



Summer Fellowship Report

On

Yaksh: Addition Of A New Code Evaluator

Submitted by

Arpit Kaushik

Under the guidance of

Prof. Prabhu Ramachandran

Department of Aerospace Engineering

IIT Bombay

July 4, 2018

Acknowledgment

I wish to express our profound gratitude to our internship guide prof. Prabhu Ramchandran, Department of Aerospace Engineering, IIT Bombay for his constant support and supervision throughout the internship.

I am highly indebted to my project mentor Mr. Ankit R. Javalkar and my project head Mr. Mahesh Gudi for their continuous support, supervision motivation and guidance throughout the tenure of my project in spite of their hectic schedule who truly remained driving spirit in my project and their experience gave me the light in handling this project and helped me in clarifying the abstract concepts, requiring knowledge and perception, handling critical situations and in understanding the objective of my work.

Contents

1	Introduction	3
2	Moderator and User mode on Yaksh	4
2.1	Moderator Mode	4
2.2	Adding a new question	4
2.3	User Mode	6
3	Code Evaluator	8
3.1	What is Code Evaluator?	8
3.2	STDIO Evaluator :- For Standard Input/Output	8
3.3	Assertion Evaluator :- For Assertion test cases	10
4	Ruby Code Evaluator	11
4.1	Ruby	11
4.2	Ruby STDIO Code Evaluator	11
4.3	Ruby Assertion Evaluator	13
5	Test Cases for Ruby Code Evaluators	16
5.1	Tests for STDIO Evaluator	16
5.2	Tests for Assertion Evaluator	20
5.3	Add Ruby in Backend	22

Chapter 1

Introduction

Yaksh is an Online Test Interface for Conducting online programming quiz. It supports various programming languages like :- C, C++, Python and simple Bash. User can solve any questions by using these languages. Yaksh uses "test cases" to test the the implementations of the students. It also supports simple multiple choice questions and file uploads so that user can easily submit his code. Not only you can practice the questions even you can also conduct a programming quiz that supports various languages. Yaksh also provides various programming courses like:- C++, C and Python. So, we can Yaksh is a best platform to improve our programming skills.

Chapter 2

Moderator and User mode on Yaksh

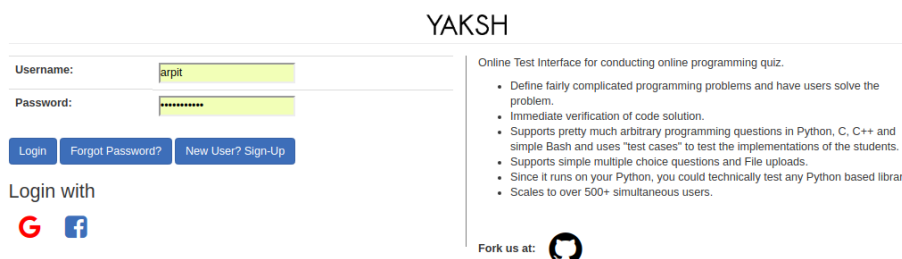
2.1 Moderator Mode

As we know that yaksh is made for conducting quiz. So, In Yaksh, Moderator can easily make questions on various languages like:- Python, C, C++ and Bash. So, for making new questions we have to follow these steps:-

- first login as moderator and go to Questions page.
- Here you can add new questions by click on add question button.
- In add question Page you have to submit all informations like:-Summary, Language, Type of Question, Points, Description, Solution and Many other details related to that question.
- after filling these information you can save this by click on save option.

2.2 Adding a new question

- login as a Moderator



- Click On Questions

YAKSH

[Questions](#)
[Courses](#)
[Monitor](#)
[Grade User](#)
[Regrade](#)
[Change Password](#)

Moderator's Dashboard

List of quizzes! Click on the given links to have a look at answer papers for a quiz.

Courses	Quizzes			
PYTHON_DATA_STRUCTURE	Quiz	Taken By	No. of users Passed	No. of us
	pyhton	1 user(s)	1	1
	Quiz1	1 user(s)	1	1
	addition	0 user(s)	0	0
Copy Of PYTHON_DATA_STRUCTURE	No Quizzes			

- Click on Add Question +

Add Question +

Download Selected ↓

Test Selected

Delete Selected -

- Fill All these details

Add Question

Summary:

Language:

Type:

Points:

Rendered:

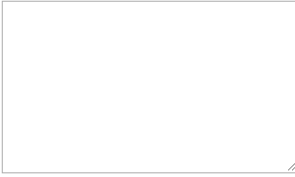
Description:

Tags:

Rendered Solution:

Solution:

- Now save the question

Snippet: 

Minimum Time(in minutes):

Partial Grading:

Grade Assignment Upload:


File: No file chosen

Add Test Case:

2.3 User Mode

In User mode user can submit their code on editor on exam portal and check their whether it is correct or not.

