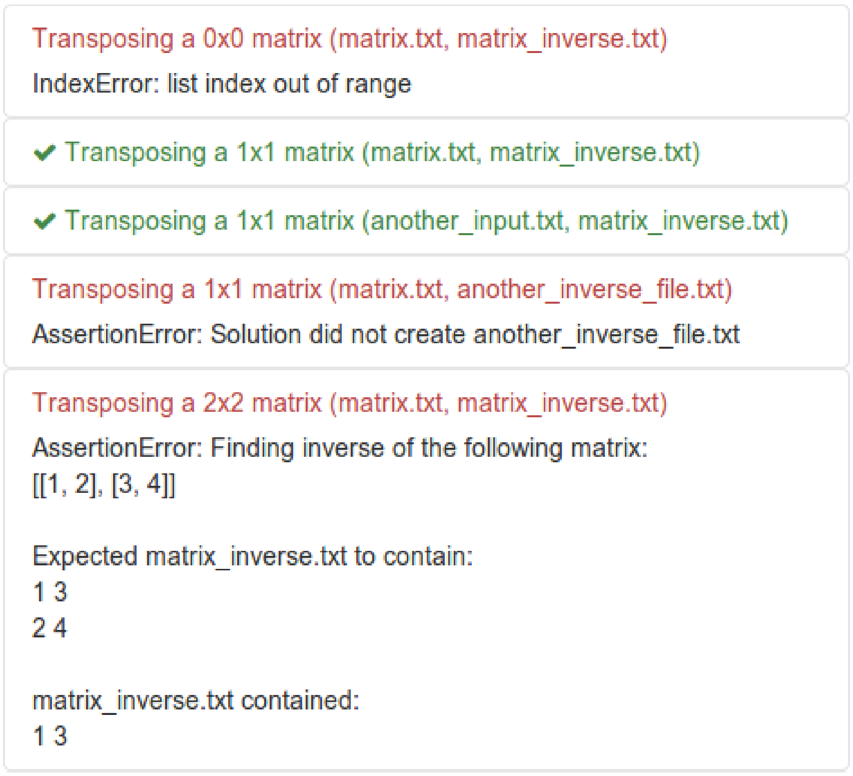
Since files created for each test should have different contents and file names, generating functions based on test data is needed to avoid code duplication. We might generate such functions using a loop, but as Python functions are late-binding [13], this would not work as expected.

A better way to generate test functions is to write our own **test generator** as shown in Listing 2.6 (sample feedback in Figure 2.3).

```
from grader import *
import os

def matrix_gen(matrix,
               infile_name="matrix.txt",
               outfile_name="matrix_inverse.txt"):
```

```
    # Description management for the tests
    rows, cols = size(matrix)
    description = "Transposing a {rows}x{cols} matrix ({inf}, {out})"

    description = description.format(rows=rows,
                                     cols=cols,
                                     matrix=matrix,
                                     inf=infile_name,
                                     out=outfile_name)

    # transform the list of lists into a multi-line string
    infile_contents = stringify(matrix)
    expected_outfile = stringify(transpose(matrix))

    # Internally create a function and register as a test
    # All the file checking logic goes in there.
    # Before the test is executed, the needed file is generated
    @test
    @before_test(create_file(infile_name, infile_contents))
    @set_description(description)
    def _inner_test(m):
        m.stdin.put(infile_name)
        m.stdin.put(outfile_name)

        assert os.path.exists(outfile_name), \
            "Solution did not create "+outfile_name

        with open(outfile_name) as f:
            output = f.read()

        # compare output to expected (ignore trailing whitespace)
        assert output.strip() == expected_outfile, (
            "Finding inverse of the following matrix:\n{matrix}\n\n"
            "Expected {outfile} to contain:\n{expected}\n\n"
            "{outfile} contained:\n{got}"
            .format(got=output, expected=expected_outfile,
                    matrix=matrix, outfile=outfile_name)
        )

# create 6 tests
matrix_gen([])
matrix_gen([[1]])
matrix_gen([[1]], infile_name="another_input.txt")
matrix_gen([[1]], outfile_name="another_inverse_file.txt")
matrix_gen([[1,2], [3, 4]])
matrix_gen([[1,2,3], [4,5,6], [7,8,9], [10,11,12]])
```

