HallgrimJS user manual

for version 0.6.5

Toni Ringling

Contents

1 Introduction

HallgrimJS is a direct Successor to Hallgrim, which allows the easy automatic creation of ILIAS-tasks with python scripts - including the capability to create multiple instances, which can, with randomization, be used to create large amounts of variations of a task.

HallgrimJS expands this through the ability to add JavaScript code to your tasks to allow for (figuratively) unlimited possibilities in task design. Additionally it offers many built-in capabilities such as automatic IO-management as well as various converters and interpreters.

Large parts of this document were taken from the original documentation of hallgrimJS by Jan Maximilian Michal, as the non-JavaScript parts remain nearly unchanged.

Please report any errors or issues to the gitlab repository.

2 Installation and configuration

To install HallgrimJS use pip3 (**Note**: HallgrimJS *does not* work with Python 2):

```
pip3 install hallgrimJS
```

A help section is reached by invoking `hallgrimJS` without arguments:

```
usage: hallgrimJS [-h] {init,new,gen,upload,doc} ...

positional arguments:
  {init,new,gen,upload,doc}
    init                Initializes a directory for the use with hallgrimJS
    new                 The utility to generate new scripts.
    gen                 Subcommand to convert from script to xml.
    upload              Subcommand to upload created xml instances.
    doc                 Subcommand to copy the documentation into the current
                        working directory.

optional arguments:
  -h, --help            show this help message and exit
```

Begin by calling `hallgrimJS init` and configuring hallgrimJS, including selecting a directory for the conversion results.
Using the `new` argument will create small examples, which are sufficient for showing how to make non-JavaScript tasks. Further examples may be found on this gitlab page.

3 Convertible files

For a file to be processable by HallgrimJS with the `gen` command it has to be a valid Python 3 script. Correspondingly, the file name should end in `.py`. Furthermore it should always contain the fields `meta`, `task` and `feedback`. The fields `task` and `feedback` follow rules specific to the type of question.
The `meta` field is a dictionary contains the author, name of the task, the number of points (optional) and, most importantly, the question type. Such a field could look like:

```
meta = {
    'author': 'John Doe',
    'title': 'A very difficult task',
    'type': 'gap',
    'points': 3000,
}
```

Various other entries may be used in specific types of task.

For further clarification, consult the examples created with the `new` argument or the examples on this gitlab repository.

HallgrimJS makes use of a variety of tags which are written into tasks in the form `[tag(extraInfo)]content[/tag]`. Although the syntax may suggest this possibility, these can generally not be nested.

4 Non-JavaScript tasks

These types of tasks have no faculties for easy JavaScript-injection and when processed the behaviour will be very similar to the original Hallgrim.

4.1 Choice questions

`task` and |feedback| can have arbitrary content. The possible answers are given in the field `choices`. The answers are shuffled by ILIAS by default. To disable this add `'shuffle': False,` to the `meta` field.

4.1.1 Single choice

In this type of task only one answer can be selected by the student, but varying points can be given for each answer. `choices` could look like this:

```
choices = """
    [ ] a very wrong answer
    [ ] not correct
    [2] this is half correct
    [4] this is the best solution
"""
```

If the number of points is given in the meta field it is possible to just use an `X` instead of the number of points:

```
[ ] not correct
[ ] not correct
[X] the only correct answer
```

4.1.2 Multiple choice

In this type of task multiple answers can be selected by the student and varying points can be given for each answer. `choices` could look like this:

```
choices = """
    [0.5] correct
    [ ] not correct
    [1] correct
    [ ] not correct
"""
```

It is again possible to mark correct answers with an `X` if `meta['points']` is given. In contrast to other question types, `meta['points']` will now be interpreted as the amount of points per correct answer and not in total.

4.2 Gap questions

Gap questions may contain any mix of the following gap types. These follow the general syntax `[gap_type(x.xP)]content[/g` (part for points not always contained), which contains the type of the gap and the amount of points for a correct answer. They should be inserted into the `task` field.

4.2.1 Text gap

This type of gap is just written as `gap` with the content consisting of all the correct answers separated by commata. No line break should occur within the gap. This could look like:

```
[gap(2.0P)]Answer 1, Answer 2, Answer 3[/gap]
```

4.2.2 Numeric gap

This type of gap is written as `numeric` with the content consisting of either just the correct value or the value, a minimum bound and a maximum bound separated by commata. The value should be inside these bounds. This therefore has any of the forms:

```
[numeric(4.0P)]<value>,<min>,<max>[/numeric]
[numeric(4.0P)]<value>[/numeric]
```

In the latter case, only the exact value is accepted as correct. These values may be integers or floating point numbers.

4.2.3 Selection gap

This type of gap is written as `select` with the content consisting of the answers preceeded by their corresponding amount of points with each pair separated by line breaks. The value should be inside these bounds. This therefore has any of the forms:

```
[select]
[1.0] good choice
[ ] bad choice
[0.5] acceptable choice
[ ] bad choice
[/select]
```

4.3 Order questions

For ordering questions the field `order` must be added, which contains the answers in the correct order. This can be written like this:

```
order = """
-- Answer A
-- Answer B
-- Answer C
-- Answer D
"""
```

It can also be written like this:

```
order = "Answer A -- Answer B -- Answer C -- Answer D"
```

4.4 Free questions

This type of question only allows a task description and feedback. The students will be able to insert their answers into a single unformatted HTML textarea.

