Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

Applying the Virtual Programming Lab with Python

Paulo Henrique Pisani
André C. P. L. F. de Carvalho

Nº 419

ICMC TECHNICAL REPORT

São Carlos, SP, Brazil
July/2017

Applying the Virtual Programming Lab with Python

Paulo Henrique Pisani[1]
André C. P. L. F. de Carvalho[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) - Campus de São Carlos
13560-970 São Carlos - SP, Brazil
e-mail: {phpisani, andre}@icmc.usp.br

July, 2017

Abstract

Evaluation programming exams/assignments can require a large amount of time, particularly if there are too many exams to be evaluated. Some proposals of automatic evaluation of this kind of exams have been presented. This technical report particularly describes how we applied the Virtual Programming Lab in Moodle to evaluate programs coded in Python. Several aspects for this application are described and discussed: creating the activity, writing the evaluation code, testing it, applying and grading the exam.

Keywords: Moodle, Virtual programming lab, Python

Contents

1 Introduction

The Virtual Programming Lab (VPL) is a type of activity that can be created in the Moodle Environment. VPL enables the automatic grading of programming exams/assignments. The home page of the VPL is http://vpl.dis.ulpgc.es/.

This report shows how we have used VPL for the evaluation of Python programs in an Introduction to Computer Science class. This report is written in a step by step way, making it easier for other instructors use the VPL system. Next sections are organized as follows: Section 2 describes how to create a VPL activity; Section 3 discusses suggestions to implement the evaluation code; Section 4 presents methods to test the evaluation code and the activity; Section 5 introduces a set of guidelines we have adopted to apply a VPL-based exam; and, Section 6 shows how to perform grading, including both automatic and manual grading.

2 Creating a VPL activity

This section describes how to create a VPL activity. Some fields are not described, so they can be left with the default values.

2.1 Basic setup

- Turn editing on in the course page (it is a button in the top right corner).
- Click on *add activity or resource* and choose *Virtual Programming Lab*, as shown in Figure 1.
- The main form for the activity will open, as shown in Figure 2:
  - **Fill the name, description and submission period**: Include the questions and other information for the students in the full description field.
  - Submission restrictions:
    - **Maximum number of files**: it is the number of files that the student will submit. For example, if there are 4 questions and each require a file, then this number must be 4. As discussed later in this text, different versions of the exam can be created and this can be implemented by the inclusion of a file that contains the exam version. In this case, the number of files is 5 (one for the exam version plus four for the questions).

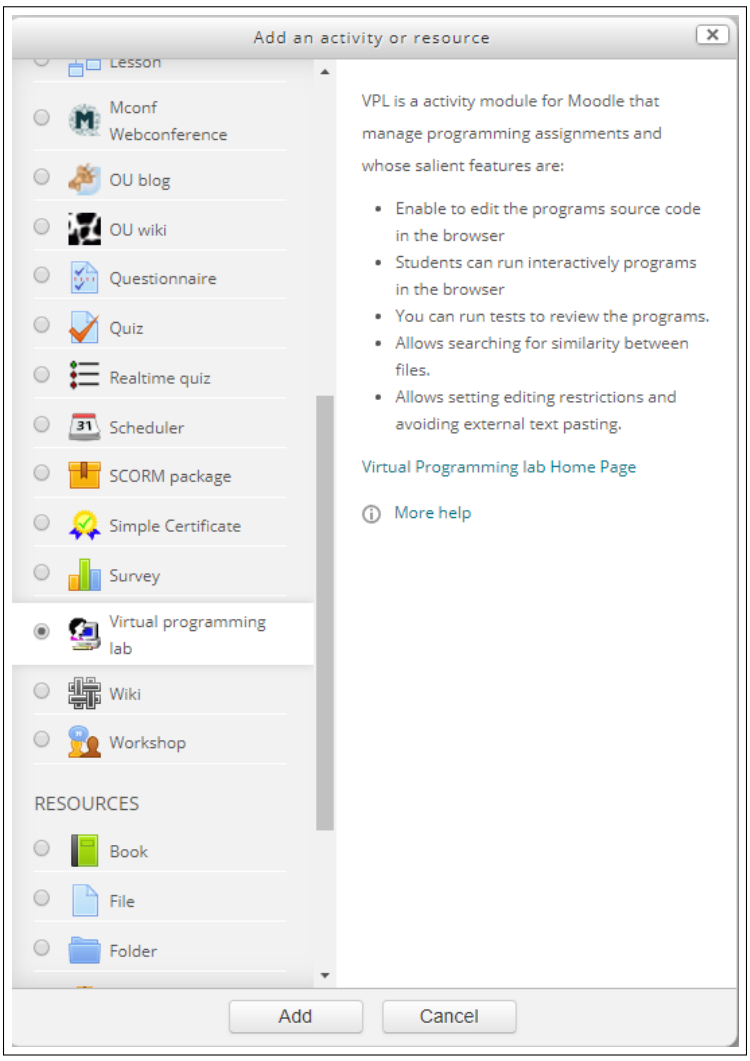

Figure 1: Adding a VPL activity.

* **Type of work**: keep as individual work.
* **Disable external file upload, part and drop external content - VPL editor only**: *Yes* means that the submission can only be performed if the student writes the answer in the editor of the VPL. Copy and paste are disabled in this case. If the option *No* is selected, then the student can work on the answer outside the VPL and then simply copy and paste it in the VPL editor.
* **This activity acts as an example**: keep as *No*.
* **Maximum upload file size**: it is usually not a problem for simple programming questions. Anyway, to avoid excessive large files, it can be set to 32KB, for example.
* **Password**: defines a password to access the exam. The students will need to insert this password. If left blank, no password is required.

– Grade:

