MoodleTeacher Documentation

Release 0.1.18

Peter Tröger

Apr 19, 2020

Contents

This Python library is intended for teachers with courses in the Moodle learning management system. They can easily automate their grading or course management procedures, so that clicking around on the web page is no longer neccessary.

CHAPTER 1

Getting Started

These instructions will get you the library up and running on your local machine.

1.1 Installation

The library demands Python 3. Install the software with:

```
pip3 install moodleteacher
```

1.2 Playing around

Start an interactive Python session and load the library:

```
(venv) shaw:moodleteacher ptroeger$ python
Python 3.6.5 (default, Apr 25 2018, 14:23:58)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import moodleteacher as mt
```

First, you need to provide the neccessary connection information. The host name is the web server where your Moodle installation lives. The token can be retrieved from your Moodle user settings (Preferences -> User account -> Security keys -> Service “Moodle mobile web service”):

```
>>> conn=mt.MoodleConnection(moodle_host="https://lms.beuth-hochschule.de", token=
↪"aaaaabbbbbcccccdddddeeeee12345")
```

You can now fetch the list of assignments that are accesssible for you:

```
>>> assignments=mt.MoodleAssignments(conn)
>>> for assign in assignments:
```

(continues on next page)

(continued from previous page)

```
...     print(assign)
...
```

You can filter for particular assignments or courses, based on the Moodle IDs for them:

```
>>> assignments=mt.MoodleAssignments(conn, course_filter=[4711], assignment_
↪filter=[1234])
>>> for assign in assignments:
...     print(assign)
...
```

Assignment objects provide a list of submissions. Each submission object provides the files (or text) submitted by the student:

```
>>> for assignment in assignments:
...     for submission in assignment.submissions():
...         print("User {0} submitted {1} files.".format(submission.userid,␣
↪len(submission.files)))
```

Student file uploads can be downloaded with the *moodleteacher.submissions.MoodleSubmission* class and previewed with a small integrated GUI application. The preview supports:

- HTML text
- PDF files
- Images
- Any other content, just shown in text form
- ZIP files of any of the above

Here is an example for using the preview:

```
>>> for assignment in assignments:
...     for submission in assignment.submissions():
...             for file_url in submission.files:
...                     print(file_url)
...
https://lms.beuth-hochschule.de/webservice/pluginfile.php/725647/assignsubmission_
↪file/submission_files/32245/task03_ft.pdf
https://lms.beuth-hochschule.de/webservice/pluginfile.php/725647/assignsubmission_
↪file/submission_files/75356/Fehlerbaum.jpg
https://lms.beuth-hochschule.de/webservice/pluginfile.php/725647/assignsubmission_
↪file/submission_files/23454/Faultchar%2B-fertig.png

>>> stud_upload=mt.MoodleSubmissionFile(conn=conn, url="https://lms.beuth-hochschule.
↪de/webservice/pluginfile.php/725647/assignsubmission_file/submission_files/75356/
↪Fehlerbaum.jpg")
>>> stud_upload.is_pdf
False
>>> stud_upload.is_image
True
>>> from moodleteacher import preview
>>> mt.preview.show_preview("Preview Window", [stud_upload])
```

Submissions can be trivially graded:

```
>>> for assignment in assignments:
...     for submission in assignment.submissions:
...         submission.save_grade(0.0, "Everybody fails in this assignment. You too.")
```

CHAPTER 2

Library Reference

2.1 moodleteacher.assignments

Functionality dealing with Moodle assignments.

class moodleteacher.assignments.**MoodleAssignment**(*course*, *assignment_id*, *course_module_id=None*, *duedate=None*, *cutoffdate=None*, *deadline=None*, *name=None*, *allows_feedback_comment=None*)

A single Moodle assignment.

classmethod from_assignment_id(*course*, *assignment_id*)

Create a `MoodleAssignment` object just from an assignment ID.

classmethod from_course_module_id(*course*, *course_module_id*)

Create a `MoodleAssignment` object just from a course module ID.

classmethod from_raw_json(*course*, *raw_json*)

Create a `MoodleAssignment` object from raw JSON information.

get_user_submission(*user_id*, *must_have_files=False*)

Create a new `MoodleSubmission` object with the submission of the given user in this assignment, or None.

