Simple CodeRunner calculation exercise

1. Introduction

Moodle XML file for this exercise (Download and import it in Moodle)

CodeRunner offers even more flexibility when you need to include varying initial values for calculations in your exercises. The Calculated question type still requires an input from the teacher as to what values are used, even if they are randomized between students. CodeRunner allows for each student to input their own initial values, say, from a laboratory exercise, and we can check their homework automatically.

We will examine the functionality of these "expanded Calculated" type of questions with two examples. Later we will dig deeper with an example where the student must perform a linear fit of their data, but first let's look at a simple question, calculation of the amount of substance from a weighed sample. Figure 1 shows what the student sees. At its core, CodeRunner runs the calculations based on the students' input and checks whether the answers are correct.

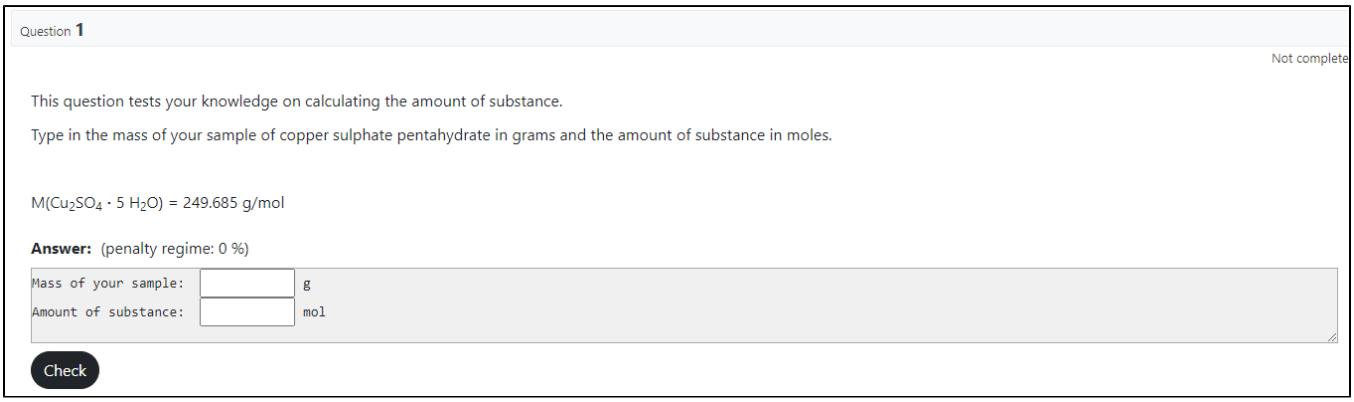

Figure 1. Student's view of the question.

2. Setting up the question

When creating these kinds of questions, we need to give input to five sections (in order of appearance):

1) CodeRunner question type

2) Customisation -> Template

3) General -> Question name and Question text

4) Global extra

5) Test cases -> Expected output

All common settings are already set if you use the given Moodle XML file. Then you only need to change the actual contents of the question.

In CodeRunner question type we set the Question type to python3. Next, we tick the box next to Customise. Then, depending on what you are doing with the question, you can set Marking to fit your needs. This is designed to be something that the students can do over and over again to see when they have the correct answer without any penalties, so we set Penalty regime to 0.

2.1. Setting what the student sees

A logical order for the actual content of the question is to first write the question text in General -> Question text. This is the piece of text that students see in Figure 1. between Question **1** and **Answer**, i.e., the bulk of the question. This is similar to every other question type in Moodle.

Next, we make the answer field, which is defined by Global extra, i.e., it asks for the student's input. This is the part with the gray background in Figure 1. Figure 2 shows what we need to type in order to produce the fields seen in Figure 1.

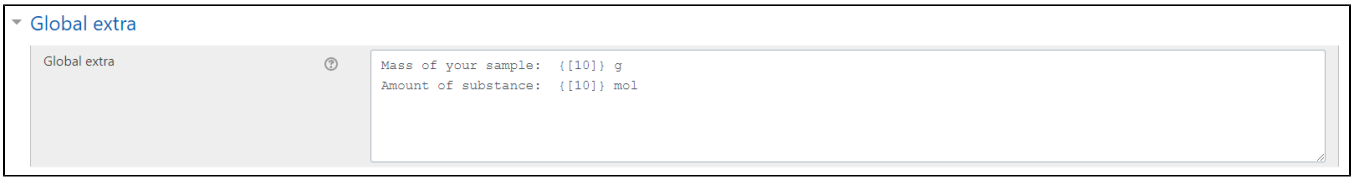

Figure 2. Setting up the Answer field with Global extra.