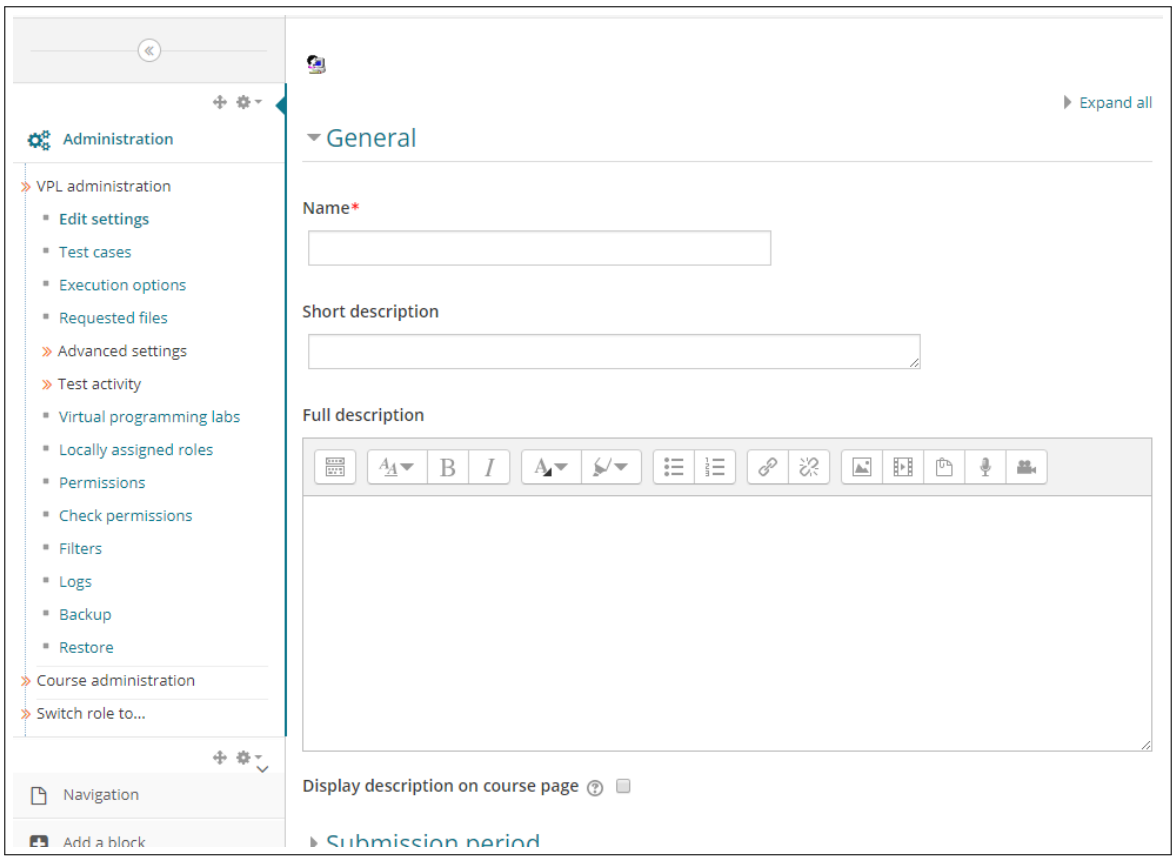

Figure 2: Basic settings.

  - ∗ **Grade**: fill the maximum grade (e.g. 10). It is important to correctly fill in this field because it may be hard to change it after the grading has started (some changes are even not allowed by the system).
  - ∗ **Visible**: it is recommended to keep it as *No*. This is way, the students cannot access the grades. After the grades have been assigned (either manually or with the automatic grader), this can be changed to *Yes*, so all the students can check their grades.
- Common module settings:
  - ∗ **Visible**: Change it to *Hide*. When the exam is applied, then change it to *Show*. This way, the students do not access the questions before the exam.

- Click on *Save and display*

Now the basic data for the activity has been set. The next steps involve setting up execution options and the request files.

2.2 Execution options and requested files

- In the *Administration* menu in the left (Figure 3), go to *Execution options*.

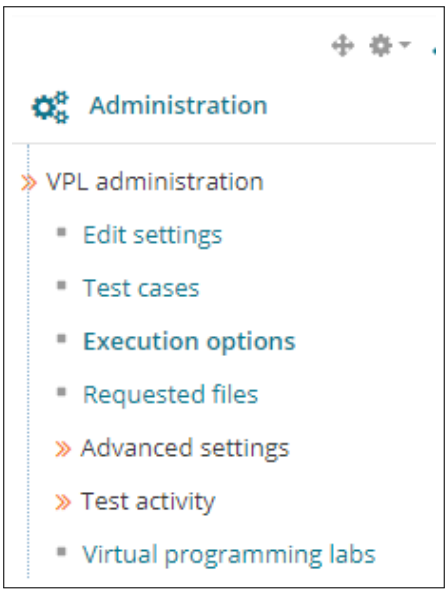

Figure 3: Administration menu (left side).

- **Run, Debug and Evaluation**: If any of these options are left as *Yes*, the student can run, debug or evaluate all questions during the exam. When all are set to *No*, the student can only submit the questions, without running it.
- **Automatic grade**: If set to *Yes*, the automatic grade is used by the Moodle directly. If this is set to *No*, running the automatic grader will not assign a grade. Instead, it will just output any message coded in the grader, which can be used later by the instructor to manually set the grades.
- **Observation (application scenario 1)**: in our previous exams, we have adopted *No* for all options in the page. Therefore, the students can submit their questions, but cannot execute anything via the VPL system. This was done because we faced some network problems during a preliminary test, in which only part of the computers (the ones connected via wireless network) were able to run the VPL, while the others (connected via wired network) were unable to run it. It was probably a firewall issue in the wired network, since the evaluation occurs by a local connection to the VPL server.

  In this case, only after the exam was finished, we set *Evaluate* and *Automatic grade* to *Yes*. Then, the automatic grader was executed (more details on that will be shown later in this report). After that, we returned to this page and set to *No* both options. This is important to avoid that the students click on the *Evaluate* button on their tests afterwards.

  A key disadvantage of the current application scenario is that the

students were allowed to use an external Python development environment during the exam. This way, the students could test their programs before copying and pasting into the VPL for submission. The problem is the difficulty to recognize student attempts to copy from previous exercises in the class. This is because a file opened in the external Python development environment cannot be easily recognized whether it is a file loaded by the student from a previous class or a file that was produced during the exam. To deal with this problem, application scenario 2 shown next can be a solution.

  - **Observation (application scenario 2)**: if the network problem is solved, all computers would be able to run VPL. Then, we recommend to use application scenario 2. though it has never been tested before by our group (but can be checked in the future). In this case, option *Run* can be enabled, so the student can test the code in the VPL. This way, there is no need to use an external Python environment during the exam, making it much easier to spot student attempts to copy from previous exercises. To make it even more effective, the option *Disable external file upload, part and drop external content - VPL editor only* from the initial setup can be set to *Yes*. As a consequence, the student has to type the code in the VPL editor and cannot copy and paste from an external source.

- In the *Administration* menu in the left, go to *Requested files*. In the first time it is opened, it will ask for the name of the first file. If an exam version file is used (it will be discussed later), then insert the name of this file, e.g., "Prova.txt". If a single exam version is used, then insert the name of the first file to be submitted, e.g., "Q1_Somatorio.py".