When must_have_files is set to True, only submissions with files are considered.

submissions(*must_have_files=False*)

Get a list of `MoodleSubmission` objects for this assignment.

class moodleteacher.assignments.**MoodleAssignments**(*conn*, *course_filter=None*, *assignment_filter=None*)

A list of `MoodleAssignment` instances.

2.2 moodleteacher.compiler

Functions dealing with the compilation of code.

`moodleteacher.compiler.`**`compiler_cmdline`**(*compiler=['gcc', '-o', '{output}', '{inputs}'], output=None, inputs=None*)

Determine the command-line to be run for a compiler execution.

Parameters

- **`compiler`** (*`tuple`*) – A compiler definition. Predefined values are GCC, GPP, and JAVAC.
- **`output`** (*`str`*) – The file path for the compiler output.
- **`inputs`** (*`tuple`*) – A list of input files for the compiler.

2.3 moodleteacher.connection

A connection to a chosen Moodle server.

`class` `moodleteacher.connection.`**`MoodleConnection`**(*moodle_host=None, token=None, interactive=False, is_fake=False, timeout=5*)

A connection to a Moodle installation.

`__init__`(*moodle_host=None, token=None, interactive=False, is_fake=False, timeout=5*)

Configures a connection to a Moodle server.

Parameters

- **`moodle_host`** – The base URL of the Moodle installation.
- **`token`** – The client security token for the Moodle API access.
- **`interactive`** (*`bool`*) – Prompt interactively for parameters, if needed.
- **`is_fake`** (*`bool`*) – Create fake connection for testing purposes.
- **`timeout`** (*`int`*) – Timeout for HTTP requests.

2.4 moodleteacher.courses

Functionality to deal with Moodle courses.

`class` `moodleteacher.courses.`**`MoodleCourse`**(*conn, course_id, fullname='', shortname=''*)

A single Moodle course.

`classmethod from_course_id`(*conn, course_id*)

Create a *`MoodleCourse`* object just from a course ID.

`classmethod from_raw_json`(*conn, raw_json*)

Create a MoodleCourse object from raw JSON information.

`get_folders`()

Determine folders that are part of the course.

Returns List of `MoodleFolder` objects.

get_group(*group_id*)
Determines the user group for a given group ID.

Returns `MoodleGroup` object for this user id, or None if not known.

get_user(*user_id*)
Determine the user for a given user ID.

Returns `MoodleUser` object for this user id, or None if not known.

get_user_grades(*user_id*)
Fetch grade table for this user, or all users, in the given course.

2.5 moodleteacher.exceptions

Defintion of common exceptions in MoodleTeacher.

exception moodleteacher.exceptions.**JobException**(*info_student=None*, *info_tutor=None*)
An exception that occured while using the Job API.

exception moodleteacher.exceptions.**NestedException**(*instance*, *real_exception*, *output=None*)
An exception occured while running the student program.

exception moodleteacher.exceptions.**NoFilesException**(*info_student=None*, *info_tutor=None*)
Indication that the student submission contains no files.

exception moodleteacher.exceptions.**RunningProgramException**(*instance*, *output=None*)
A problem that occured while running a student program.

Parameters

- **instance** (`RunningProgram`) – The RunningProgram instance raising this issue.
- **output** (*str*) – All the output data being produced so far.

exception moodleteacher.exceptions.**TerminationException**(*instance*, *real_exception*, *output=None*)

exception moodleteacher.exceptions.**TimeoutException**(*instance*, *real_exception*, *output=None*)

exception moodleteacher.exceptions.**ValidatorBrokenException**(*info_student=None*, *info_tutor=None*)
Indication that the validator script is broken.

exception moodleteacher.exceptions.**WrongExitStatusException**(*instance*, *expected*, *got=None*, *output=None*)

2.6 moodleteacher.runnable

class moodleteacher.runnable.**RunningProgram**(*name*, *arguments=[]*, *working_dir='.'*, *timeout=30*, *encoding=None*)
A running program that you can interact with.

This class is a thin wrapper around the functionality of pexpect (http://pexpect.readthedocs.io/en/stable/overview.html).

name
The name of the binary that is executed.

Type str

working_dir
The working directory to be used during execution.

Type str

arguments
The command-line arguments being used for execution.