Listing 2.6: Code for testing matrix transposing using files. Each of the last 6 lines registers a test each with different input data.



```
Transposing a 0x0 matrix (matrix.txt, matrix_inverse.txt)
IndexError: list index out of range

✓ Transposing a 1x1 matrix (matrix.txt, matrix_inverse.txt)

✓ Transposing a 1x1 matrix (another_input.txt, matrix_inverse.txt)

Transposing a 1x1 matrix (matrix.txt, another_inverse_file.txt)
AssertionError: Solution did not create another_inverse_file.txt

Transposing a 2x2 matrix (matrix.txt, matrix_inverse.txt)
AssertionError: Finding inverse of the following matrix:
[[1, 2], [3, 4]]

Expected matrix_inverse.txt to contain:
1 3
2 4

matrix_inverse.txt contained:
1 3
```

Figure 2.3: Example feedback from tester in Listing 2.6. The first error reported is caused by an exception in solution code.

2.6 Extending the module for new task types

While it is possible to test everything that can be tested with this module using the primitives introduced in the previous examples, it will sometimes lead to code duplication. To avoid that, users of the module need tools to write their own custom generators and decorators to support new task types and also a way to bundle them with this module.

To avoid having to copy often-used generators into each tester, it is possible to add extra files and Python modules as assets when testing. The asset files are placed in the same folder as the tester and solution and thus can be used in testers. Adding those files means adding an extra parameter when invoking the tester via command line.

2.6.1 Improving function testing

Some function-based tasks might have more specific requirements than simply checking return values of functions.

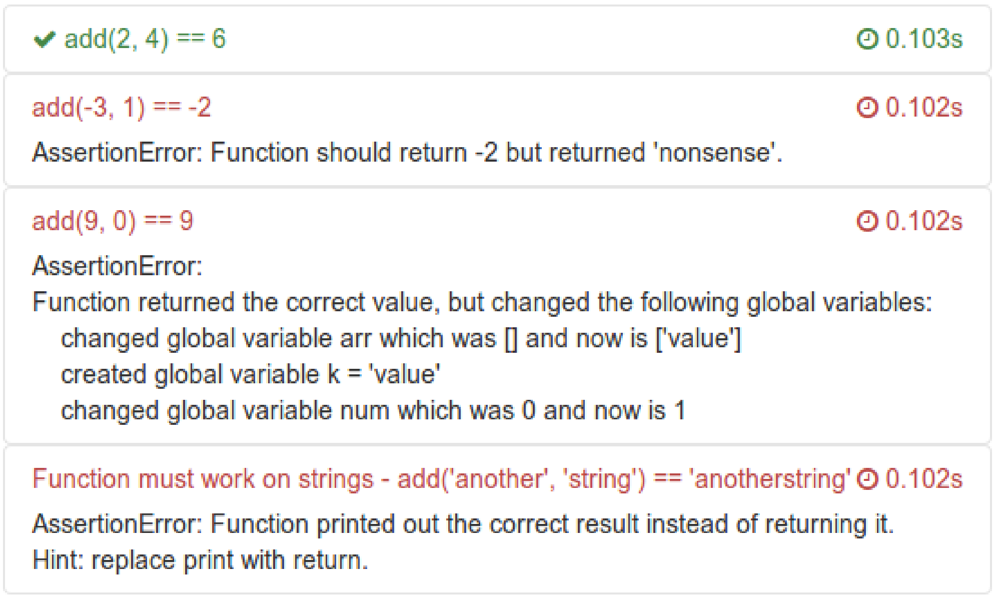
For example, it might be useful to check for common mistakes such as checking if the function did printed the answer instead of returning it, if it modified any global variables and if it returns the a different result when called multiple times.

It is possible to test for these mistakes by writing our own test generator which extends the capabilities of `check_function` used in Section 2.4.