  - Additional files can be added using the new file button.
  - After adding all files, click in the *Save* button.
  - **Observation - amount of files**: The maximum number of files in the basic setup should be set correctly before adding the files here. For example, if the maximum number is set to 1 there, then it will not be possible to add more than one file here.
  - **Observation - file contents**: leave all files blank. This is how the files will appear to the students when they access the exam.
  - **Observation - changing files later**: please, pay additional attention to this step. From our tests, if there is a need to change file names or add more files after leaving this section, it will not work well. Instead, the VPL activity will stay "broken". The only solution we found in

case we really had to rename or add a file was to create a new VPL activity and discard the current one. Hence, please carefully plan the file names to fill in the their names in the system.

The next steps involve setting up the code for the automatic grader.

2.3 Automatic grader setup

- In the *Administration* menu in the left, go to *Advanced settings*, then go to *Execution files* (Figure 4). As default, four files are already included, they are: "vpl_run.sh", "vpl_debug.sh", "vpl_evaluate.sh" and "vpl_evaluate.cases". To setup the automatic grader code, just the file "vpl_evaluate.sh" has to be edited, as follows:

vpl_evaluate.sh version to enable the use of a custom grader in Python.

```
#! /bin/bash

echo "#! /bin/bash" > vpl_execution
echo "python3 eval.py" >> vpl_execution

chmod +x vpl_execution
```

- Observations regarding the evaluation files:
  - This code tells the automatic grader to call the file "eval.py" to evaluate the student exams. The file "eval.py" has to meet some requirements on how to output the grade. This will be discussed later here.
  - All required files for the grader should be uploaded here. At least one file is required: "eval.py". If this file requires others, then upload all of them. In our implementation, we created an additional file named "vplfunctions.py" plus one for each question: "evalQ1.py", "evalQ2.py", "evalQ3.py" and "evalQ4.py". We will describe how to write these files later.
  - Click in the *Save* button after adding all files.
  - Note that the code above is using Python 3. All the code here is for Python 3, so it is important to keep it that way. In the first tests, we used Python 2.7 too. For that, "python3" has to be changed to "python2.7". In addition, the "eval.py" becomes a bit more complex due to the increased difficulty to load a subprocess and insert/read data from a subprocess in Python 2.7.

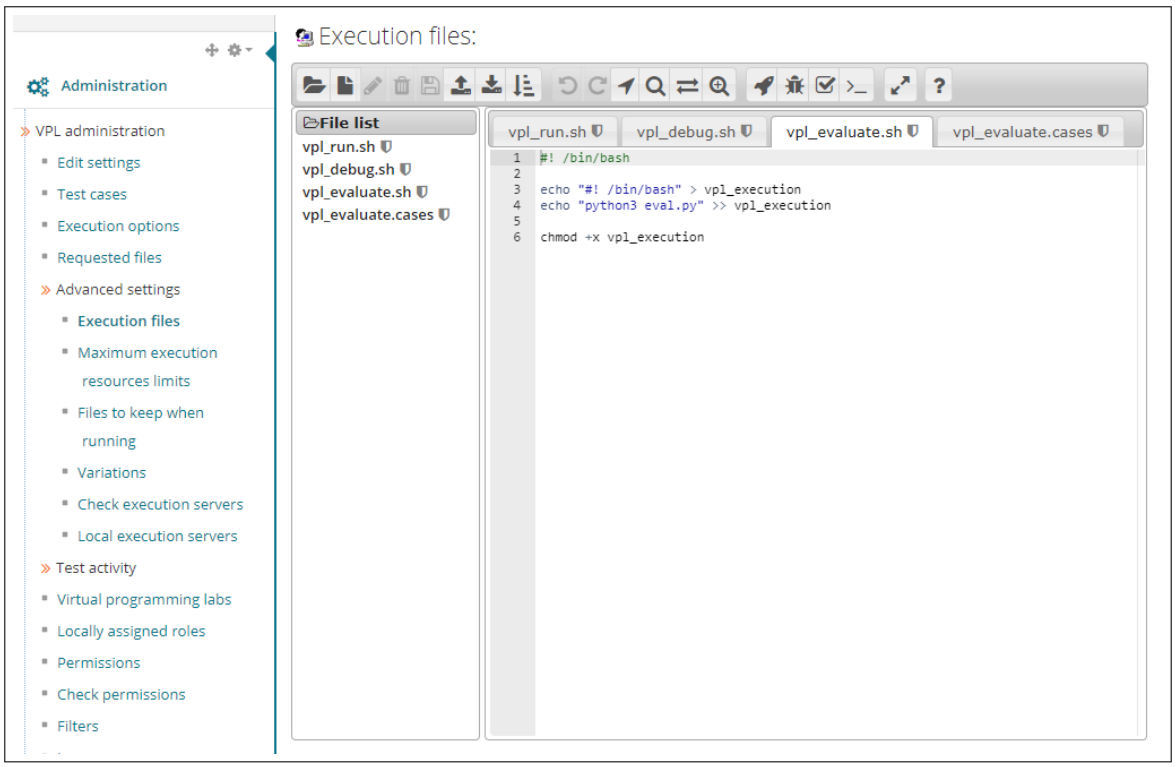

Figure 4: Execution files.

- In the *Administration* menu in the left, go to *Advanced settings*, then go to *Maximum execution resources limits*.
  - **Maximum execution time**: it is important to set a limit, as some students may submit code that run infinite loops. In a previous exam, we adopted the value 16 seconds.
  - **Maximum memory used**: student program can consume too much memory if not properly written, then it is important to set a limit. We adopted 32MB in a previous exam.
  - **Maximum execution file size**: we adopted 256KB in a previous exam (this is the lowest allowed value).
  - Click in *Save options*.
- In the *Administration* menu in the left, go to *Advanced settings*, then go to *Files to keep when running* (Figure 5). Check all uploaded files. The four "vpl_" files do not need to be checked. Click in *save options*.