- Submit the question
 - When you submitted correct answer

Write your program below: [Undo Changes](#) 

```
1 def add(a,b)
2   return a+b
3 end
```


Note: You have already attempted this question successfully

Arpit Kaushik(6787) Logged in as arpit

– When you submitted Incorrect answer

Write your program below:

```
1 def add(a,b)
2   return a-b
3 end
```

Check Answer 

Error No. 1

```
Error:
8:in `assert': Assertion failed (AssertionError)
11:in `
'
```


Chapter 3

Code Evaluator

3.1 What is Code Evaluator?

- Basically Yaksh is made for conducting online programming quiz.
- It supports various programming questions Based on Python, C, C++ and Bash and uses test cases to test the implementations of students.
- these code evaluators Evaluate user answer and show the result as successful submission.
- It also shows Errors that user did in his answer when Output is wrong.
- For different languages we have to use different evaluator.
- So, there are two types of evaulators in every language to evaluate the code that submitted by students :-
 - STDIO Evaluator :- For Standard Input/Output.
 - Assertion Evaluator :- For Assertion test cases.

3.2 STDIO Evaluator :- For Standard Input/Output

STDIO :- In computer programming, standard streams are preconnected input and output communication channels between a computer program and its environment when it begins execution. The three input/output (I/O) connections are called standard input (stdin), standard output (stdout) and standard error (stderr). Originally I/O happened via a physically connected system console (input via keyboard, output via monitor), but standard streams abstract this. When a command is executed via an interactive shell, the streams are typically connected to the text terminal on which the shell is running, but can be changed with redirection or a pipeline. More generally, a child process will inherit the standard streams of its parent process.

The three input/output (I/O) connections are :-

- **Standard Input (stdin)**

Standard input is stream data (often text) going into a program. The program requests data transfers by use of the read operation. Not all programs require stream input. For example, the `dir` and `ls` programs (which display file names contained in a directory) may take command-line arguments, but perform their operations without any stream data input.

- **Standard Output (stdout)**

Standard output is the stream where a program writes its output data. The program requests data transfer with the write operation. Not all programs generate output. For example, the file rename command (variously called `mv`, `move`, or `ren`) is silent on success.

- **Standard error (stderr)**

Standard error is another output stream typically used by programs to output error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. This solves the semipredicate problem, allowing output and errors to be distinguished, and is analogous to a function returning a pair of values see Semipredicate problem: Multivalued return. The usual destination is the text terminal which started the program to provide the best chance of being seen even if standard output is redirected (so not readily observed). For example, output of a program in a pipeline is redirected to input of the next program, but errors from each program still go directly to the text terminal.

3.3 Assertion Evaluator :- For Assertion test cases

Assertion :- In computer programming, an assertion is a statement that a predicate (Boolean-valued function, i.e. a true/false expression) is always true at that point in code execution. It can help a programmer read the code, help a compiler compile it, or help the program detect its own defects. For the latter, some programs check assertions by actually evaluating the predicate as they run and if it is not in fact true, an assertion failure, the program considers itself to be broken and typically deliberately crashes or throws an assertion failure exception. An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program. It raised error when assertion failed due to any error. Like if If expression evaluates to TRUE, `assert()` does nothing. If expression evaluates to FALSE, `assert()` displays an error message on `stderr` and aborts program execution.

Chapter 4

Ruby Code Evaluator

4.1 Ruby

Ruby is a dynamic, interpreted, reflective, object-oriented, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan. According to the creator, Ruby was influenced by Perl, Smalltalk, Eiffel, Ada, and Lisp. It supports multiple programming paradigms, including functional, object-oriented, and imperative. It also has a dynamic type system and automatic memory management.