A complete implementation of such a generator is shown in Listing 2.7 (without helper methods) with example feedback in Figure 2.4. It is also available in the module **grader.extensions.adv_functions** [14].

```
def check_function(
        sample_function, arguments,
        expected_result=None,
        description=None,
        # check for print instead of return
        check_print=True,
        # number of times to call the function
        n_calls=3,
        # check if globals change after calling the function.
        check_globals=True):

    # get expected result from function, function name and
    # test description
    fn_name = sample_function.__name__
    if expected_result is None:
        expected_result = sample_function(*arguments)

    description = get_description_string(fn_name,
                                          arguments,
                                          expected_result,
                                          prefix=description)
    # Internally create a function and register as a test
    # All the checking logic goes in there.
    @test
    @set_description(description)
    def _inner_test(m):
        assert m.finished, "Program didn't finish execution"
        assert hasattr(m.module, fn_name), (
          "Function named {} was not defined.".format(repr(fn_name))
        )
        fn = getattr(m.module, fn_name)
        for i in range(1, n_calls + 1):
            start_vars = variables_snapshot(m.module)
            result = fn(*arguments)

            output = m.stdout.read()
            if check_print and result is None and \
                    str(expected_result) in output:
                raise AssertionError(
                    "Function printed out the correct result "
                    "instead of returning it.\n"
                    "Hint: replace print with return."
                )

            # if answer isn't what was expected, raise error
            assert result == expected_result, \
                get_error_description(result, expected_result, i)
            # check if any globals changed
            if check_globals:
                # get changed variables as a dictionary
                end_vars = variables_snapshot(m.module)
                diff = dict_diff(start_vars, end_vars)
                # if any variables changed, raise error accordingly
                assert not diff, globals_error_msg(diff)
```

Listing 2.7: Advanced generator for function testing. Note that instead of testing a function directly `check_function` generates and registers a new test function.



```
✓ add(2, 4) == 6                                                    0.103s

add(-3, 1) == -2                                                    0.102s
AssertionError: Function should return -2 but returned 'nonsense'.

add(9, 0) == 9                                                      0.102s
AssertionError:
Function returned the correct value, but changed the following global variables:
    changed global variable arr which was [] and now is ['value']
    created global variable k = 'value'
    changed global variable num which was 0 and now is 1

Function must work on strings - add('another', 'string') == 'anotherstring' 0.102s
AssertionError: Function printed out the correct result instead of returning it.
Hint: replace print with return.
```

Figure 2.4: Example feedbacks for an add function tested using Listing 2.7.

2.6.2 Fill in the blanks task type

As mentioned in Chapter 1, some tasks require that the student may not use some functions or language constructs in their code.

One variation of such a task is as follows: Students are asked to complete a program given by their instructor, but are only allowed to modify parts of the program.

```
while True:
    number = int(input("Input a number: "))
    if number > _____: # fill in the blank here...
        print("Non-negative!")
    else:
        ... # and here! Can be several lines
```

Listing 2.8: Template for the example task. Students should output "Negative!" and stop if the number is negative otherwise output "Non-negative!" and ask for another number.

Such tasks are tested by comparing the parsed representation of the solution program against the template and checking for differences. As seen from Listing 2.8, the template may contain two different kinds of placeholders:

- Underscores, which should be replaced by a single statement or expression.
- Triple dots, which mean that it should be replaced with 0 or more statements.

To write a tester for this task, we first need access to the abstract syntax tree (AST) of the solution code. This can be added as an argument to a test function using pre-hooks (Chapter 3.3), as the built-in decorator `expose_ast` in Listing 2.9 does.

```
def expose_ast(test_function):
    import ast
    from grader.core import before_test

    def _hook(info):
        code = read_code(info["solution_path"])
        # add the solutions AST as a named argument to the test
            function
        info["extra_kwargs"]["AST"] = ast.parse(code)
    # add function _hook as a pre-hook to test_function
    return before_test(_hook)(test_function)
```

Listing 2.9: Implementation of `expose_ast` decorator.

After having the AST of both the solution and the template, the two need to be compared. We can do that recursively by checking equality of each node. Two special cases for placeholders need to be taken into account however:

- Since nodes of underscores match every other node, we can stop recursion here.
- Triple dots are matched similarly to how the regular expression .* is matched - all possible combinations need to be checked. Note that the current implementation requires exponential number of comparisons based on the number of potential statements matched.