Type tuple

__init__(*name*, *arguments=[]*, *working_dir='.'*, *timeout=30*, *encoding=None*)
Initialize a running program.

Parameters

- **name** – The file path for the executable.
- **arguments** – The command-line arguments for the executable.
- **working_dir** – The current working directory when running the program.
- **timeout** – The timeout for program execution.
- **encoding** – The text encoding for the program output, e.g. 'utf-8'. If this parameter is not set, then the output is interpreted as bytes.

expect_end()
Wait for the running program to finish.

Returns A tuple with the exit code, as reported by the operating system, and the output produced.

expect_exitstatus(*exit_status*)
Wait for the running program to finish and expect some exit status.

Parameters **exit_status** (*int*) – The expected exit status.

Raises `WrongExitStatusException` – The produced exit status is not the expected one.

expect_output(*pattern*, *timeout=-1*)
Wait until the running program performs some given output, or terminates.

Parameters

- **pattern** – The pattern the output should be checked for.
- **timeout** (*int*) – How many seconds should be waited for the output.

The pattern argument may be a string, a compiled regular expression, or a list of any of those types. Strings will be compiled into regular expressions.

Returns The index into the pattern list. If the pattern was not a list, it returns 0 on a successful match.

Return type int

Raises

- `TimeoutException` – The output did not match within the given time frame.
- `TerminationException` – The program terminated before producing the output.
- `NestedException` – An internal problem occured while waiting for the output.

get_exitstatus()
Get the exit status of the program execution.

Returns

Exit status as reported by the operating system, or None if it is not available.

Return type int

get_output()
Get the program output produced so far.

Returns Program output as text. May be incomplete.

Return type str

sendline(*text*)
Sends an input line to the running program, including os.linesep.

Parameters **text** (`str`) – The input text to be send.

Raises

- `TerminationException` – The program terminated before / while / after sending the input.
- `NestedException` – An internal problem occured while waiting for the output.

2.7 moodleteacher.submissions

class moodleteacher.submissions.**MoodleSubmission**(*conn=None, submission_id=None, assignment=None, user_id=None, group_id=None, status=None, gradingstatus=None, textfield=None, files=[]*)

A single student submission in Moodle.

classmethod from_local_file(*assignment, fpath*)
Creation of a local-only fake submission object. Mainly needed for the test suite.

load_feedback()
Retreives the current feedback for this submission from the Moodle server.

parse_plugin_json(*raw_json*)
Parses a plugin block from Moodle JSON and updates the object accordingly.

save_feedback(*feedback*)
Saves new feedback information on the Moodle server.

See also https://moodle.org/mod/forum/discuss.php?d=384108.

save_grade(*grade, feedback=None*)
Saves new grading information for this student on the Moodle server, and sets the workflow state to "graded".

2.8 moodleteacher.validation

Implementation of validation jobs.

class `moodleteacher.validation.`**`Job`**(*submission*, *validator_file*, *preamble*)

A validation job checks a single student submission, based on a validator script written by the tutor.

Check the validation section in the moodleteacher documentation for more details.

`__init__`(*submission*, *validator_file*, *preamble*)

Prepares a validation job by putting all relevant files into a temporary directory.

`submission`

The student submission object.

Type *MoodleSubmission*

`validator_file`

The validator file object.

Type MoodleFile

`preamble`

The preamble text for each feedback message targeting students.

Type str

`ensure_files`(*filenames*)

Checks the student submission for specific files.

Parameters **`filenames`** (*`tuple`*) – The list of file names to be checked for.

Returns Indicator if all files are found in the student archive.

Return type bool

`grep`(*regex*)

Scans the student files for text patterns.

Parameters **`regex`** (*`str`*) – Regular expression used for scanning inside the files.

Returns Names of the matching files in the working directory.

Return type tuple

`prepare_student_files`(*remove_directories=True*, *recode=False*)

Unarchive student files in temporary directory.

Parameters

- **`remove_directories`** (*`boolean`*) – When the student submission is an archive, remove all contained directories and unpack flat. This makes sense for all but Java code. When the student submission is not an archive, this flag has no effect.
- **`recode`** (*`boolean`*) – Recode the submission files to UTF-8 text, to avoid compiler problems. When the student submission is an archive, this flag has no effect.

`run_build`(*compiler=['gcc', '-o', '{output}', '{inputs}'], inputs=None, output=None, timeout=30*)

Combined call of 'configure', 'make' and the compiler.