4.2 Ruby STDIO Code Evaluator

For Ruby Stdio Evaluator We have to make A Class called RubyStdIOEvaluator that inherit Another class called StdIOEvaluator. It contains four functions :-

- Contructor

So, this function sets the values for user answer, file path and partial grading by using metadata object that return the values after calling get function from code server. Also, it does same to set the values for expected input, expected output and weight by using testcasedata object. let's see the function:-

```
def __init__(self, metadata, test_case_data):
    self.files = []

    # Set metadata values
    self.user_answer = metadata.get('user_answer')
    self.file_paths = metadata.get('file_paths')
    self.partial_grading = metadata.get('partial_grading')

    # Set test case data values
    self.expected_input = test_case_data.get('expected_input')
    self.expected_output = test_case_data.get('expected_output')
    self.weight = test_case_data.get('weight')
```

- Teardown

This function removes the existing files from our system. everytime when user submits his solution this function removes previous save files from our system.

```
def teardown(self):
    os.remove(self.submit_code_path)
    if self.files:
        delete_files(self.files)
```

- Compile Code

This is the Key function for Ruby stdio evaluator. In this function we make a empty ruby file and save it to a `submit_code_path` by using `create_submit_code_path` function. After that we copy the user answer into `submit_code_path`. for taking input we use StringIO function. At last we run `popen` function using `subprocess` object that allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

```
def compile_code(self):
    self.submit_code_path = self.create_submit_code_file('submit.rb')
    self.write_to_submit_code_file(self.submit_code_path, self.user_an

    if self.expected_input:
        self.expected_input = self.expected_input.replace('\r', '')
        input_buffer = StringIO()
        input_buffer.write(self.expected_input)
        input_buffer.seek(0)
        sys.stdin = input_buffer

    self.proc = subprocess.Popen('ruby {0}'.format(self.submit_code_pa
                                   shell=True,
                                   stdin=subprocess.PIPE,
                                   stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE,
                                   preexec_fn=os.setpgrp
                                   )
```

- Check Code

this function evaluate the user answer using `evaluate_stdio` function that return the value of success and error to code server. it also set the value for `mark_fraction` if there any partial grading in question.

```
def check_code(self):
    success = False
    err = ''
    mark_fraction = 0.0
    success, err = self.evaluate_stdio(self.user_answer,
                                      self.proc, self.expected_input,
                                      self.expected_output)
    mark_fraction = 1.0 if self.partial_grading and success else 0.0
    return success, err, mark_fraction
```

4.3 Ruby Assertion Evaluator

For Ruby Assertion Evaluator We have to make A Class called `rubyCodeEvaluator` that inherit Another class called `BaseEvaluator`. It also contains same functions as `stdio` :-

- Constructor

So, this function sets the values for user answer, file path and partial grading by using metadata object that return the values after calling get function from code server. Also, it does same to set the values for expected input, expected output and weight by using `testcasedata` object.

```
def __init__(self, metadata, test_case_data):
    self.files = []
    self.process = None
    self.submit_code_path = ""

    # Set metadata values
    self.user_answer = metadata.get('user_answer')
    self.file_paths = metadata.get('file_paths')
    self.partial_grading = metadata.get('partial_grading')

    # Set test case data values
    self.test_case = test_case_data.get('test_case')
    self.weight = test_case_data.get('weight')
```

- Teardown

This function removes the existing files from our system. everytime when user submits his solution this function removes previous save files from our system.

```
def teardown(self):
    # Delete the created file.
    if os.path.exists(self.submit_code_path):
        os.remove(self.submit_code_path)
    if self.files:
        delete_files(self.files)
```

- Compile Code

This is the Key function for Ruby Assertion evaluator. In this function we make a empty ruby file and save it to a `submit_code_path` by using `create_submit_code_path` function and then copy `test_case_data` at same place . after that we use `_run_command` function using subprocess object that allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

```
def compile_code(self):
    self.submit_code_path = self.create_submit_code_file('submit.rb')
    submit_f = open(self.submit_code_path, 'w')
    submit_f.write(self.user_answer.lstrip())
    submit_f.write('\n')
    submit_f.write(self.test_case.lstrip())
    submit_f.close()
    self.process = self._run_command(
        'ruby {0}'.format(self.submit_code_path),
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE
    )
```

- Check Code

Function validates student code using instructor code as reference. The first argument `ref_code_path`, is the path to instructor code, it is assumed to have executable permission. The second argument `submit_code_path`, is the path to the student code, it is assumed to have executable permission.

```
def check_code(self):
    success = False
    mark_fraction = 0.0
    proc, stdnt_out, stdnt_stderr = self.process
    stdnt_stderr = self._remove_null_substitute_char(stdnt_stderr)
    if stdnt_stderr == '':
        if proc.returncode == 0:
            success, err = True, None
            mark_fraction = 1.0 if self.partial_grading else 0.0
        else:
            err = "{0} \n {1}".format(stdout, stderr)
    else:
        err = "Error:"
        try:
            error_lines = stdnt_stderr.splitlines()
            for e in error_lines:
                if ':' in e:
                    err = "{0} \n {1}".format(err, e.split(":", 1)[1])
                else:
                    err = "{0} \n {1}".format(err, e)
        except:
            err = "{0} \n {1}".format(err, stdnt_stderr)
    return success, err, mark_fraction
```


Chapter 5

Test Cases for Ruby Code Evaluators

For Checking the correct functionality of code evaluators we have to write some functions to test the user answer whether it is correct or incorrect. It also checks for infinite loop, syntax error, array input etc. So, We made two different classes for STDIO and Assertion Evaluator.