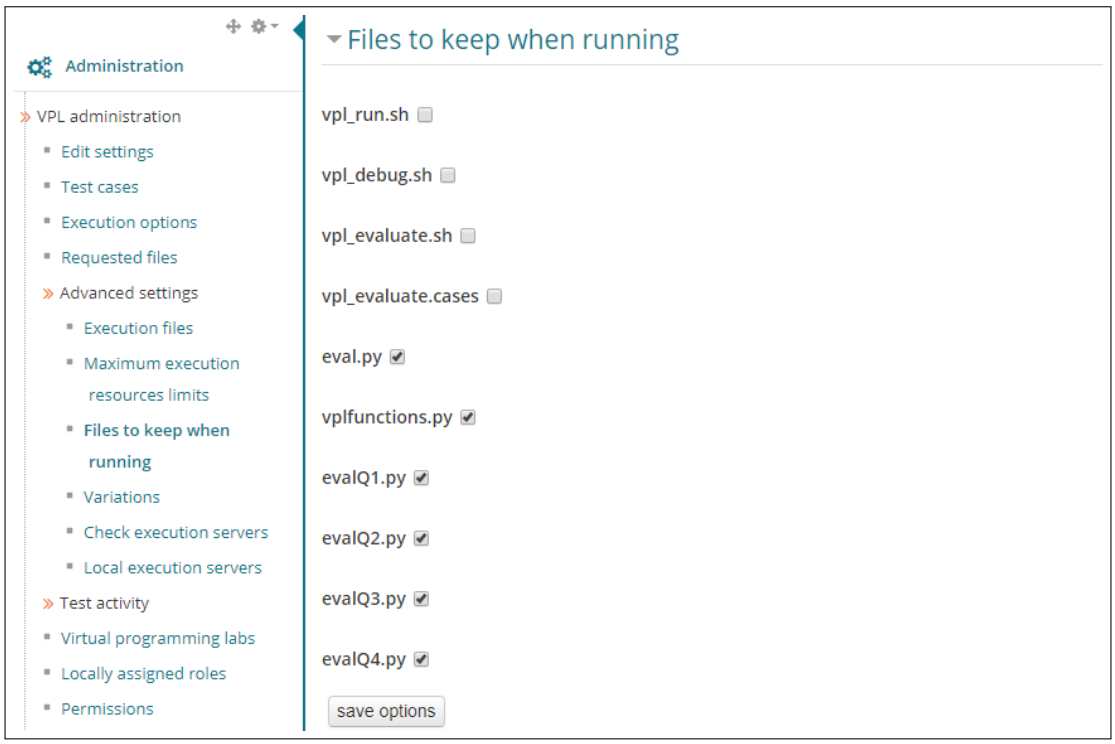

Figure 5: Files to keep during when running.

3 Evaluation code: eval.py

The “eval.py” has the following general structure:

eval.py - general structure.

```
import sys
import vplfunctions
import evalQ1
import evalQ2
import evalQ3
import evalQ4

def evalQuestions():
    grade_value = 0.0
    try:
        vplfunctions.comment("Iniciando avaliacao")

        grade_value += evalQ1.evaluateQ1()
        grade_value += evalQ2.evaluateQ2()
        grade_value += evalQ3.evaluateQ3()
        grade_value += evalQ4.evaluateQ4()

        if (grade_value > 10.0): # should not happen
            print('!ERRO!')
            grade_value = 0.0

        vplfunctions.grade(grade_value)
```

```
    except:
        print('Erro: ' + sys.exc_info()[0])

evalQuestions()
```

- This code if for an exam with 4 questions.
- It follows a simple structure. First, it evaluates each question separately. The code to evaluate each question is on $evalQ1$, $evalQ2$, $evalQ3$ and $evalQ4$, respectively. The points are added to the $grade\_value$, which is then output using the function $vplfunctions.grade$.
- When the evaluation starts, the VPL creates a virtual machine that contains the files submitted by the student in the same path that the "eval.py" is executed. It is performed one student at a time.
- So, it is possible to load the students files as a text file to check whether functions not allowed in the exam were used. It is also possible to run the program or load it as library to access its functions. This is discussed later here.

3.1 Outputting the grade and comments to the student

The VPL recognizes two key expressions for grading and comments. If the expression $Comment :=>> Teste$, the comment "Teste" will be shown to the student. For grading, VPL has the expression $Grade :=>> 8.0$. It would output the grade 8.0. To make it easier, we included this key expressions in the "vplfunctions.py" file.

vplfunctions.py.

```
def comment(s):
    print('Comment :=>> ' + s)

def grade(num):
    print('Grade :=>> ' + str(num))
```

As the VPL receives the grade and comments from a "print" command, it is recommended to run students programs in another thread (subprocess). This way, the student program would not be able to print a comment or a grade in the main thread.

3.2 Implementing different exam versions

In our latest exams, we applied different versions of the exams in order to avoid that one student copies the answer from another student. For example,

there could be two exam versions: A and B. Then, we can adopt a criterion to separate part of the students to work on exam A and part on exam B. We also adopted three versions in some exams: A, B and C.

In order to implement that, we divided the students between A and B in the beginning of the exam. Then, we took notes of which student did which version in a list. During the submission of the exam, apart from the codes for the questions, the students also had to submit the letter (A, B or C) corresponding to the exam version. After the exam was finished, we had to manually check if all students correctly filled the exam version file according to the list we took note in the beginning of the exam.

The "eval.py" file with exam version is shown below. Note that the $examVersion$ is passed as an argument for the evaluation of each question.

eval.py - with different exam versions.

```
import sys
import vplfunctions
import evalQ1
import evalQ2
import evalQ3
import evalQ4

def getExamVersion():
    provaTXT = open("Prova.txt")
    exam_version = provaTXT.read(1).upper()
    if (exam_version == 'P'): # Aluno digitou "Prova X" ao inves de colocar
         somente a letra
        exam_version = provaTXT.read(6).upper()[5]
    provaTXT.close()

    if ((exam_version != 'A') and (exam_version != 'B')):
        raise Exception('Tipo invalido de prova = ' + exam_version + ' (
            deveria ser A ou B)')
    return exam_version

def evalQuestions():
    grade_value = 0.0
    try:
        vplfunctions.comment("Iniciando avaliacao")
        examVersion = getExamVersion()

        vplfunctions.comment("Prova: " + examVersion)

        grade_value += evalQ1.evaluateQ1(examVersion)
        grade_value += evalQ2.evaluateQ2(examVersion)
        grade_value += evalQ3.evaluateQ3(examVersion)
```

```
        grade_value += evalQ4.evaluateQ4(examVersion)

        if (grade_value > 10.0): # should not happen
            print('!ERRO!')
            grade_value = 0.0

        vplfunctions.grade(grade_value)
    except:
        print('Erro: ' + sys.exc_info()[0])

evalQuestions()
```

Another way to implement multiple exam versions would be to create two VPL activities, one for each version. Then, the student would have to choose the correct one. This approach may need an additional care to compute the grades in Moodle. A possible solution is to create a grading group and including both versions of the exam in this group. The final grade then comes from this group. Apart from checking if the student did the correct exam version, in this case it is also needed to check whether the student did just one of the versions.

3.3 Evaluating student questions

In this section we discuss how to implement the $evaluateQX$ functions, where X is the number of the question. The implementation will depend on the the question type. We list three types of questions used in previous exams.