The syntax is pretty simple, everything you type in the Global extra field will appear in the answer field for the students. The boxes are created inserting a number within square brackets inside curly brackets, like in the example we have typed `{[10]}`. This creates a box with a width of 10 characters.

Typesetting tip: Global extra is monospace, so every character and space will be of same width.

2.2. Checking the answers

Fourth step is setting up what CodeRunner uses as input, what it calculates, and what it checks. This is done in Customisation. Before setting up anything else, the bottom of the Customisation field has the section Input UIs, here you should select Gapfiller from the drop-down menu. Other options are fine as is.

Using the question template of Antti Karttunen, Figure 3 shows the relevant parts we need to consider. There are three main areas:

1) Setting up what we read as input

2) Setting up the calculations

3) Checking the results

Customisation

Template

```
1  """
2  Aalto CHEM CodeRunner Gapfiller question template
3  Developed for CodeRunner version 3.7
4
5  Template version 2020-08-19 antti.karttunen@aalto.fi
6  """
7
8  """
9  STEP 1: Define gapfiller fields. Fields correspond to those defined in CodeRunner's "Global Extra" setting.
10     key -> Key (name) of the field, used in check_answers,
11     desc -> Description of the field, students see this in feedback messages
12     type -> Type of the field (float, int, str)
13     minvalue -> minimum value of the field (optional)
14     maxvalue -> maximum value of the field (optional)
15 """
16 gapfiller_fields = [
17     {"key": "m", "desc": "m(sample)", "type": float, "minvalue": 0.001, "maxvalue": 1000},
18     {"key": "n", "desc": "n(sample)", "type": float, "minvalue": 0.001, "maxvalue": 1000},
19
20 ]
21
22 """
23 STEP 2: Check the answer of the student
24 """
25 def check_answers(answers):
26     # answers is a dict, where keys have been defined above and values are the student's answers
27     import math
28
29     M = 249.685  # g/mol
30     n_sulphate = answers["m"] / M # mol
31
32     """
33     Print out the "judgement". If this printout matches CodeRunner's
34     "Expected output" field, the student gets the points.
35     """
36     # Amount of substance must be correct with 0.1% relative tolerance
37     if math.isclose(answers["n"], n_sulphate, rel_tol = 0.001):
38         print("The amount of substance is correct")
39     else:
40         print("The amount of substance is incorrect. Your nominator and denominator that they are the right way around.")
41
42 """
43 STEP 3: Define language
44     Options: "en", "fi"
45 """
46 lang = "fi"
47
48 ##################### DO NOT MODIFY BELOW THIS LINE ####################
```

Figure 3. How to setup reading the user input, doing the calculations and checking the answers.

2.2.1. Reading the student's input

What we read as input is governed by `gapfiller_fields =[ ]` (Lines 16 – 18 in Figure 3). It is a dictionary of the boxes defined by Global extra. We define (at minimum) three attributes for the student inputs that are the key, description, and type. Key is effectively the name of the field and the key is used as a variable in creating the calculation steps, i.e., how we get the correct answers to which we match the students' answers. Description is as expected, a description of the input. Type refers to what type of input it is, it can be a floating-point number (float), an integer (int) or a string of text (str). In both examples all the inputs will be floating-point numbers.

The attributes for each input are written inside curly brackets with the syntax seen in Figure 3, and different input fields are separated by commas. They run in the order at which they are given in Global extra. In our case, we have two boxes defined in Global extra, so we have two sets of curly brackets. First one corresponds to the mass of the sample and second to the student's input for the calculated amount of substance.

2.2.2. Doing the calculations

Next, we write the mathematics of the question. In Figure 3, this is done in two lines, 29 and 30. In line 29 we save the value of the molecular mass to the variable `M`, and in line 30 we calculate the amount of substance with the molecular mass and the student's input for the mass of the sample. Note the syntax for using values of student inputs. At line 17 we named the key for the mass of the sample as `m`, so when we use it in a calculation like at line 30, we call it with `answers["m"]`. The calculation is saved to the variable `n_sulphate` so we can compare it to the student's answer.

