**Instruction guide for creating QMB adaptive problems**

# Basic Workflow

The basic workflow is simple, consisting of 3 steps

1. Create problem in Excel
2. Run Matlab script to generate XML files from the Excel file.
3. Load and test XML files on studio.EdX.org
4. Create the Knowledge Component Spreadsheet

The following sections will break down these sections in detail. Step 1 is the trickiest and receives the most attention.

# Creating problems in Excel

There are separate Excel files for each unit of the course, e.g. “Arrays problems.xlsx” or “Loops problems.xlsx”. These are in the Github project under “QMB-Problem-Maker\matlab\_attempt\Excel problems”.

Inside these Excel files is a sheet for each Content Group of the unit, so there are sheets named things like “CG1.3.1” or “CG4.3.4”. Each sheet will have multiple problems.

## Simple example

The basic layout for a problem is demonstrated in the following example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sum5 | questionText | <p> What is $AA + BB/$?  </p> |  |  |
|  | variable | AA | randi(10,1) |  |
|  | variable | BB | AA + 1 |  |
|  | variable | CC | AA + BB |  |
|  | answer | CC | KC1 | TRUE |
|  | problemType | Numerical |  |  |
|  | dynamic | TRUE |  |  |
|  | solutionText | <p>  $AA + BB/$ is the sum of numbers $AA/$ and $BB/$. Therefore, the answer is $CC/$ </p> |  |  |
|  | tolerance | 0.02 |  |  |
|  | contentGrouping | CG3.1.2 |  |  |

The top cell in the 1st column is the **problem ID** (also called the **Item ID** by EdX), which in this case is sum5. Notice that all the rows below it are empty. This marks the boundaries between problems, i.e. a new problem begins when a non-empty cell is encountered in the 1st column.

The 2nd column contains the name of a parameter of the problem, e.g. **questionText** is the text displayed to the user that asks the question, and **solutionText** is the explanation that is displayed after the student gets the question correct.

Columns 3 and above contain information relating to the value of that parameter. In most cases, only column 3 is needed, e.g. for the **dynamic** property, the only necessary info is TRUE or FALSE, which specifies whether this problem is dynamic (can have multiple instances) or static (can only have one instance).

The two special properties that require multiple columns are **variable** and **answer**.

## Variables

### Variables are define with Matlab statements that are executed with eval()and saved as strings

For **variables**, the text in the 2nd column is a variable name. In the example above, there are three variables names AA, BB, and CC. The 3rd column is a matlab command that will be evaluated and “stored” to the variable name. I use “stored” in quotes because there this doesn’t create a Matlab variable per se. Instead, the statement in the 3rd column is evaluated with the Matlab function eval, and the output is converted to a string.

Then, whenever the variable name appears, it will be replaced with the string that is the output of the eval(). For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | AA | randi(10,1) |  |

This will have Matlab execute the line randi(10,1)which will generate a random integer. This integer will be converted to a string, e.g. '2'. Now, whenever the sequence of letters AA is encountered (even in following **variable** lines, the letter AA will be replace with the number 2. This is the reason the number is converted to a string, so variable letters can be replaced with strrep(), e.g. strrep('AA','2').

### Variable statements can call other variables

The next variable line is

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | BB | AA + 1 |  |

This will assign the string '3' to BB. What actually gets executed by Matlab is eval('2 + 1'), since the statement in this case is simply AA + 1, and AA was replaced with the value 2 using strrep().

The final line is

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | CC | AA + BB |  |

This will assign the string '5' to CC. AA and BB have values of 2 and 3, respectively, and the statement tells Matlab to add them together.

### Variable lines must be in order of execution

It’s important to note that for successive **variable** lines, you can only reference variables that have already been evaluated. For example, the following two lines would produce an error

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | AA | BB |  |
|  | variable | BB | randi(5,1) |  |

The AA line tries to reference BB, which has not yet been declared. Matlab would execute eval(BB) since it was unable to replace BB with a value using strrep().

### Order does not matter for any other parameter

In the above example, the first row with **questionText** parameter contains the variables AA and BB, despite coming before the **variable** lines. This is allowed. The **variable** lines are always executed first. Then, the script will go back and replace variable names with their values in *every* cell. The letters AA and BB will be replaced with the numbers 2 and 3 in the **questionText** and **solutionText**. The letters CC will be replaced with the number 5 in the **solutionText** and **answer**.

In other words, you could have the **solutionText** parameter even before the **questionText** parameter and the variable values will still be replaced. The order of the other parameters does not matter. For example, you could have the row with **contentGrouping** parameter before the row with the **dynamic** parameter. The only lines that need to be in order are the **variable** lines.