3.3.1 Type 1: Input and Output program

This is the simplest and probably most common type of question. In this type, the question asks the student to write a program that will receive one or multiple inputs and will output something in the screen. An example would be: *write a Python program that receives the numbers A and B, then output the sum of these numbers. The program should work as follows*:

```
Example of the program execution.
```

```
A:10
B:50
60
```

We suggest that the input and output format be as simple as possible. In addition, we strongly recommend that an example of the program execution is shown. It is to ensure that the student submit a program that meets the requirements for the automatic grader.

Before implementing the evaluation code, a function to load the student program in a separate process is needed. This function will input data into the program and read output. This function can be added to the "vplfunctions.py", as follows:

vplfunctions.py - function to load a student program to check its output for a given set of inputs.

```
import subprocess
from signal import signal, SIGPIPE, SIG_DFL

signal(SIGPIPE, SIG_DFL)

...

def getStudentOutput(testInput, arquivoPrograma):
    proc = subprocess.check_output(["python3", arquivoPrograma], input='\n
        '.join(testInput), universal_newlines=True)
    studentOutput = proc.split('\n')
    studentOutput = studentOutput[:-1]
    return studentOutput[0]
```

Now, lets implement the function $evaluateQ1$. For that, we will assume that the answer for this question is stored in the file "Q1_Somatorio.py". The code is shown below. It basically inserts random values for A and B in the program and check whether the answer is as expected. Note that the output format of the student program should meet the expected one, emphasizing the importance of adopting a simple input/output format and including a program execution example.

evalQ1.py - evaluation of question 1.

```
import random
import vplfunctions

def getSomatorio(A, B):
    soma = A + B
    return soma

def getDesiredResultQ1(A, B):
    return getSomatorio(A, B)

def getStudentOutputForQ1(A, B):
    testInput=list()
    testInput.append(str(A))
    testInput.append(str(B))
    studentOutput = vplfunctions.getStudentOutput(testInput, "Q1_Somatorio.
        py")
```

```
    studentOutput = str(studentOutput).split(':')[2]
    return float(studentOutput)

def roundOutput(soma):
    return float("{0:.3f}".format(soma))

def evaluateQ1():
    try:
        ''' A questao vale 4 pontos '''
        grade_value_q1 = 4.0

        # Q2 - Caso 2
        for testIndex in range(1,5):
            vplfunctions.comment('Q1 - Caso ' + str(testIndex) + ' - Inicio
                ')

            testInput1=random.randint(1, 20)
            testInput2=random.randint(1, 20)
            studentOutput = roundOutput(getStudentOutputForQ1(testInput1,
                testInput2))

            desiredResult = roundOutput(getDesiredResultQ1(testInput1,
                testInput2))
            if (studentOutput != desiredResult):
                vplfunctions.comment('Q1 - Caso ' + str(testIndex) + ' -
                    Finalizado(0)')
                return 0.0
            else:
                vplfunctions.comment('Q1 - Caso ' + str(testIndex) + ' -
                    Finalizado(OK)')

        return grade_value_q1
    except:
        vplfunctions.comment('Q1 - Erro')
        return 0.0
```

Lets assume now that there are two exam versions. Version A is the sum we just described. Version B will be: *write a Python program that receives the numbers A and B, then output the double of the sum of these numbers. The program should work as follows*:

```
Example of the program execution.
```

```
A:10
B:50
120
```

This can be implemented as follows. Note that the examVersion is received

by the evaluateQ1 function. The version is loaded by the eval.py as described in Section 3.

evalQ1.py - evaluation of question 1 considering two exam versions.

```
import random
import vplfunctions

def getSomatorioVersaoA(A, B):
    soma = A + B
    return soma

def getSomatorioVersaoB(A, B):
    soma = A + B
    soma = 2 * soma
    return soma


def getDesiredResultQ1(examVersion, A, B):
    if (examVersion=='A'):
        return getSomatorioVersaoA(A, B)
    elif (examVersion=='B'):
        return getSomatorioVersaoB(A, B)
    else:
        raise Exception('Q1: Invalid examVersion = ' + examVersion)


def getStudentOutputForQ1(A, B):
    testInput=list()
    testInput.append(str(A))
    testInput.append(str(B))
    studentOutput = vplfunctions.getStudentOutput(testInput, "Q1_Somatorio.
       py")
    studentOutput = str(studentOutput).split(':')[2]
    return float(studentOutput)

def roundOutput(soma):
    return float("{0:.3f}".format(soma))

def evaluateQ1(examVersion):
    try:
        ''' A questao vale 4 pontos '''
        grade_value_q1 = 4.0

        for testIndex in range(1,5):
            vplfunctions.comment('Q1 - Caso ' + str(testIndex) + ' - Inicio
               ')

            testInput1=random.randint(1, 20)
```

```
            testInput2=random.randint(1, 20)
            studentOutput = roundOutput(getStudentOutputForQ1(testInput1,
               testInput2))

            desiredResult = roundOutput(getDesiredResultQ1(examVersion,
               testInput1, testInput2))
            if (studentOutput != desiredResult):
                vplfunctions.comment('Q1 - Caso ' + str(testIndex) + ' -
                   Finalizado(0)')
                return 0.0
            else:
                vplfunctions.comment('Q1 - Caso ' + str(testIndex) + ' -
                   Finalizado(OK)')

        return grade_value_q1
    except:
        vplfunctions.comment('Q1 - Erro')
        return 0.0
```

Lets assume now that the question does not allow the use of the function SUM available in Python. A possible way to check if the student used it is to load the submitted file and search for the string "SUM", as follows:

```
checking the usage of not allowed functions.
```

```
...

def evaluateQ1(exam_type):
    try:
        programFile = open('Q1_Somatorio.py').read().upper()
        if ('SUM' in programFile):
            vplfunctions.comment('Q1 - Exit')
            return 0.0

...
```

A problem of this approach is that the student cannot use the word SUM, even if it is part of a variable name only. Otherwise, the automatic grader returns zero for the question.

3.3.2 Type 2: Function

In this type of question, the student is asked to implement a function. Hence, only the function code should be submitted. This type of question is easier to test as we only need to import the student file and then call the function. Example of question: *write a function in Python that receives the*

numbers A and B, then returns the sum of the two numbers. The function should be called somanumeros(A,B).

We will assume that the answer for this question is stored in the file "Q2_FuncaoSoma.py". The same code for the type 1 question can be used here. We just need to change the function $getStudentOutputForQ1(A, B)$ to load the student file in a different way, as shown below:

evalQ2.py - evaluation of question 2.

```
...

def getStudentOutputForQ2(A, B):
        try:
                import Q2_FuncaoSoma
                studentOutput = Q2_FuncaoSoma.somanumeros(A, B)
                return studentOutput
        except:
                vplfunctions.comment("Q2 - Error while calling student
                   function")
                raise Exception()

...
```

It is important to check if the student file has a PRINT or an INPUT command. Otherwise, importing the student file could print data in the output of the automatic grader. The same procedure shown in the previous section to check if the function SUM was used can be used for this purpose.