The comparison algorithm and a test generator for this task are implemented in **grader.extensions.ast** submodule and a sample tester can be seen in Listing 2.10.

```
from grader import *
from grader.extensions import ast

template = """
number = int(input("Input a number: "))
if number > _____: # fill in the blank here...
    print("Non-negative!")
else:
    ... # and here! Can be several lines
"""

ast.template_test(template)
```

Listing 2.10: Tester for the example task. Note that additional tests are also needed to verify the behaviour of the program.

Chapter 3

Implementation

This chapter discusses the architecture of the `grader` testing module.The overall structure of the module is discussed, explaining the purpose of different components and how they were built. Last part of the chapter deals with how to secure the module.

3.1 General overview

3.1.1 Test registration

When the tester is started, at first all the tests are registered. For that, the file containing the tests is imported which as a side-effect adds all the functions decorated with `@test` into a dictionary. Later on, the functions can be accessed from the same test case dictionary using the test description. By default the test description is either the function name or documentation string if it exists, but this can be overridden by using the `@set_description` decorator.

In case of the interactive search example in Section 2.3.3, the decorator `test_cases` internally creates 12 functions, all of which are registered as separate tests. Each of those functions in turn call the decorated function `search_tester` with a specific number that is to be searched.

3.1.2 Preparations for test run

Hollingsworth, author of the first known automatic programming grader, made the observation that it is possible for a student to submit a program which undeliberately does some damage to the graders' execution environment [15]. For example, the students program might open files for writing and never close them, making reading them impossible; it might overwrite functions in different modules or even overwrite solution and tester code files.

To avoid such problems, proper isolation is needed. All tested files are copied to a temporary location at the start of each test and the test function itself is run in a separate process. Each such process is only allowed to run for a limited time after which it is automatically killed to avoid issues with infinite loops.

To get access to the correct test function, the new process repeats the test registration procedure and finds the test function with appropriate index that was sent to the process.

A so-called module container object is prepared in a separate thread, which contains references to the faked input and output streams of the tested program. It also contains a reference to the solution module - object which can be used to access all the functions and variables declared in solution program. The module is created by starting the solution program using the `exec` statement, taking care so the resulting module does not have access to the context of tester code.

3.1.3 Test run, results communication

After the preparations are done, the test function is started in the main thread, with the program container passed as an argument to the function. Note that the solution program is already running in parallel in the other thread since `exec` was called.

While running the test function, if an exception is raised in either thread the test is stopped. The exception is then caught by the tester and the error message and traceback are extracted. In case both threads raised an error, the earlier one is reported.

After the test function call is complete, JSON containing both the error message and exception traceback is output. Both are empty strings if no exceptions were raised.

After the test running process exits, the original process reads the JSON output and adds some fields to it like test description, execution time and a flag if the test was a success. Should the test process run too long however, it is automatically killed and the error message set to the string 'Timeout'.

This process is repeated for all tests and a JSON dictionary containing all the test results is output.

3.2 Synchronization

When testing interactive tasks we need to be sure that the solution program has finished their output by the time that the tester tries to read it. Another facet of the same problem is that all the functions and variables must exist by the time that the tester tries to access them, otherwise slow but correct programs might get inaccurate feedback (for example that the tested function or variable does not exist).

To make it easier for the test writer, the `grader` module does implicit synchronization between the solution and tester threads. More specifically, at all times either one or the other is executing, but never the two at once.

This is achieved by using locking - when the solution program is executed, the solution thread acquires the lock, only releasing it when the solution tries to read input or reaches the end of code. It then releases the lock and in the first case waits until the tester has acquired and released it before re-acquiring it.

The tester thread behaves in a similar way - after the solution thread has released the lock for the first time, tester acquires the lock and starts executing the test code. When it reaches a statement that adds something to the input stream of the solution code, it releases the lock and waits until the lock has been acquired and released before re-acquiring or until the solution code finishes executing. Once solution code finishes executing, the locking mechanism is ignored from that point onward.

This approach guarantees that each time the tester tries to read the output of the solution program there will not be any subsequent writes by the solution until the tester adds strings to the solutions input stream.