The success of 'configure' and 'make' is optional. The arguments are the same as for run_compiler.

`run_compiler`(*compiler=['gcc', '-o', '{output}', '{inputs}'], inputs=None, output=None, timeout=30*)

Runs a compiler in the working directory.

Parameters

- **`compiler`** (*`tuple`*) – The compiler program and its command-line arguments, including placeholders for output and input files.
- **`inputs`** (*`tuple`*) – The list of input files for the compiler.

- **`output`** (*`str`*) – The name of the output file.

`run_configure`(*mandatory=True*, *timeout=30*)
Runs the 'configure' program in the working directory.

Parameters **`mandatory`** (*`bool`*) – Throw exception if 'configure' fails or a 'configure' file is missing.

`run_make`(*mandatory=True*, *timeout=30*)
Runs the 'make' program in the working directory.

Parameters **`mandatory`** (*`bool`*) – Throw exception if 'make' fails or a 'Makefile' file is missing.

`run_program`(*name*, *arguments=[]*, *timeout=30*, *encoding=None*)
Runs a program in the working directory to completion.

Parameters

- **`name`** (*`str`*) – The name of the program to be executed.
- **`arguments`** (*`tuple`*) – Command-line arguments for the program.
- **`timeout`** (*`int`*) – The timeout for execution.
- **`encoding`** (*`str`*) – The text encoding for the program output, e.g. 'utf-8'. If this parameter is not set, then the output is interpreted as bytes.

Returns A tuple of the exit code, as reported by the operating system, and the output produced during the execution.

Return type tuple

`send_fail_result`(*info_student*, *info_tutor='Test failed.'*)
Reports a negative result for this validation job.

Parameters

- **`info_student`** (*`str`*) – Information for the student(s)
- **`info_tutor`** (*`str`*) – Information for the tutor(s)

`send_pass_result`(*info_student='All tests passed. Awesome!'*, *info_tutor='All tests passed.'*)
Reports a positive result for this validation job.

Parameters

- **`info_student`** (*`str`*) – Information for the student(s)
- **`info_tutor`** (*`str`*) – Information for the tutor(s)

`spawn_program`(*name*, *arguments=[]*, *timeout=30*, *encoding=None*)
Spawns a program in the working directory.

This method allows the interaction with the running program, based on the returned RunningProgram object.

Parameters

- **`name`** (*`str`*) – The name of the program to be executed.
- **`arguments`** (*`tuple`*) – Command-line arguments for the program.
- **`timeout`** (*`int`*) – The timeout for execution.
- **`encoding`** (*`str`*) – The text encoding for the program output, e.g. 'utf-8'. If this parameter is not set, then the output is interpreted as bytes.

Returns An object representing the running program.

Return type *RunningProgram*

`start`(*log_level=20*)

Execute the validate() method in the validator script belonging to this job.

CHAPTER 3

Validation tutorial

MoodleTeacher supports the automated validation of student submissions through a custom Python3 script, called a **validator**, which is written by the teacher.

A validator can come in two flavours:

- As single Python file named *validator.py*.
- As ZIP / TGZ archive with an arbitrary name, which must contain a file named *validator.py*.

The second option allows you to use additional files during the validation, such as profiling tools, libraries, or simply code not written by students.

The validation functionality of MoodleTeacher performs the following activities for you:

- Creation of a temporary working directory.
- Download (and unpacking) of the student submission in this directory.
- Download (and unpacking) of the validator in this directory.
- Execution of the validator.
- Reporting of results to Moodle, explicitely from the validator or implicitly.
- Cleanup of the temporary directory.

A validator script can use special functionality from the `moodleteacher.validation.Job` class, which includes:

- Test for mandatory files in the student package.
- Compilation of student code.
- Execution of student code, including input simulation and output parsing.
- Reporting of validation results as teacher feedback in Moodle.

Examples for validators can be found online.

Our companion project MoodleRunner wraps the validation functionality in a Docker image that is directly usable with your existing Moodle installation.

3.1 How to write a validator

We illustrate the idea with the following walk-through example:

Students get the assignment to create a C program that prints 'Hello World' on the terminal. The assignment description demands that they submit the C-file and a *Makefile* that creates a program called *hello*. The assignment description also explains that the students have to submit a ZIP archive containing both files.