### Use the variable name CODE to run Matlab statements without assigning them to a variable

If you name a variable CODE, then the Matlab statement in the 3rd column will be evaluated without assigning it to a variable. This is useful for multi-line statements like if statments. You’ll still need to fit the multi-line statement into a single line by using commas and semicolons. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | CODE | if AA < 5, myArray = 1:10; end |  |
|  | variable | DD | myArray(7) |  |

Matlab will execute this statement, which creates a Matlab variable named myArray, which can then be called from other lines. If you need multiple if statements, use multilple lines with the variable name CODE.

Also, notive that myArray is declared with a semicolon. Since CODE statements are not assigned to variables and simply evaluated, you need semicolons to suppress output to the Command Line

Referencing myArray will call the actual Matlab variable, unlike the variable AA, which is a string that is inserted in the correct place with strrep(). This trick is helpful for arrays, since the following lines would produce an error

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | AA | [1, 5, -9, 0] |  |
|  | variable | BB | AA(2) |  |

Here, the first variable AA is note save as an array, but the string '[1 5 -9 0]'. Matlab will first run

eval('[1, 5, -9, 0]') which returns the array, which is then converted to a string with mat2str(). Therefore, when AA is referenced in the next line, Matlab will replace the letter AA with the string and try to execute

eval('[1 5 -9 0](2)'). This will produce an error.

To index from an array, you can use a CODE statement to create a Matlab variable, e.g.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | CODE | myArray = [1, 5, -9, 0]; |  |
|  | variable | BB | myArray(2) |  |

Or, you can use the extract() function I wrote for just this purpose.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | AA | [1, 5, -9, 0] |  |
|  | variable | BB | extract(AA,2) |  |

\*\*For a long time, I used num2str() instead of mat2str(), so AA would actually be the string without square brackets, e.g. '1 5 -9 0'. That’s while you’ll see things like extract([AA],2) which would enclose the variable in square brackets to recreate the array.\*\*

### If a variable returns a cell array, Matlab will extract the first element

I didn’t add support for cell arrays. If you try to define a cell array with the line

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | XX | {'hey',1,[0 4]} |  |

Then the variable XX will have the first element of this cell array, i.e. the string 'hey'. I added this functionality mostly for the randsample() function. For example, if you wish to select a random string from a list of strings:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | YY | randsample({'one','two','three'},1) |  |

Normally, the randsample() function would return a 1x1 cell containing a string. Since I added the “take the first element” functionality, the above line will return the string inside the cell.

### Variable names must not be substrings of each other

Since variable values are filled in with the strrep() function, you must make sure that the name of one variable is not a substring of another. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | RR | randi(10,1) |  |
|  | variable | ARRAY | randi(20,5) |  |

Here, the array name RR can be found within the variable name ARRAY. Now, whenever you wish to call the ARRAY variable, the strrep() function will replace the two R characters with the value stored in RR, so you’ll get something like A8AY.

Therefore, I tend to have strange variable names like XX that are all caps and unlikely to be found within another word.

### Excel removes single quotation marks when it’s the first character in a cell, so enclose string declarations in square brackets

If you wish to define a string with the line:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | STR | 'Hello World!' |  |

Excel will remove the first quotation mark ', so you’ll see

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | STR | Hello World!' |  |

This behavior can probably be turned off somewhere in the settings, but I just got into the habit of enclosing strings in square brackets, e.g.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | STR | ['Hello World!'] |  |

This only occurs when the quotation mark is the first character in the cell. Other quotation marks are fine.

### When calling string variables, enclose them in single-quotes

Remember that variables are converted to strings so the parsing script can use strrep() function. This can make it tricky when the variables themselves are string. For example, the following lines will produce an error

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | STR1 | ['Hello World!'] |  |
|  | variable | STR2 | upper(STR1) |  |

Here, we want STR2 to be the all upper-case version of STR1. However, due to the use of strrep(), Matlab will attempt to execute upper(Hello World!). This will produce an error since Matlab will try to find a function or variable named Hello.

To call strings properly, enclose the variable call in single-quotes, e.g.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | STR1 | ['Hello World!'] |  |
|  | variable | STR2 | upper('STR1') |  |

This will create the correct behavior, e.g. Matlab will execute upper('Hello World!').