3.2.1 Tradeoffs

The advantages of this synchronization approach is that it is simple, easy to implement and works well for the tasks in the targeted courses.

However, there are also a number of disadvantages:

1. Testing solutions which have infinite loops and do no reads is impossible. This is because the lock acquired by the solution is never released and the process is automatically killed due to timing out. One example of such a task might be simply displaying a graphical user interface.
2. When testing functions which read from input stream, the input data needs to be prepared before the function is called by the tester. This is because the function call executes in the same thread as the tester code.

3. Testing threaded code in general is a tricky issue since there might be deadlocking issues.

Future revisions of the module might allow for different synchronization strategies if there is demand for it.

3.3 Pre- and postprocessing

There is often a need to add procedures that must be done before a test begins (such as creating files in Section 2.5) or for processing the test result somehow (adding grades for tasks).

```
def pre_hook_example(test_info):
    # add extra argument to the test function
    test_info['extra_args'].append('arg')
    # add extra named argument to the test function
    kwargs = test_info['extra_kwargs']
    kwargs['keyword_arg'] = 'keyword_arg'

def post_hook_example(results):
    # add points to results
    if results["success"]:
        results["points"] = 1
    else:
        results["points"] = 0

@test
@before_test(pre_hook_example)
@after_test(post_hook_example)
def some_test(m, arg, keyword_arg):
    ...
```

Listing 3.1: Example of using pre- and post-hooks. The `before_test` and `after_test` decorators add the function which is its argument as a pre- and post-hook respectively.

The module provides **pre-** and **post-hooks** for these purposes:

- Pre-hooks are executed right before the test function is called. Each pre-hook gets passed various information about the test such as the solution file path, test name and extra arguments list and keyword argument dictionary it may modify.
- Post-hooks get called in the main process after a test has finished executing. It gets passed in a dictionary with a test result which it may modify.

3.4 Security

The goal of the `grader` module is to allow testing of untrusted and unreviewed code submitted by students. This poses a challenge since a student might want to break into the system or otherwise disturb the behaviour of the application using the module. Also - beginners might submit solutions inadvertently which do harm to the tester system.

By default the `grader` module uses no sandbox. This makes it easier to install and use the library and also makes it possible to use the module on non-Linux machines. However, this chapter shows how to use a Docker sandbox to secure the module.

3.4.1 Requirements, threat model

When running a public service such as a web server which uses the module, it should have a few desired properties such as:

- **Isolation:** if multiple solutions are being tested at the same time, they should not be able to interfere with each other or with the system. Ideally, they should not even be aware of other processes running on the host machine.
- **Resource limiting:** each solution being tested should only be able to use a fixed proportion of the hosts CPU, memory and disk space.
- **Access limitations:** the tested program should not have access to the network and files which it does not need.
- **Cleanup:** all side-effects of testing code should be automatically be removed. This includes reverting edited files.
- **Easy to install:** used sandbox should be easy to install, maintain and hard to mis-configure.

Note that these requirements do not deal with correctness of test results.

Historically, various techniques have been used to deal with remote code execution. Chroot() jails with additional resource limiting has been used by many online judges [6] [5] and by HackerRank [16].This solution has one disadvantage in the author's opinion - namely it is hard to set up and maintain correctly.

3.4.2 Docker

The `grader` module can be sandboxed using Docker containers in Linux enviroments.

Docker [17] is an abstraction built on top of Linux Containers (LXC) and cgroups and it allows the creation and teardown of complete Linux environments (containers) within a fraction of a second. The created containers share the host systems kernel, but have their own isolated filesystem, process tree and networking stack. Appropriately, LXC has been described as "chroot on steroids" [18].

Docker has been used to secure large programming competitions such as Stripe CTF 3 [19] and by various code running services such as `codecube.io` [20].

3.4.3 Securing module using Docker

Docker achieves isolation by creating a fresh container from a base image each time a solution is tested. For efficiency reasons, all tests on a single solution share the same sandbox. In addition to operating system essentials, the created container has only Python and the `grader` module installed. Also because each container has their own process tree and file system, the tested file cannot be aware of other tested solutions.

When the container is started, memory and CPU limits are automatically set and networking is also turned off within the container.