5.1 Tests for STDIO Evaluator

for stdio evaluator we make a class called RubyStdIOEvaluationTestCases that test user answer for different conditions like:- correct answer, incorrect answer, infinite loop and many more. let's see some basic functions :

- **Setup**

In this function we set the values for expected output, expected input and weight etc and save timeout message for infinite loop condition.

```
def setUp(self):
    self.test_case_data = [{'expected_output': '11',
                            'expected_input': '5\n6',
                            'weight': 0.0,
                            'test_case_type': 'stdiobasedtestcase'
                           }]
    self.in_dir = tempfile.mkdtemp()
    self.timeout_msg = ("Code took more than {0} seconds to run. "
                        "You probably have an infinite loop in"
                        " your code.").format(SERVER_TIMEOUT)
    self.file_paths = None
```

- **Test correct answer**

In this function we write correct solution and make python dictionary called kwargs that contains value of user answer, file paths, test case data etc. that kwargs passed as a argument in evaluate function that evaluate that answer and return result.

```
def test_correct_answer(self):
    # Given
    user_answer = dedent("""
    val1 = gets
    val2 = gets
    print (val1.to_i + val2.to_i)
    """)
    kwargs = {
        'metadata': {
            'user_answer': user_answer,
            'file_paths': self.file_paths,
            'partial_grading': False,
            'language': 'ruby'
        },
        'test_case_data': self.test_case_data
    }

    # When
    grader = Grader(self.in_dir)
    result = grader.evaluate(kwargs)

    # Then
    self.assertTrue(result.get('success'))
```

- **Test incorrect answer**

In this function we write incorrect solution and make python dictionary called kwargs that contains value of user answer, file paths, test case data etc. that kwargs passed as a argument in evaluate function that evaluate that answer return result and error.

```
def test_incorrect_answer(self):
    # Given
    user_answer = dedent("""
    val1 = gets
    val2 = gets
    print (val1.to_i * val2.to_i)
    """)
    kwargs = {
        'metadata': {
            'user_answer': user_answer,
            'file_paths': self.file_paths,
            'partial_grading': False,
```

```

        'language': 'ruby'
    },
    'test_case_data': self.test_case_data
}

# When
grader = Grader(self.in_dir)
result = grader.evaluate(kwargs)

# Then
lines_of_error = len(result.get('error')[0].get('error_line_number'))
result_error = result.get('error')[0].get('error_msg')
self.assertFalse(result.get('success'))
self.assert_correct_output("Incorrect", result_error)
self.assertTrue(lines_of_error > 0)

```

- **Test infinite loop**

In this function we write infinite loop condition and make python dictionary called kwargs that contains value of user answer, file paths, test case data etc. that kwargs passed as a argument in evaluate function that evaluate that answer return result and error.

```

def test_infinite_loop(self):
    # Given
    user_answer = dedent("""
        m=1
        loop do
            puts "232"
            m+=1
            break if m==0
        end
    """)
    timeout_msg = ("Code took more than {0} seconds to run. "
                  "You probably have an infinite loop in"
                  " your code.").format(SERVER_TIMEOUT)

    kwargs = {'metadata': {
        'user_answer': user_answer,
        'file_paths': self.file_paths,
        'partial_grading': False,
        'language': 'ruby'},
              'test_case_data': self.test_case_data
             }

    # When
    grader = Grader(self.in_dir)
    result = grader.evaluate(kwars)

```

```
# Then
self.assert_correct_output(timeout_msg,
                           result.get("error")[0]['message']
                           )
self.assertFalse(result.get('success'))
parent_proc = Process(os.getpid()).children()
if parent_proc:
    children_procs = Process(parent_proc[0].pid)
    self.assertFalse(any(children_procs.children(recursive=True)))
```