This behavior might seem silly, but it’s helpful for creating code statements. For example, suppose you want to randomize a comparator symbol, e.g. >, <, >=, or <=. You could write

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | COMP1 | randsample({'<','>','<=','>='},1) |  |
|  | variable | ANS | AA COMP1 BB |  |

The first variable will pick one of the four comparators, and the second will execute the logical statement. For example, using a > comparator and the values of 1 and 2 for AA and BB, respectively, then the ANS variable will be the output of eval('1 > 2'), which would return false.

### Consider using separate functions when the code required for a problem gets too complex to type into Excel

In the Github folder QMB-Problem-Maker\matlab\_attempt\util, there are many functions with names like arry17.m or loop4.m. These are functions that have code written for a specific problem. These are problems where it was easier to simply write everything in Matlab as opposed to worrying about all the issues I have so far described for writing code in Excel.

You can call these functions with CODE variables. For example, here is a template for a Multiple choice problem:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MC\_prob1 | questionText | <p> This is the question text </p> |  |  |
|  | variable | CODE | [correct,incorrect] = ...  prob\_func1(); |  |
|  | variable | ANS1 | correct |  |
|  | variable | ANS2 | incorrect{1} |  |
|  | variable | ANS3 | incorrect{2} |  |
|  | variable | ANS4 | incorrect{3} |  |
|  | variable | ANS5 | incorrect{4} |  |
|  | answer | ANS1 | KC1,KC2 | TRUE |
|  | answer | ANS2 | KC1,KC2 | FALSE |
|  | answer | ANS3 | KC1,KC2 | FALSE |
|  | answer | ANS4 | KC1,KC2 | FALSE |
|  | answer | ANS5 | KC1,KC2 | FALSE |
|  | problemType | MC |  |  |
|  | dynamic | TRUE |  |  |
|  | solutionText | <p>This is the solution text</p> |  |  |

In this example, the CODE variable calls the function prob\_func1(). The returned values are then placed into answer variables, so they contain the text that will be displayed as the answer choices in this Multiple Choice problem.

Also, to repeat something said in 2.2.5, CODE statements require semicolons to suppress output to the command line

### Summary of Variable tips

* Variables are defined with Matlab statements that are executed with eval()and saved as strings
* Variable statements can call other variables
* Variable lines must be in order of execution
* Order does not matter for any other parameter
* Use the variable name CODE to run Matlab statements without assigning them to a variable
* If a variable returns a cell array, Matlab will extract the first element
* Variable names must not be substrings of each other
* Excel removes single quotation marks when it’s the first character in a cell, so enclose strings in square brackets
* When calling string variables, enclose them in single-quotes
* Consider using separate functions when the code required for a problem gets too complex to type into Excel

## Answers

### Answer rows contain the Answer, Knowledge Components and TRUE/FALSE

Answer rows are simple. The answer row in our earlier example was:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | answer | CC | KC1 | TRUE |

The 3rd column contains the actual answer. For Multiple Choice and Checkbox problems, this will be the text that is displayed to the user. For Numerical or Text problems, this will be the number or string, respectively, against which EdX will compare the user’s input.

In this line, the 3rd column has the variable name CC. This will be replaced with the value assigned to CC, which in our case was the number 5.

The 4th column contains a comma-separated list of Knowledge Component (KC) IDs. Our example has one Knowledge Component named KC1.

The 5th column contains either TRUE or FALSE. This signifies whether the answer in the 3rd column is correct or incorrect. This is mostly used for Multiple Choice and Checkbox problems that have multiple answers

### Numerical and Text problems usually have 1 answer, but can have >1

Students answer Numerical and Text problem by typing their answer into EdX, which is then compared with the answer set in the 3rd column. You can have more than one answer row set to TRUE, which will make an <additional\_answer> tag in the final XML file. Then, students could enter more than one correct answer. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | answer | 3.21 | KC2, KC3 | TRUE |
|  | answer | 6.42 | KC2, KC3 | TRUE |

Students could type in either 3.21 or 6.42 and receive credit.

### Multiple Choice and Checkbox answers have multiple answer rows, which contain the choices that will be displayed to the users

The 3rd columns in the answer rows for Multiple Choice and Checkbox problems will be displayed to the users. The 5th column contains whether each answer is true or false. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | answer | ANS1 | KC2, KC3 | TRUE |
|  | answer | ANS2 |  | FALSE |
|  | answer | ANS3 |  | FALSE |
|  | answer | ANS4 |  | FALSE |

The answers here are variables, and only ANS1 here is marked as TRUE, so EdX will only accept this choice as the correct answer. Also, notice that the Knowledge Components are only listed in the TRUE row. It is not necessary to put the KCs in the FALSE rows.