After testing is done, the created container is automatically deleted. This means that each time tests are run, the testing environment is in exactly the same condition as described in the docker image.

Installing docker is described in Appendix A, but it should be noted that this requires a recent Linux kernel (version 3.8 and above) and since it requires LXC support, it excludes some operating systems like OpenBSD.

Docker registry is currently used to host docker image files. This means that updating the sandbox is a single command, but the image can also be manually built. The image is also automatically updated each time the grader source repository is changed.

3.4.4 Using sandboxes with `grader` module

When starting the grader via command line, it is possible to use docker sandbox as follows:

```
python -m grader --sandbox docker path/tester.py path/solution.py
```

Installation instructions are in Appendix A.

It is also possible to use some other sandbox with the grader by creating a script which accepts two arguments - tester path and solution path and outputs testing results in the same format as a normal call to grader module would. The tester and solution path provided are in a temporary directory which also contains other asset files passed to the module.

Chapter 4

Usage in courses

This chapter discusses how the tester was used in courses at the University of Tartu, experiences learned from those attempts and also ideas for future iterations of the module.

4.1 MTAT.03.100 Computer Programming

The created module was first used in autumn 2013 for automatically testing midterms and tests for over 300 students in this course.

Tests were written by the author and his supervisor Aivar Annamaa. The tests were run on a virtual machine. Results of the tests were published as a web page accessible to the lab assistants who used it to grade the submissions.

In general the lab assistants found the automated feedbacks useful when grading solutions and said it saved them a lot of time. Most if not all lab assistants used the automatic feedback generated for grading all three assessments that semester. There were some slight problems with file encodings when downloading tasks from Moodle and due to this some solutions did not receive proper feedback from the tester. Later revisions of the testing scripts fixed this problem.

Feedback for the solutions was presented in the form of a traceback. This was found to be mostly confusing, since usually only the final error message contained relevant information. The traceback did however help lab assistants to give feedback for solutions which were unfinished or buggy and exited with exceptions, since they did not need to download the tasks and run them manually.

A common problem was that students often tried to solve a slightly different problem than the task specification described. For example, students might ask for arguments from standard input on function-based tasks, mix up the order of arguments or ask for a file name when it was fixed in task description.

The author believes those problems stem from four main issues:

1. Homeworks were not automatically tested so the grading criteria differed between labs and tests and were probably more lax in the first.
2. Students did not have previous experience using automatic tests, leading them to make small mistakes the tester stumbled on.
3. Time pressure left students less time to properly finish their solutions.
4. Predicting what mistakes will be made is hard before manually checking some solutions first. For example, testers for many midterm tasks were revised before announcing results.

Two of the midterms also included tasks involving using the `turtle` module to draw pictures on the screen. These were not automatically graded, but an animation was made using a script made by the author [21]. As this solution requires a windowing system being used on the testing machine and is resource intensive, this solution will likely not be used when testing tasks on the server.

4.2 MTAT.03.256 Introduction to Programming II

At the time of writing, the module is being used to provide automatic feedback for some homework tasks through Moodle study environment.

4.2.1 Integration with the Moodle study environment

In Moodle, programming tasks can be assessed by using Moodle Virtual Programming Lab plugin [2]. As mentioned in Section 1.2.1, by default this system only supports IO based testing. It is, however, possible to use custom grading scripts which output a grade and custom feedback shown to the student.