Your job, as the assignment creator, is now to develop the `validator.py` file that checks an arbitrary student submission. Create a fresh directory that only contains an example student upload and the validator file:

```
1  def validate(job):
2      job.prepare_student_files(remove_directories=True)
3      job.run_make(mandatory=True)
4      exit_code, output = job.run_program('./hello')
5      if output.strip() == "hello world":
6          job.send_pass_result("The world greets you! Everything worked fine!")
7      else:
8          job.send_fail_result("Wrong output: " + output)
```

The `validator.py` file must contain a function `validate(job)` that is called by MoodleTeacher when a student submission should be validated. In the example above, this function performs the following steps for testing:

- Line 1: The validator function is called when all student files (and all files from the validator archive) are unpacked in a temporary working directory on the test machine. In case of name conflicts, the validator files always overwrite the student files.
- Line 3: The *make* tool is executed in the working directory with *`run_make()`*. This step is declared to be mandatory, so the method will throw an exception if *make* fails.
- Line 4: A binary called *hello* is executed in the working directory with the helper function *`run_program()`*. The result is the exit code and the output of the running program.
- Line 5: The generated output of the student program is checked for some expected text.
- Line 6: A positive validation result is sent back to Moodle with *`send_pass_result()`*.
- Line 7: A negative validation result is sent back to Moodle with *`send_fail_result()`*.

Validators are ordinary Python code, so beside the functionalities provided by the job object, you can use any Python functionality. The example shows that in Line 4.

If any part of the code leads to an exception that is not catched inside `validate(job)`, than this is automatically interpreted as negative validation result. The MoodleTeacher code forwards the exception as generic information to the student. If you want to customize the error reporting, catch all potential exceptions and use your own call of *`send_fail_result()`* instead.

3.2 Validator examples

The following example shows a validator for a program in C that prints the sum of two integer values. The values are given as command line arguments. If the wrong number of arguments is given, the student code is expected to print *"Wrong number of arguments!"*. The student only has to submit the C file.

```
from moodlerunner.compiler import GCC

test_cases = [
    [['1', '2'], '3'],
```

(continues on next page)

(continued from previous page)

```
5      [['-1', '-2'], '-3'],
6      [['-2', '2'], '0'],
7      [['4', '-10'], '-6'],
8      [['4'], 'Wrong number of arguments!'],
9      [['1', '1', '1'], 'Wrong number of arguments!']
10 ]
11
12 def validate(job):
13     job.prepare_student_files(remove_directories=True)
14     job.run_compiler(compiler=GCC, inputs=['sum.c'], output='sum')
15     for arguments, expected_output in test_cases:
16         exit_code, output = job.run_program('./sum', arguments)
17         if output.strip() != expected_output:
18             job.send_fail_result("Oops! That went wrong! Input: " + str(arguments) +
   ↪", Output: " + output, "Student needs support.")
19             return
20     job.send_pass_result("Good job! Your program worked as expected!", "Student seems␣
   ↪to be capable.")
```

- Line 1: The *GCC* tuple constant is predefined in `moodleteacher.compiler` and refers to the well-known GNU C compiler. You can also define your own set of command-line arguments for another compiler.
- Line 3-10: The variable *test_cases* consists of the lists of inputs and the corresponding expected outputs.
- Line 14: The C file can be compiled directly by using `run_compiler()`. You can specify the used compiler as well as the names of the input and output files.
- Line 15: The for-loop is used for traversing the *test_cases*-list. It consists of tuples which are composed of the arguments and the expected output.
- Line 16: The arguments can be handed over to the program through the second parameter of the `run_program()` method. The former method returns the exit code as well as the output of the program.
- Line 17: It is checked if the created output equals the expected output.
- Line 18: If this is not the case an appropriate negative result is sent to the student and teacher with `send_fail_result()`
- Line 19: After a negative result is sent there is no need for traversing the rest of the test cases, so the *validate(job)* function can be left.
- Line 20: After the traversal of all test cases, the student and teacher are informed that everything went well with `send_pass_result()`

The following example shows a validator for a C program that reads an positive integer from standard input und prints the corresponding binary number.

```
from moodlerunner.exceptions import TerminationException

test_cases = [
    ['0', '0'],
    ['1', '1'],
    ['8', '1000'],
    ['9', '1001'],
    ['15', '1111']
]

def validate(job):
    job.prepare_student_files(remove_directories=True)
```