### Answers do not always have to be variables

In the above examples, all the answers in the 3rd columns have been variables, but this isn’t necessary. You can save something like the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | answer | ANS1 | KC2, KC3 | TRUE |
|  | answer | ANS2 |  | FALSE |
|  | answer | ANS3 |  | FALSE |
|  | answer | None of these |  | FALSE |

The last answer is simply the text “None of these”. Any text is acceptable, I simply use variables most of the time since the answers are often dynamically generated.

Also, I use the phrase “None of these” instead of “None of the above” since the answers are always shuffled, so this choice will not always be last.

### All the Knowledge Components for a problem must be placed in one of the answer rows marked as TRUE

In the above example in 2.3.3, the Knowledge Components were listed in a single cell as a comma-delimited list. It might be the case that ANS1 has information about KC2 while ANS4 has information about KC3, but all the Knowledge Components for the problem must be placed together in the 4th column of one of the rows that is marked as TRUE

For Checkbox problems, you might want to have the number of correct answers dynamically generated, e.g. sometimes there are 2 correct answers, while other times there are 3. To accomplish this, you can have variable names in the 5th column that are filled with the strings 'TRUE' or 'FALSE'. For example,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | answer | ANS1 | KC6, KC9 | IS\_CORRECT1 |
|  | answer | ANS2 | KC6, KC9 | IS\_CORRECT2 |
|  | answer | ANS3 | KC6, KC9 | IS\_CORRECT3 |
|  | answer | ANS4 | KC6, KC9 | IS\_CORRECT4 |

Here, there are 4 variables named some variant of IS\_CORRECT that will contain the strings 'TRUE' or 'FALSE', depending on the values in ANS. To make sure that the row marked TRUE has the Knowledge Components, I copied and pasted the same components to each row. For the rows that will be marked as FALSE, the Knowledge Components will be ignored.

### For Numerical problems that have code that produces an error, use “e” as the right answer

For Text problems, if I want the user to know the code in the question produces an error, I will make the correct answer “Error”. The problem description for Text problems will always tell the user: ‘Type “Error” if this code would produce an error (Warning: case sensitive!)’

For Numerical problems, though, the EdX system requires a number as the answer. To mimic the Text question behavior, I make the correct answer the number , i.e. the base of the natural logarithm or approximately 2.718. Users can then type the single letter “e” to get credit for the right answer. Because of this, I avoid writing any problems where happens to be the answer.

### Summary of answer tips

* Answer rows contain the Answer, Knowledge Components and TRUE/FALSE
* Numerical and Text problems usually have 1 answer, but can have >1
* Multiple Choice and Checkbox answers have multiple answer rows, which contain the choices that will be displayed to the users
* Answers do not always have to be variables
* All the Knowledge Components for a problem must be placed in one of the answer rows marked as TRUE
* For Numerical problems that have code that produces an error, use “e” as the right answer

## XML tags

### XML can be placed in questionText, solutionText, and answer cells

The **questionText**, **solutionText**, and **answer** boxes contain the text that will be displayed to the users. Therefore, they can have XML tags to handle formatting. Here is the solutionText from the example above

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | solutionText | <p>  $AA + BB/$ is the sum of numbers $AA/$ and $BB/$. Therefore, the answer is $CC/$ </p> |  |  |

Notice the paragraph tags <p>. You can have any desired XML tags in these boxes, with a few quirks described below

### Matlab code can be inserted with the shorthand symbols $ and /$, or $$, and /$$

In the example above, the variables are surrounded by the dollar symbol $ since they are displayed to the user as Matlab code. The $ symbol is a shorthand for the <code> tag, or more specifically, it’s shorthand for

<code class="lang-matlab"> which makes sure that the code has the proper syntax coloring (reserved words are blue, strings are purple, etc.). The /$ are shorthand for the closing tag, so </code>.

Note that this shorthand is replaced with XML tags using strrep(), so currently there’s no way to display an actual $ symbol to the user.

You can also use two dollar sign symbols $$ to specify <pre><code> tags for preformatted text. For example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Loop1 | questionText | <p> What is the value of $x/$ after the following loop?  </p>  <p>  $$x = 0;  for i = 1:10  x = x + i;  end/$$  </p> |  |  |

The text in the 3rd column will be parsed into XML as:

<p>

What is the value of <code class="lang-matlab">x</code> after the following loop?