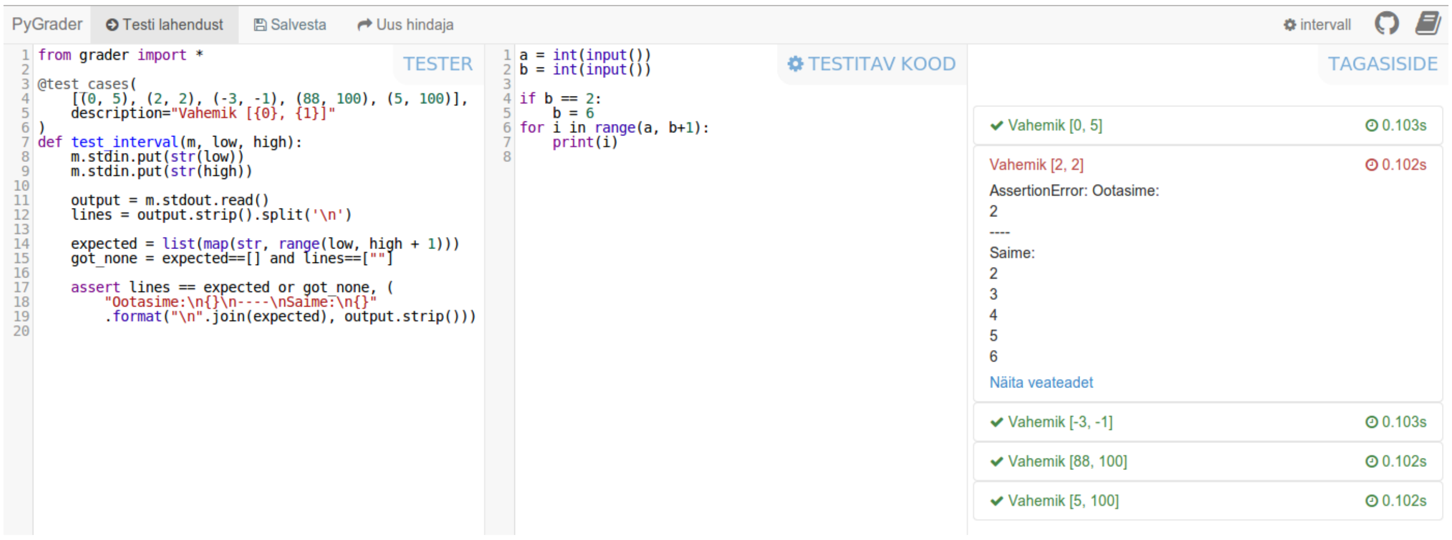
To use `grader` module for that purpose, it needs to be installed (without docker) onto a jail server which runs the code for VPL. Next a Python script can be written which uses the `test_module` function within `grader` module to assess a solution and outputs results in format described in [22].

Grades are also calculated by weighing tests (with weights automatically set to ones) and counting successful test weights towards the maximum grade set in Moodle.

4.3 Web IDE

A small web IDE for test development was also created by the author [23]. The goal was to present the tester code, sample solution and formatted task feedback on one page, where the tasks can be shared and edited.

The goal of the IDE is to make writing testers more accessible for teachers who would not need to install anything on their machines. It can also be used for introducing the module to potential users.



```
from grader import *

@test_cases(
    [(0, 5), (2, 2), (-3, -1), (88, 100), (5, 100)],
    description="Vahemik [{0}, {1}]"
)
def test_interval(m, low, high):
    m.stdin.put(str(low))
    m.stdin.put(str(high))

    output = m.stdout.read()
    lines = output.strip().split('\n')

    expected = list(map(str, range(low, high + 1)))
    got_none = expected==[] and lines==[""]

    assert lines == expected or got_none, (
        "Ootasime:\n{}\n----\nSaime:\n{}"
        .format("\n".join(expected), output.strip()))

```

```
a = int(input())
b = int(input())

if b == 2:
    b = 6
for i in range(a, b+1):
    print(i)

```

```
✓ Vahemik [0, 5]        0.103s
Vahemik [2, 2]          0.102s
AssertionError: Ootasime:
2
----
Saime:
2
3
4
5
6
Näita veateadet
✓ Vahemik [-3, -1]      0.103s
✓ Vahemik [88, 100]     0.102s
✓ Vahemik [5, 100]      0.102s
```

Figure 4.1: Screenshot of the web IDE

4.4 Future development

As the module is planned to be used in autumn 2014, development of the module will not stop. This section discusses some long- and short term ideas surrounding the module.

- Further support for ast searching and manipulation. This would allow to support more limited tasks. The Python library macropy [24] could possibly be used to enhance AST searching support.
- Currently the built-in function generators output feedback in English, however the all the courses currently using the module are taught in Estonian. This means that some kind of internalization support is needed from the module.
- Good error messages and test descriptions are essential for the student to understand what went wrong with their solution. Proper guidelines for these are needed.
- Integration with programming book by Aivar Annamaa [25].
- Test parallelization.
- GUI testing support. This would mostly require for the testing servers to support windowing systems.