3.3.3 Type 3: Expression

The third type of question involves asking the student to write a Python expression. Example of question: *write an expression in Python that returns the sum of A and B*. In this question, the student is expected to write just the expression, e.g., $A + B$.

Assuming that the expression is stored in the file "Q3_Expressao.txt", the evaluation code could be implemented just by changing the function $getStudentOutputForQ3(A, B)$ from the type 1 example, as follows:

evalQ3.py - evaluation of question 3 (arithmetic expression).

```
...

def getStudentOutputForQ3(A, B):
        try:
                file = open('Q3_Expressao.txt', 'r')
                expressaoA=file.readline().upper()
                studentOutput = eval(expressaoA)
```

```
                return studentOutput
        except:
                vplfunctions.comment("Q3 - Error while using the expression
                    ")
                raise Exception()

...
```

The same principle can be used to evaluate logical expressions too. For example: *write a logical expression in Python that returns whether a given number is a multiple of 15*. In this question, the student is expected to write just the expression, e.g., $z\%15 == 0$.

Assuming that the expression is saved in the file "Q3_Expressao.txt", the evaluation code could be implemented as follows:

evalQ3.py - evaluation of question 3 (logical expression).

```
import random
import vplfunctions

def evaluateQ3():
    # Carrega expressoes
    vplfunctions.comment('Q3 - Carrega expressoes')
    try:
        file = open('Q3_Expressao.txt', 'r')
        expressaoA=file.readline().upper()

        # Expressao A -> Z % 15 == 0
        vplfunctions.comment('Q3 - Expressao A - Teste')
        if expressaoA == '':
            vplfunctions.comment('Q3 - Expressao A: Esta em branco.')
        else:
            expressaoA_OK = True
            Z = 15
            if (not eval(expressaoA)): expressaoA_OK = False
            Z = 75
            if (not eval(expressaoA)): expressaoA_OK = False
            Z = 1320
            if (not eval(expressaoA)): expressaoA_OK = False
            Z = 16
            if (eval(expressaoA)): expressaoA_OK = False
            Z = 10
            if (eval(expressaoA)): expressaoA_OK = False
            Z = random.randint(1, 50000)
            if (eval(expressaoA)!=(Z % 15 == 0)): expressaoA_OK = False
            Z = random.randint(1, 50000)
            if (eval(expressaoA)!=(Z % 15 == 0)): expressaoA_OK = False
```

```
        if expressaoA_OK:
            return 1.0
        else:
            return 0.0
    except:
        return 0.0
        vplfunctions.comment('Q3 - Expressao A: Erro na execucao')
```

4 Testing the VPL activity

This section describes two kinds of tests to be performed:

- Testing the evaluation code locally: the first is to test the evaluation code locally before uploading it to the VPL system. It is possible to run the "eval.py" in the local computer (using a Python programming environment). This allows to debug possible problems in the evaluation code. We recommend that several possibilities be tested here to avoid unexpected errors when the code is used in the VPL system.

  The virtual machine of the VPL runs Linux, so the code shown here was designed for this operating system. To run it in Windows, some small changes have to performed in the "vplfunctions.py" file, as described next:

  - Remove the following lines:

    ```
    from signal import signal, SIGPIPE, SIG_DFL

    signal(SIGPIPE, SIG_DFL)
    ```

  - Change "python3" to "python". In Windows, we must ensure in the Python programming environment that the correct Python verion is being used.

- Testing the activity: the second test involves testing the VPL activity online to certify that everything is working as expected for the exam. All the code for the activity evaluation should be uploaded first. Then, the test can be performed by clicking on the *Test Activity* → *Edit* in the *Administration* panel (Figure 6).

  - This shows how the activity will look like for the students.
  - To test it, just insert the code for the questions, then click on Evaluate to check the uploaded evaluation code. It should present the given grade in the right panel.

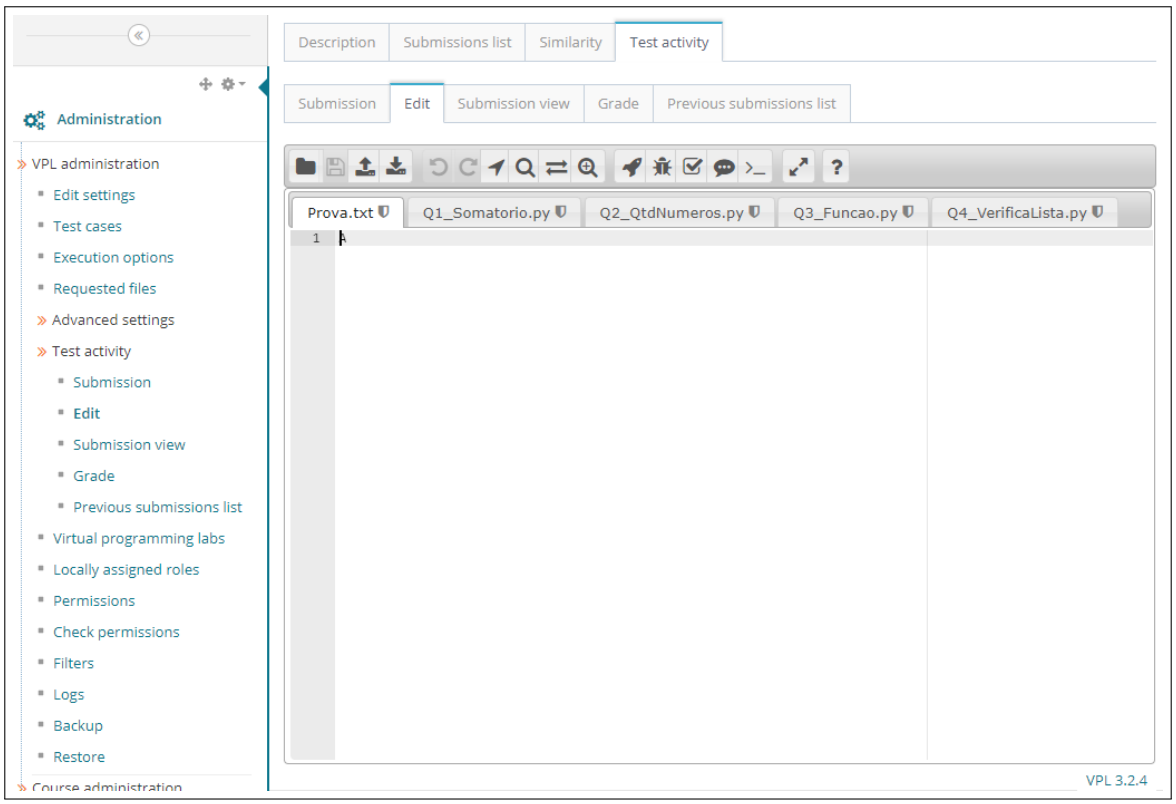

Figure 6: Test VPL activity.