</p>

<p>

<pre><code class="lang-matlab">x = 0;

for i = [-9,9,6,-4]

x = x + i;

end</code></pre>

</p>

### Use line break tags <br/> to insert vertical space between paragraphs in solutionText

When using <p> tags in the questionText, EdX will insert some vertical space between successive paragraphs, but it doesn’t in the solutionText. Therefore, I will insert extra line breaks. For example, the two text boxes below will appear the same to the user

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | questionText | <p>  This is a paragraph  </p>  <p>  This is another paragraph  </p> |  |  |
|  | solutionText | <p>  This is a paragraph  </p>  <br/>  <p>  This is another paragraph  </p> |  |  |

The solutionText has the extra <br/> to insert some vertical space.

### Use the mimic\_array\_output() function with $$ tags to mimic Matlab’s Command Line output

When you enter an array into Matlab, it will be formatted in a readable way, e.g.

>> magic(5)

ans =

17 24 1 8 15

23 5 7 14 16

4 6 13 20 22

10 12 19 21 3

11 18 25 2 9

You can mimic this kind of output to the user in EdX using my mimic\_array\_output() function, which takes in an array and outputs a string with the correct line breaks and field widths used by Matlab. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | variable | NAME | ['myArray'] |  |
|  | variable | DISPLAY | ['$$ mimic\_array\_output(magic(5),'NAME') '/$$'] |  |
|  | answer | DISPLAY | KC4 | TRUE |

This creates two variables called NAME and DISPLAY. NAME will be the string 'myArray' and DISPLAY will be a long string with <pre><code> tags and a string with multiple line breaks and tab characters. This variable DISPLAY is then set to one of the answers. In the EdX problem, the user will see

myArray =

17 24 1 8 15

23 5 7 14 16

4 6 13 20 22

10 12 19 21 3

11 18 25 2 9

Notice the mimic\_array\_output() is given the name of the array as a second argument. If no name is passed, the display will show ans (just like Matlab would).

We needed the $$ tags instead of a single $ since the string output of mimic\_array\_output() has line breaks that require a <pre> tag.

### Summary of XML tips

* XML can be placed in questionText, solutionText, and answer cells
* Matlab code can be inserted with the shorthand symbols $ and /$, or $$, and /$$
* Use line break tags <br/> to insert vertical space between paragraphs in solutionText
* Use the mimic\_array\_output() function with $$ tags to mimic Matlab’s Command Line output

## Files and Uploads

### Users can be asked download files using linked in questions

There is a Files and Uploads section for the course on EdX which can store files. These can be linked or displayed in questions with the appropriate XML tags. For example, you can ask the user to download a Matlab data file:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | questionText | <p>  Download <a href="/static/myData.mat">this file</a> and load it into Matlab. You should have an array named $myArray/$. What is $myArray(XX,YY)/$  </p> |  |  |

This uses an <a> tag to insert a link. All files uploaded to the Files and Uploads section of the EdX course will be in the static folder, so the full path is /static/myData.mat.

### Images require an informative alt text for visually-impaired users

Images uploaded to EdX can be displayed with an <img> tag, but they require an informative alt text that can used to answer the question if the user is visually-impaired. For example, I have some questions showing images of different sizes and asking how many rows or columns the image would be if loaded into an array in Matlab. In this case, I would have an alt text saying how many pixels tall or wide the image is.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | questionText | <p>  Suppose you loaded the image below into Matlab.  </p>  <p>  <img src="/static/image1.png" alt="An image that is 400 pixels wide and 300 pixels tall"/>  </p>  <p>  How many rows would the array have?  </p> |  |  |

The alt text must have enough information to answer the question without seeing the image. I have one problem where this isn’t possible (the user must read a word printed on to an image), but to compensate, I wrote an alternative way of solving the problem by listening to a sound file.

### Images must be public domain or created for this course

The images I use in questions fall into two categories:

1. Original images created for the course
2. Public domain images

In the latter case, the images must be free to use and edit with no restrictions. I generally use [pixabay.com](https://pixabay.com/en/) as a source for images. Links to the images are saved in a text file in the Github folder

*QMB-Problem-Maker\matlab\_attempt\Images\image\_links.txt*

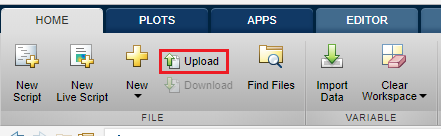
### If the image is too large to see, link the image with a “target” attribute

Below is an html snippet for making the image display a link. Using target="\_blank" in the <a> tag will make the link open the image in a new tab/window. This is helpful if the image is too large to see in the problem itself.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | questionText | <p>    <a href="/static/image1.png" target="\_blank">      <img src="/static/image1.png" alt=" An image that is 400 pixels wide and 300 pixels tall"/>    </a>  </p> |  |  |

### If students are using Matlab Online, they won’t be able to click and drag files into Matlab to load them

In a few course videos, Mike mentions that files can often be loaded into Matlab simply by clicking and dragging the file into the Command Window. You can’t do this with Matlab Online, but you can upload files to the cloud and use them. There’s an “Upload” button on the Home tab in Matlab Online.