5.2 Tests for Assertion Evaluator

for assertion evaluator we make a class called `RubyAssertionEvaluationTestCases` that test user answer for different conditions like:- correct answer, incorrect answer, infinite loop and many more. let's see some basic functions :

- **Setup**

In this function we set the values for test case data that assert with user answer and save timeout message for infinite loop condition.

```
def setUp(self):
    self.f_path = os.path.join(tempfile.gettempdir(), "test.txt")
    with open(self.f_path, 'wb') as f:
        f.write('2'.encode('ascii'))
    tmp_in_dir_path = tempfile.mkdtemp()
    self.tc_data = dedent("""
class AssertionError < StandardError
end

def assert(message=nil, &block)
    unless(block.call)
        raise AssertionError, (message || "Assertion failed")
    end
end
""")
    self.test_case_data = [{"test_case": '{0}\n{1}'.format(self.tc_data,
        "test_case_type": "standardtestcase",
        "weight": 0.0
    },
        {"test_case": '{0}\n{1}'.format(self.tc_data,
        "test_case_type": "standardtestcase",
        "weight": 0.0
    },
        {"test_case": '{0}\n{1}'.format(self.tc_data,
        "test_case_type": "standardtestcase",
        "weight": 0.0
    },
    ], ]
    self.in_dir = tmp_in_dir_path
    self.timeout_msg = ("Code took more than {0} seconds to run. "
        "You probably have an infinite loop in your "
        " code.").format(SERVER_TIMEOUT)
    self.file_paths = None
```

- **Test correct answer**

In this function we write correct solution and make python dictionary called `kwargs` that contains value of user answer, file paths, test case data etc. that `kwargs` passed as a argument in `evaluate` function that evaluate that answer and return result.

```

def test_correct_answer(self):
    # Given
    user_answer = "\ndef add(a,b)\n\treturn a+b\nend\n"
    kwargs = {
        'metadata':
            {
                'user_answer': user_answer,
                'file_paths': self.file_paths,
                'partial_grading': False,
                'language': 'ruby'
            },
        'test_case_data': self.test_case_data,
    }

    # When
    grader = Grader(self.in_dir)
    result = grader.evaluate(kwargs)
    # Then
    self.assertTrue(result.get('success'))

```

- **Test incorrect answer**

In this function we write incorrect solution and make python dictionary called kwargs that contains value of user answer, file paths, test case data etc. that kwargs passed as a argument in evaluate function that evaluate that answer return result and error.

```

def test_incorrect_answer(self):
    # Given
    user_answer = "\ndef add(a,b)\n\treturn a-b\nend\n"
    kwargs = {
        'metadata': {
            'user_answer': user_answer,
            'file_paths': self.file_paths,
            'partial_grading': False,
            'language': 'ruby'
        },
        'test_case_data': self.test_case_data,
    }

    # When
    grader = Grader(self.in_dir)
    result = grader.evaluate(kwargs)

    lines_of_error = len(result.get('error')[0].splitlines())
    self.assertFalse(result.get('success'))
    self.assert_correct_output("Error", result.get('error'))
    self.assertTrue(lines_of_error > 1)

```

- **Test infinite loop**

In this function we write infinite loop condition and make python dictionary called kwargs that contains value of user answer, file paths, test case data etc. that kwargs passed as a argument in evaluate function that evaluate that answer return result and error. let's see the function:-

```
def test_infinite_loop(self):

    # Given
    user_answer = "\ndef add(a, b)\n\twhile true\n\tend\nend\n"
    kwargs = {'metadata': {
        'user_answer': user_answer,
        'file_paths': self.file_paths,
        'partial_grading': False,
        'language': 'ruby'},
        'test_case_data': self.test_case_data,
    }

    # When
    grader = Grader(self.in_dir)
    result = grader.evaluate(kwargs)

    # Then
    self.assertFalse(result.get("success"))
    self.assert_correct_output(self.timeout_msg,
                               result.get("error")[0]["message"]
                               )
    parent_proc = Process(os.getpid()).children()
    if parent_proc:
        children_procs = Process(parent_proc[0].pid)
        self.assertFalse(any(children_procs.children(recursive=True)))
```

5.3 Add Ruby in Backend

After making Code Evaluators we have to add ruby to our backend that can be understand by these step:-

- Add Ruby into code evaluators dictionary.
- Add ruby into forms.py for adding into drop drag menu. enditemize

Reference

- for python built-in function <https://docs.python.org/2/library/subprocess.html>
- for Ruby go to <https://www.ruby-lang.org/en/>
- <https://www.ruby-forum.com/topic/120280>