  - We recommend to test some answer possibilities to check if the code is working properly.

5 Applying the exam

When the exam is applied, the first thing to do is to explain the rules for the students, some of them are shown next. After explaining the rules, make the activity visible. This can be done by editing the activity settings in *Common module settings*, then changing *Visible* to *Yes*.

This is a list of guidelines we have followed in previous exams:

- Submit the questions ONLY via the *Edit* tab. This can be done by including the code in the tab correspondent to each question.
- The student must click on *Save* to confirm the submission.
  - The student can save the questions as many times as needed during the submission period. So, the student can save the code for question 1, then come back later and fix an error in the same question and save it again.
  - Sometimes, the machine of the student can stop working for unexpected reasons. So, it is important to recommend that the students

save their work regularly during the exam to avoid data loss in case an unfortunate problem like that occurs.

- The student can check if the submission is ok by clicking on the *Submission view* tab. If all files are shown there correctly, then everything is ok.

- Remind the students to insert the exam version in the corresponding file. During the exam, check if all students already included the letter of the exam in the system.

- In the last exams, we have asked the students to sign a list when they leave the classroom. The instructor would then write the time that the student left the classroom.

  - After the exam is finished, the instructor can check if the last submission time is lower than the time written in the list. If it is not, it is likely that the last submission was done outside the classroom.
  - In addition, the instructor has to check whether there is a student that submitted the exam but has not signed the list. This could indicate someone that submitted from outside the class.

This is a simple measure to avoid submissions from outside the classroom, although not entirely effective. A better way to avoid such a problem would be to restrict the access to the activity to the local network.

6 Grading

After the exam is over, the grading of the programs can be performed. It consists of two main phases: automatic grading and manual check (which can involve manual grading). The steps for the automatic grading are shown next:

- **Enable automatic grading**: in the *Administration* panel, click on *Execution options*, then set *Evaluate* and *Automatic grade* to *Yes*, as shown in Figure 7.

- **Execute automatic grader**: go back to the main page of the activity. Then go to the *Submission list* tab (Figure 8).

  - In *Submissions selection*, set it to *All submissions*.
  - Then, in the *Evaluate* drop box, click on *All*.
  - The automatic grader then starts. It may take a while to finish.

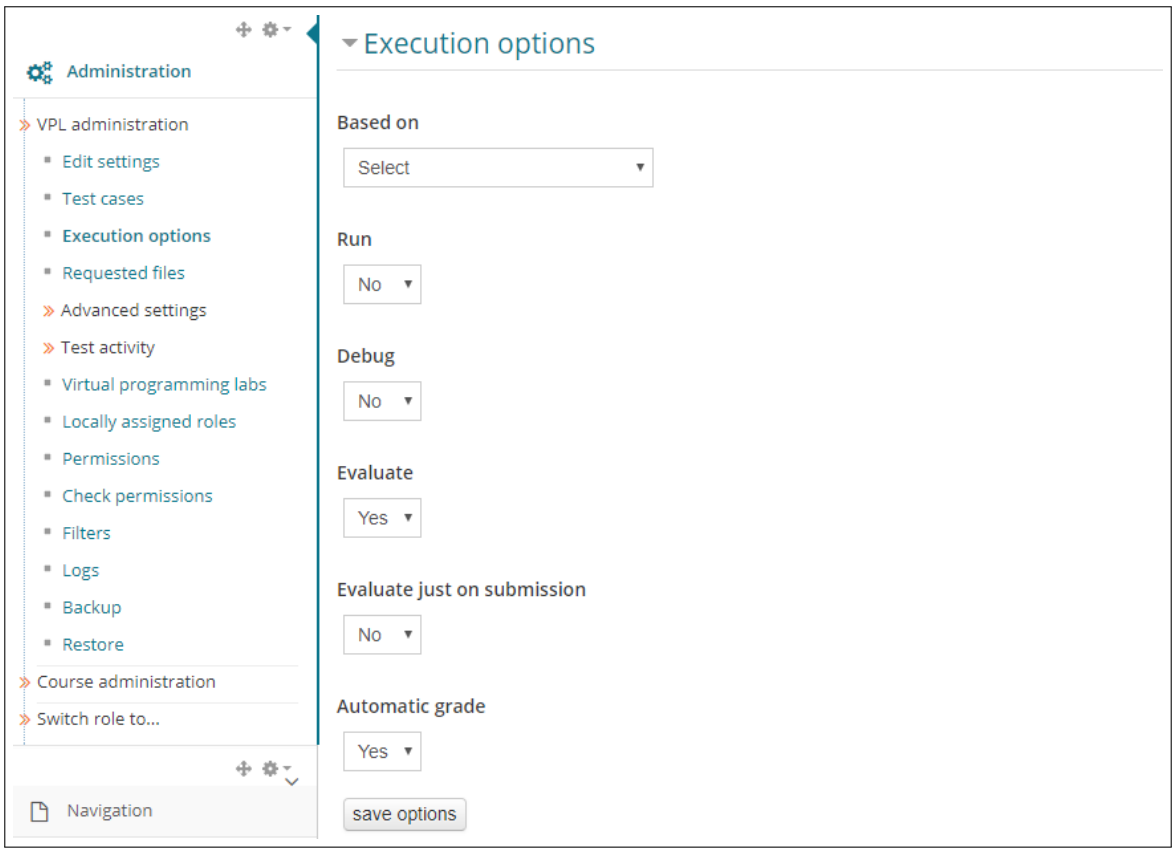

Figure 7: Enable automatic grader.

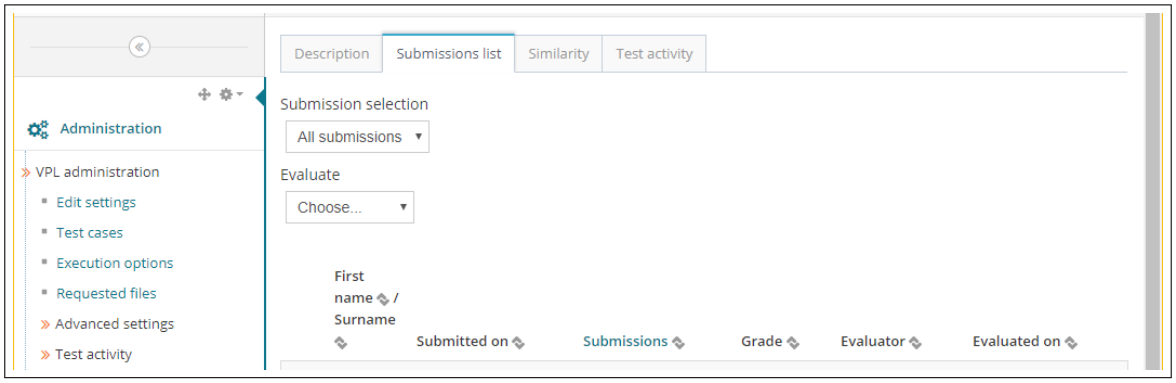

Figure 8: Submission list.