### Summary of Files and Upload tips

* Users can be asked download files using linked in questions
* Images require an informative alt text for visually-impaired users
* Images must be public domain or created for this course
* If the image is too large to see, link the image with a “target” attribute
* If students are using Matlab Online, they won’t be able to click and drag files into Matlab to load them

## Other Properties

As demonstrated in the example from 2.1, there are other fields besides **variable** and **answer**. Here is a description of the required and optional fields

### Every problem has questionText and solutionText fields

The **questionText** and **solutionText** fields contain the text that will be displayed to the user. As mentioned in 2.4.1, these fields can contain XML tags for formatting the text. They can also contain variable names which will be replaced after the variable values are generated. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sum5 | questionText | <p> What is $AA + BB/$?  </p> |  |  |

The variable names AA and BB will be replaced with the generated numbers. Users will see something like “What is 3 + 4?”

The **solutionText** should be informative to the specific generated question. For example, I have some questions where the correct answer is sometimes “None of the above”, which requires a different type of explanation than the case when the correct answer is something else. For this reason, I sometime write the solution text in a separate function (see 2.2.10) so I can customize the **solutionText** to match the problem.

### Every problem must contain the problemType and dynamic fields

The **problemType** field clearly denotes the problem type. It can be Numerical, Text, Checkbox, MC (multiple choice) or AnyText. Here is a quick breakdown

* *Numerical* – Asks the users to type in a number and checks to see if the response is equal to a value (or within a defined tolerance). EdX evaluates the user response, e.g. users can type “3+4” and be marked as correct if the answer was “7”.
* *Text* – Asks the users to type in some text and checks to see if the response is the same as an answer string. The default behavior is case-insensitive
* *MC* – Displays several answers and gives credit if the user picks the correct one.
* *Checkbox* – Similar to multiple choice, but can have multiple correct answers. Users must select all the right answers and none of the wrong answers. EdX uses a partial credit system, e.g. selecting 2/3 correct answers is worth more than 1/3 correct answers.
* *Anytext* – Users can type in anything and it will be marked correct.

The **dynamic** field is either TRUE or FALSE and defines whether the problem is static (can only have one instance) or dynamic (can contain multiple instances). If the **dynamic** field is false, then only one XML file will be created for this problem. If true, then there will be an XML file for each instance of a problem.

### There are optional fields, some of which are type-specifc

Here is a breakdown of optional fields

* **max\_attempts** – Integer value, e.g. 3. This is the number of times a user can attempt to answer correctly. The only time I’ve used this field is for True/False MC problems where I want to specify that users should only have 1 attempt.
* **tolerance** – Fraction between 0 and 1, e.g. 0.02. This is for Numerical problems and specifies how accurate users need to be. The default value is 0.05, meaning users can be off by 5% and still be marked correct.
* **isCaseSensitive** – True/False. This is for Text problems. The default value is False, meaning the system is case-insensitive.

### The contentGrouping and difficulty fields are optional, but not recommended

The **contentGrouping** field can contain a string that marks in which group a problem belongs. However, this information is already present on the name of the Excel sheet, so this field is optional. If you include this field, then it will overwrite the use of the sheet name for that specific problem. Here is an example usage:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | contentGrouping | CG3.3.1 |  |  |

The **difficulty** field is a deprecated field that contained a number, e.g. 1 – 3, for an estimate of a problem’s difficulty. However, the adaptive system now estimates each problem’s difficulty internally using data from users, so this field is no longer needed.

### Summary of Properties tips

* Every problem has questionText and solutionText fields
* Every problem must contain the problemType and dynamic fields
* There are optional fields, some of which are type-specifc
* The contentGrouping and difficulty fields are optional, but not recommended

# Run Matlab script to generate XML files

Once a problem is written in an Excel file, instances can be randomly generated and then converted into the XML format used by EdX.

## Testing out single problems

To test one or a few problems at a time, use the script ***generate\_single\_question.m*** located in *QMB-Problem-Maker\matlab\_attempt*. At the top of the script, you must specify the Excel file, the sheet within the file, and the ID of the problem(s).