(continues on next page)

(continued from previous page)

```
13      job.run_build(inputs=['dec_to_bin.c'], output='dec_to_bin')
14      for std_input, expected_output in test_cases:
15          running = job.spawn_program('./dec_to_bin')
16          running.sendline(std_input)
17          try:
18              running.expect(expected_output, timeout=1)
19          except TerminationException:
20              job.send_fail_result("Arrgh, a problem: We expected {0} as output for the input {1}.".format(expected_output, std_input), "wrong output")
21              return
22          else:
23              running.expect_end()
24      job.send_pass_result("Everything worked fine!", "Student seems to be capable.")
```

- Line 1: A *TimeoutException* is thrown when a program does not respond in the given time. The exception is needed for checking if the student program calculates fast enough.
- Line 3-9: In this case the test cases consist of the input strings and the corresponding output strings.
- Line 13: The method `run_build()` is a combined call of *configure*, *make* and the compiler. The success of *make* and *configure* is optional. The default value for the compiler is GCC.
- Line 14: The test cases are traversed like in the previous example.
- Line 15: This time a program is spawned with `spawn_program()`. This allows the interaction with the running program.
- Line 16: Standard input resp. keyboard input can be provided through the `sendline()` method of the returned object from line 14.
- Line 18-21: The validator waits for the expected output with `expect()`. If the program terminates without producing this output, a *TerminationException* exception is thrown.
- Line 23: After the program successfully produced the output, it is expected to terminate. The test script waits for this with `expect_end()`
- Line 24: When the loop finishes, a positive result is sent to the student and teacher with `send_pass_result()`.

Warning: When using `expect()`, it is important to explicitely catch a *TerminationException* and make an explicit fail report in your validation script. Otherwise, the student is only informed about an unexpected termination without further explanation.

The following example shows a validator for a C program that reads a string from standard input and prints it reversed. The students have to use for-loops for solving the task. Only the C file has to be submitted.

```
from moodlerunner.exceptions import TimeoutException
from moodlerunner.exceptions import TerminationException

test_cases = [
    ['hallo', 'ollah'],
    ['1', '1'],
    ['1234', '4321']
]

def validate(job):
    job.prepare_student_files(remove_directories=True)
```

(continues on next page)

(continued from previous page)

```
12     file_names = job.grep('.*for[:space:]*(.*;.*;.*).*')
13     if len(file_names) < 1:
14         job.send_fail_result("You probably did not use a for-loop.", "Student is not able to use a for-loop.")
15         return
16 
17     job.run_build(inputs=['reverse.c'], output='reverse')
18     for std_input, expected_output in test_cases:
19         running = job.spawn_program('./reverse')
20         running.sendline(std_input)
21         try:
22             running.expect(expected_output, timeout=1)
23         except TimeoutException:
24             job.send_fail_result("Your output took to long!", "timeout")
25             return
26         except TerminationException:
27             job.send_fail_result("The string was not reversed correctly for the following input: " + std_input, "The student does not seem to be capable.")
28             return
29         else:
30             running.expect_end()
31     job.send_pass_result("Everything worked fine!", "Student seems to be capable.")
```

- Line 1: A *TimeoutException* is thrown when a program does not respond in the given time. The exception is needed for checking if the student program calculates fast enough.
- Line 2: A *TerminationException* is thrown when a program terminates before delivering the expected output.
- Line 4-8: The test cases consist of the input strings and the corresponding reversed output strings.
- Line 12: The `grep()` method searches the student files for the given pattern (e.g. a for-loop) and returns a list of the files containing it.
- Line 13-15: If there are not enough elements in the list, a negative result is sent with `send_fail_result()` and the validation is ended.
- Line 17-25: For every test case a new program is spawned with `spawn_program()`. The test script provides the neccessary input with `sendline()` and waits for the expected output with `expect()`. If the program is calculating for too long, a negative result is sent with `send_fail_result()`.
- Line 26: If the result is different from the expected output a *TerminationException* is raised.
- Line 27-28: The corresponding negative result for a different output is sent with `send_fail_result()` and the validation is cancelled.
- Line 29-30: If the program produced the expected output the validator waits with `expect_end()` until the spawned program ends.
- Line 31: If every test case was solved correctly, a positive result is sent with `send_pass_result()`.

Python Module Index

Index