- **Disable automatic grading**: after the automatic grader finishes, in the *Administration* panel, click on *Execution options*, then set *Automatic grade* and *Evaluate* to *No*. This is important to ensure that the students do not trigger the automatic grader later.

After performing these steps, all students will be graded by the automatic grader. It is recommended to check if the evaluation worked well. For instance, if all students receive the grade 0 (zero), it could be an error in the automatic grader.

Apart from providing an unbiased evaluation, the purpose of automatic grading it to also reduce the amount of work to evaluate the exams. However,

this has not happened in the last exams we applied. This is because we had to manually check the exams one by one. This is due to several reasons, such as students that have not followed the expect input/output format or some answers that were partially correct, but the grader returned zero.

To manually change the evaluation, the instructor has to go to the Submission list and click on each student to view his/her exam. After manually calculating the new grade, the instructor then fills in the grade and click on *Grade* (Figure 9). The instructor can also include a comment before clicking on grade.

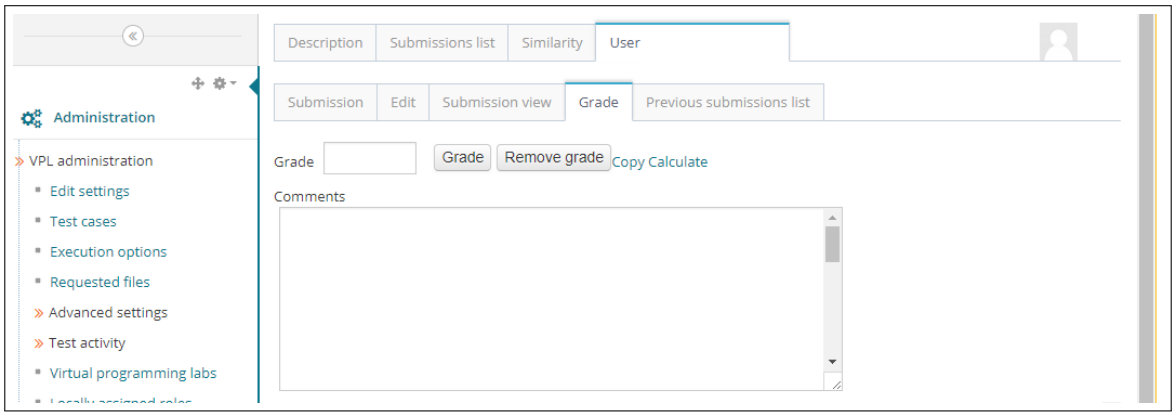

Figure 9: Manual grading.

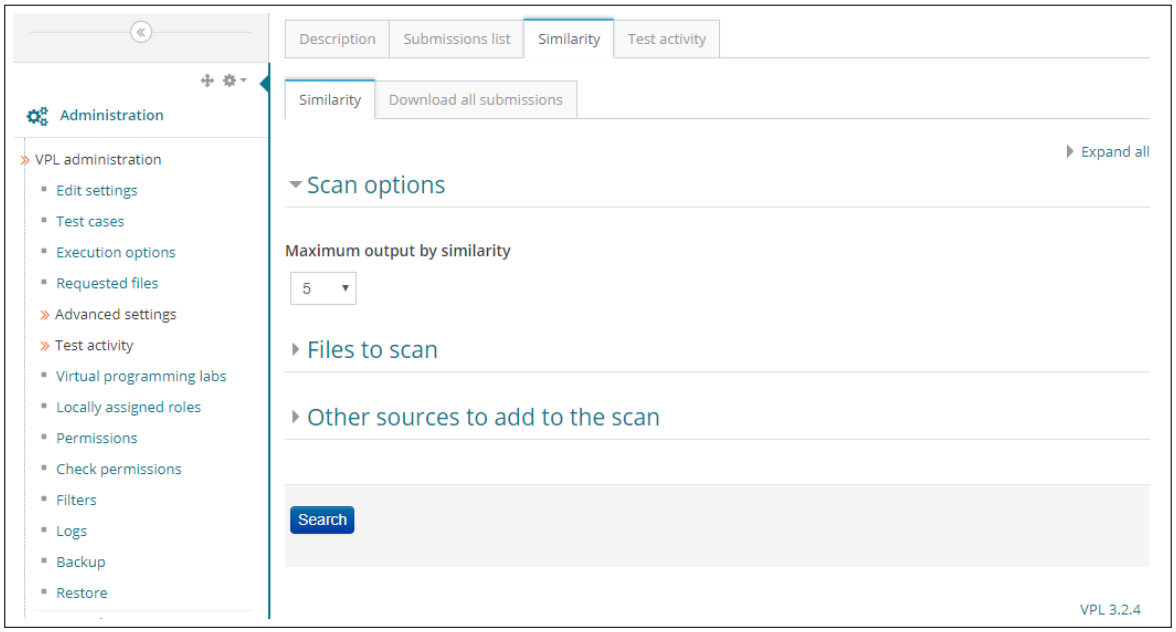

Figure 10: Checking similarities among the student submissions.

It is very uncommon, but a student can submit everything correctly using the *Edit* tab and, then, submit a null activity using the *Submission/Enviar* tab. If the student realizes that did this mistake after the submission period finishes, he/she cannot re-submit the activity. In this case, the instructor can access the previous complete submission and then evaluate this student

manually. There is a tab for the submission history of the students when the exam of the student is opened.

Although we make every effort to avoid that a student copy the answers from another student, it can still happen unnoticed during the exam. There is a feature in the VPL system to check similarities among the answers of the students. In the main exam page, go to the *Similarity* tab (Figure 10). First select the files to scan, then click on *Search*. We suggest to check one file at a time. The system will output several codes that are similar. It is then possible to click on each case to manually check it.

This feature must be used with care. Sometimes, a simple question (like the example we showed here to sum A and B) can be detected as a similarity by the system, although it is probably not. If the question is simple, it is possible that several students submit the same or similar answers, even though they have not copied from each other.