This script does two things

1. Creates an Excel file in *QMB-Problem-Maker\matlab\_attempt\Processed excel problems* that has all the generated instances of a problem. The number of instances is set by the num\_dynamic variable in the script.
2. Calls the Python code that converts the single Excel file to multiple XML files. I use a batch script to do this named *make\_xml.bat*, but the important Python file is called *convertXLSXtoXML.py*. There should now be XML files in *QMB-Problem-Maker\matlab\_attempt\XML problem files*.

## Creating problems for use

Eventually, you will want to generate 20-30 instances of all the problems for a Unit to be uploaded into the Adaptive system. To do this, use the script ***generate\_all\_questions.m*** located in *QMB-Problem-Maker\matlab\_attempt*. It is similar to *generate­\_single\_question.m* except that you simply tell it the Unit file, e.g. *Array questions.xlsx.*

This script will iterate through each sheet in the excel file (i.e. all the content groupings) and generate instances for every problem in each sheet. The same batch file is used to create the XML files. Note that this can create a few thousand XML files, depending on the number of questions in the Unit and the value set to num\_dynamic.

The python code also creates a “details” file, named *convertXLSXtoXML.py.details*, which is a tab-delimited text file that has details for all the most recently generated problems. EdX needs this details file to upload problems into the adaptive system.

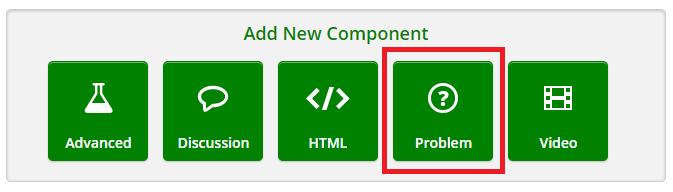
To send files to EdX, compress the XML files and the details file into a zip file and send to Colin.

# Load and test XML files on studio.EdX.org

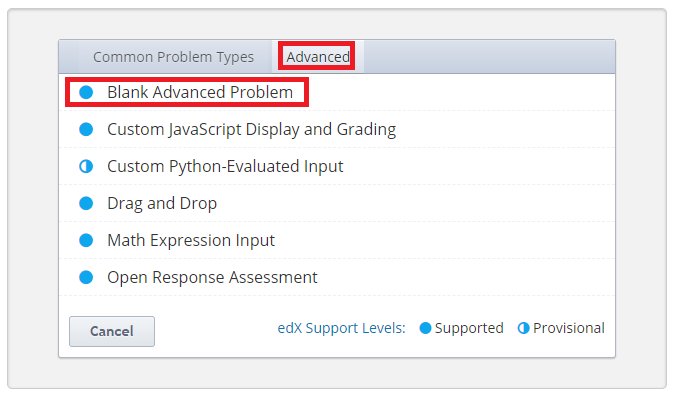
## Loading a sungle XML problem

On the course website <https://studio.edx.org/course/course-v1:HarvardX+QMB1+2T2017>, you can upload and test XML problems to see how they will look to the user.

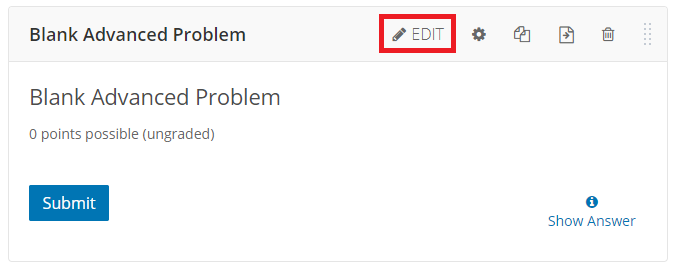
In the “Staff-Only” section, I have some pages under the “Brian’s Test Questions” subsections where questions can be uploaded. To add a new question, click “Problem” under “Add New Component”



Then select “Blank Advanced Problem” under the “Advanced” tab.



A new question should now appear. If you click “Edit”, a text box will pop up where you can copy and paste a single XML problem.



If there are no syntax errors, you can save the question and test it out. The name of the problem can be changed by clicking on “Edit” again and looking under the settings.

## See the syntax highlighting

You can also see the same view as a student by clickin on “Preview” at the very top of the page. It will load a new page with the problems. This view will also show the code syntax coloring if it’s loaded.

To add the syntax highlighting, add a “Raw HTML element” (this is an option for adding an HTML component). Then add the following lines.