2.2.3. Check the student's answer

As a final step in the Customisation field, we need to check the student's answer. This is done using the function `isclose` from Python's math library. In order to use this, we need to have the line `import math` in the Template, as is seen on line 27 of Figure 3. The example of `isclose` in Figure 3 (lines 37 – 40) is rather self-explanatory, we use `isclose` in a simple if-loop to check if the student's input is close enough to our calculated value. If the two values for the amount of substance (`answers["n"]` and `n_sulphate`) are equal with a 0.1 % tolerance, we print "`The amount of substance is correct`" , and if they differ more than 0.1 %, we print something else, in this case "`The amount of substance is incorrect. Check that your nominator and denominator are the right way around.`"

Now we have a system which checks the student's inputs and prints feedback based on whether it is correct or not. However, we still need to go to the section Test cases, and modify the field labeled Expected output, as the question is not yet answered fully, we only check whether the student's calculation is correct. Since CodeRunner checks code, in the end it compares outputs. The question is answered fully and correctly when the student's answer produces an output that matches the text in the box Expected output, in our case, "`The amount of substance is correct`", as shown in Figure 4. If we asked for more values than one, we would need to copy and paste all the correct answer prints from the checks we do. Here we have only one line of text as we check only one calculated value from the student. We will see an example with more lines in Expected output when we look at a more advanced example using linear regression.

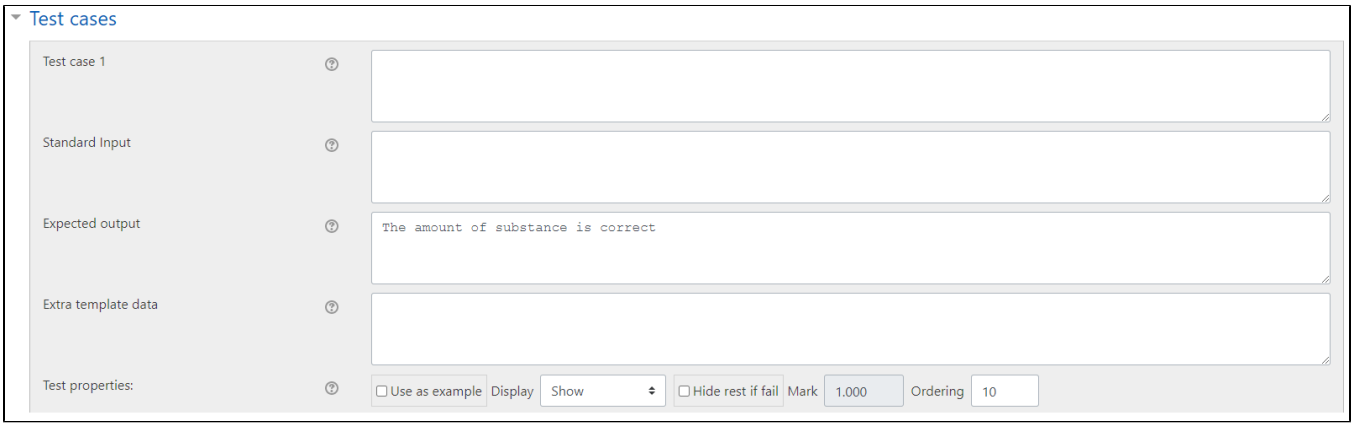

Figure 4. Setting up the expected output.

2.2.4. Language (optional)

You can change the language of possible error messages to Finnish or English at line 46 by having either `fi` or `en` inside the quotes. As line 48 reads, we do not need to touch anything below that line.

3. Important details

3.1. Enough decimals in student answers

In this example we set the error tolerance to 0.1 %, and as a result students might get the question wrong if they round it too heavily. Let's say we have weighed exactly 15 grams of ($CuSO_4 \cdot 5\ H_2O$) so the amount of substance is 0.060075695... moles. If the student has rounded their answer to 0.06 mol, CodeRunner will say it is wrong as the error to the accurate value 0.12599... % > 0.1 %. It is important to let the students know the required accuracy in the question text. With 15 grams of ($CuSO_4 \cdot 5\ H_2O$) an answer of 0.06007 would result in passing the question, so asking for four significant digits is advised.