Conclusion

As a result of this thesis, a Python module was created which allows the testing of typical assessments in introductory programming courses. The module allows the teacher to test tasks which mix functions and input-output based tasks as well as other task types and can be used within existing grading systems such as Moodle Virtual Programming Lab.

This thesis serves as an introduction to the `grader` module and contains examples on how to use and extend it for testing new task types. It also contains analysis of existing systems, design decisions of the module, how the module works and how it can be secured.

At the time of writing, the module has been used in two courses and testers for over 40 tasks are freely available in its repository. The last part of the thesis introduced how the tester was used in those courses and various experiences learned from it.

Bibliography

[1] Cynthia J. Solomon and Seymour Papert. A case study of a young child doing turtle graphics in logo. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, pages 1049–1056, New York, NY, USA, 1976. ACM.

[2] Moodle virtual programming lab. `http://vpl.dis.ulpgc.es/` (26.04.2014).

[3] Mooshak contest sytem. `https://mooshak.dcc.fc.up.pt/` (26.04.2014).

[4] Sphere online judge. `http://www.spoj.com/info/` (26.04.2014).

[5] Adrian Kosowski, Michal Malafiejski, and Tomasz Noinski. Application of an online judge & contester system in academic tuition. In Howard Leung, Frederick Li, Rynson Lau, and Qing Li, editors, *Advances in Web Based Learning - ICWL 2007*, volume 4823 of *Lecture Notes in Computer Science*, pages 343–354. Springer Berlin Heidelberg, 2008.

[6] Jordi Petit, Omer Giménez, and Salvador Roura. Jutge.org: An educational programming judge. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 445–450, New York, NY, USA, 2012. ACM.

[7] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[8] Unittest testing module. `https://docs.python.org/3/library/unittest.html` (26.04.2014).

[9] Flask web framework. `http://flask.pocoo.org/` (26.04.2014).

[10] nose testing module. https://nose.readthedocs.org/en/latest/ (26.04.2014).

[11] API documentation for grader module. https://macobo.github.io/python-grader/.

[12] Petri Ihantola. Automated assessment of programming assignments: Visual feedback, assignment mobility, and assessment of students' testing skills. 2011.

[13] Python guide - late binding. http://docs.python-guide.org/en/latest/writing/gotchas/#late-binding-closures (27.04.2014).

[14] grader module. https://github.com/macobo/python-grader.

[15] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, October 1960.

[16] Quora answera bout how HackerRank is secured. http://qr.ae/rg6vp (28.04.2014).

[17] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[18] Björn Pehrson. Virtualization analysis - CSD Fall, 2011.

[19] Blog post about how stripe capture the flag 3 competition testing architecture. https://stripe.com/blog/ctf3-architecture (28.04.2014).

[20] Blog post about how codecube.io is secured. http://hmarr.com/2013/oct/16/codecube-runnable-gists/ (28.04.2014).

[21] Turtlesnap module used to make pictures of turtle module. https://github.com/macobo/TurtleSnap.

[22] Moodle vpl documentation. http://vpl.dis.ulpgc.es/index.php/en/documentation/76-general-documentation (27.04.2014).

[23] Grader web ide source code. https://github.com/macobo/grader-webapp.

[24] Macropy module. https://github.com/lihaoyi/macropy.

[25] Aivar Annamaa. *Programming textbook*. http://programmeerimine.cs.ut.ee.

Appendix A

Installation guide

Prerequsites

- Python 3.4 or above installed.
- (for sandbox) Linux operating system supporting Linux Containers.
- (for sandbox) Linux kernel version ≥ 3.8.

`grader` module installation

- Clone repository at `https://github.com/macobo/python-grader`.
- Run in the cloned folder: `python3 setup.py install`

Docker sandbox installation

- Install docker: `https://www.docker.io/gettingstarted`.
- Download latest sandbox image: `docker pull macobo/python-grader-sandbox`

The sandbox image can also be built manually when within the cloned grader folder:

```
docker build -t macobo/python-grader-sandbox .
```