<script src="/static/hx\_test.js"></script>

<link rel="stylesheet" type="text/css" href="/static/hx.css">

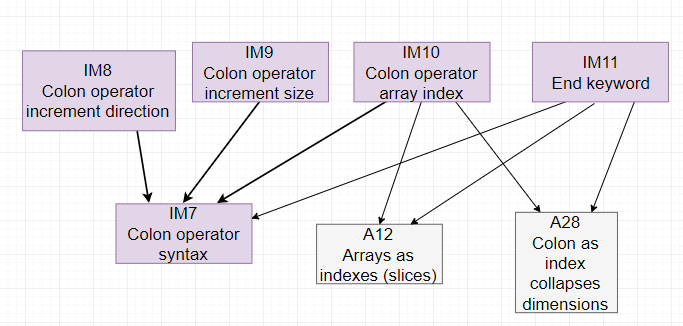
Now, when you click on “Preview”, code should have the proper Matlab colors (blue for reserved words, purple for strings, green for comments).

# Create the Knowledge Component Spreadsheet

## Create a Knowledge Component tree with draw.io

The site [draw.io](https://www.draw.io/) offers a free online program for drawing diagrams. I use this to define the names for Knowledge Components and how they link together.

You can draw the tree by adding boxes and connecting them with arrows. See the draw.io documentation if you’re having trouble doing this. Here is an example section of a tree. Each box is a Knowledge Component.



### Arrows point from post-requisites to pre-requisites

Pre-requisites are KCs that users must have a decent understanding of before the system will show the user questions with post-requisite KCs. So in the example above, users need to understand IM7 Colon operator syntax before the system will show them questions for IM8 Colon operation increment direction.

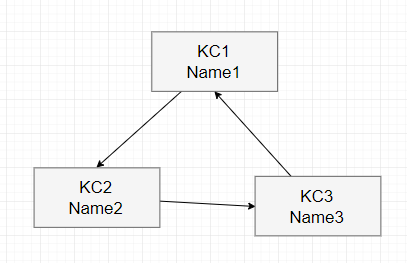
### Arrow LineWidth is used to specify connection strength

KCs links can have either “strong” or “weak” connection strengths. The strength represents how strict the system is in deciding which questions to show to a user. In other words, if a user has a poor understanding of a pre-requisite KC, then the system will be less likely to show a post-requisite question if the connection is strong.

To make a connection strong, set the LineWidth 2pt or higher. A weak connection strength is specified by a LineWidth of 1pt.

### Connections cannot form a loop

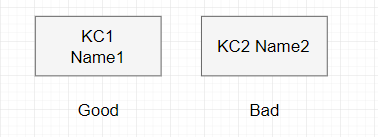
Connections between KCs can be pretty much anything you want, but they cannot form a loop, as shown in the example below. The adaptive system will check to make sure this doesn’t happen, but it’s still something to avoid



Don’t do this

### The KC name should be separated from the KC ID by a line break

The script I wrote to parse the draw.io tree extracts the first line in a KC box as the KC ID, and the second line as the KC name. So when you are typing in a box, add a line break between the ID and name. For example:



### Make a tree for each Unit and export the tree as XML

I made a separate tree for each unit, e.g. Basics or Loops. These files are saved in *QMB-Problem-Maker\matlab\_attempt\XML Trees*.

I saved the trees with the default format, an XML file. The data in it is compressed, but this is processed by the parsing script

### Copy and paste KCs that are linked between Units.

I made a separate tree for each unit, e.g. Basics or Loops. These files are saved in *QMB-Problem-Maker\matlab\_attempt\XML Trees*.

To make a link between two KCs in different units, you can copy paste a KC box from one unit to another. You only need to do this once, meaning you don’t need to show the link in both Unit trees.

In the example above in 5.1, the purple KCs were new to the Unit I was working on (Images), while the gray KCs were from previous units. The color doesn’t matter, I just use it to help me group together KCs visually. The color isn’t used by the parsing script.

### Summary of Draw.io tips

* Arrows point from post-requisites to pre-requisites
* Arrow LineWidth is used to specify connection strength
* Connections cannot form a loop
* The KC name should be separated from the KC ID by a line break
* Make a tree for each Unit and export the tree as XML
* Copy and paste KCs that are linked between Units.

## Run make\_kc\_links\_excel.m to generate the Excel file

The matlab script make\_kc\_links\_excel.m will make the Excel file. The only parameter you should need to set is the list of Units.

For the script to work, you need a draw.io XML file in *QMB-Problem-Maker\matlab\_attempt\XML Trees* and an Excel file in *QMB-Problem-Maker\matlab\_attempt\Excel problems*. These files should begin with the unit names, e.g. *Basics problems.xlsx* and *Basics tree.XML*

It will first parse the draw.io XML file to get the names of the KCs and their IDs. Then, it will parse each sheet in the Excel file to get a list of every problems and the KCs for each problem.

The output will be an Excel file that has the necessary sheets for the EdX adaptive system