5 JavaScript tasks

The following contents will generally refer to operation with JavaScript-code and not the surrounding Python-code.
Tags, anything of the form `[type(name)]content[/type]` etc. are used for automatic insertion of different elements and are put into the task and not the JavaScript part. The `name` part, internally the attribute `HGID`, will be used to identify and access the resulting elements. This `HGID` is **not** case-sensitive. Many of the input tags also allow for primitive feedback in the form of the best possible solution to be directly inserted. Expect that any elements you manually insert into the webpage, that is without the use of tags, may break.
Note however, that the tags (anything of the form) still belongs into the task and not the section for JavaScript.
All functions and global variables belonging to HallgrimJS begin with `HG`. If any such function does not appear in this document, it should not be manually called and may change significantly in use or disappear.

JavaScript tasks internally are regular text-gap-tasks with the gaps being hidden. Therefore they can only handle exact answers. This requirement can sometimes be weakened by postprocessing of inputs in the evaluator(5.1.2).

Please note, that JavaScript tasks may be manipulated by students, as they may be able to see and edit the JavaScript on their browsers. If you can, avoid having the correct answer inside your code and instead just reformulate or validate the result in some other way.

JavaScript tasks should contain the field `correctAnswers` which is a list containing the strings, which each represent all correct answers, with the given amount of points, separated by linebreaks for their respective answer gap. The amount of answer gaps is determined by the number of entries.
Such a field could look like this:

```
correctAnswers = ["""
Good answer for first gap: 1.0P
Okay answer for first gap: 0.5P
""",
"""
Good answer for second gap: 2.0P
"""
]
```

For a single answer gap this can also simply be set to the corresponding string like this:

```
correctAnswers = """
Good answer for the gap: 1.0P
Okay answer for the gap: 0.5P
"""
```

Tasks with no answer gaps (so `correctAnswers=[]`) are not supported. Instead use a single gap with `0.0P`.

To accommodate viewing test results, where the task exists twice on one page, you have the variable `HGInstace`, which tracks which of these instances you are operating in. You may check it against `HGRegularInstance` and `HGFeedbackInstance` for these purposes.
Everything which you declare in a Task is, in a certain sense, local - it is within its own scope. You can use all of it freely inside your script but outside elements, such as buttons on the webpage or any other instance of the task, can normally not access everything.
You can use `HGGlobalize(toGlobalize, globalName)` to make `toGlobalize` accessible to such outside elements under the name `globalName`. Globalizing a variable may create a copy and as such the local and global variants may vary over time. Choosing a different name for both and continuing with the global variant is encouraged. Globalizing `HGGlobalize` (especially under the same name) is a bad idea and you should not do it. Similarly `HGInput` and `HGOutput`, which will be described later, should generally not be globalized. For elements which are automatically inserted in response to tags, this globalization is generally also automatic. This globalization was not necessary until version 0.6.1 but hat to be changed to allow for the implementation of fully working feedback.

5.1 General operation

5.1.1 Loading answers back

When viewing the results of a test, continuing an interrupted test or performing another run of a task (with kept answers), the inputs are loaded back into all input fields(5.2). This is done after the initial run of the your JavaScript code. This allows for the definition of custom inputs(5.2.11).

To execute code after loading use `HGRegisterLoadingHandler(handler)` to attach a handler which will be called, without arguments, after loading is complete. Note that this will not be called when the task is being started regularly without previous attempts.

5.1.2 Evaluator

Whenever the task is voluntarily ended or the time runs out all inputs are transferred to the gap for raw inputs(5.3), the metadata to its gap(5.3) and the HGEvaluator is called.
This function is supposed to handle the filling out of the answer gaps and is to be written for each task. It is called without arguments and can be set by calling `HGRegisterEvaluator(evaluator)`.
The base evaluator simply copies the entries of the gap for raw input(5.3) into the first answer gap(5.3). These things also happen when using most of the navigation buttons, which also trigger saving in regular ILIAS. A notable exception is the button to remove questions, which remains untouched to avoid breaking the ability to remove a question from a test. By calling `HGEnableAutoEvaluation()` the evaluator will be executed every time before the answers are saved to the ILIAS server.

5.1.3 Periodic autosaving

All inputs are automatically transferred to the gap for raw inputs(5.3) and the metadata to the meta-gap(5.3) every 30 seconds and sent to the server. Activating auto-saving in ILIAS is redundant and not recommended unless the integrated auto-save fails. Should saving fail, a message will be sent to the console.
Autosaving can be deactivated by calling `HGDisableAutoSave()`, which may be necessary if the student answers become too large in volume.

5.1.4 Meta-Data

The object `HGMetaData` may be used to store any additional data which must be preserved between viewings of the task (such as when leaving and reentering the test or when viewing feedback). For this, you simply set the content like `HGMetaData["myMeta"]="metaStuff"` and access it similary like `metaVar = HGMetaData["myMeta"]`. You should only store strings in this field, as these will be converted into strings between viewings.

5.2 Inputs

For accessing these, call `HGInput(name)` with the name as chosen in your script. `HGGetter` and `HGSetter` are properties of all these which should be used to read and write from them.
The `HGGetter` should be called without arguments and returns the contents as a string. The `HGSetter` should be called with one string which will be written into the output.
Chaining `HGSetter(HGGetter())` should cause no change.

5.2.1 Text field

A text field, a single line gap for inputting text, can be created by inserting `[textField(name)]number[/textField(feedback)]` into the task with `number` being the size of the gap in characters, for example `[textField(mySmallField)]10[/textField(correct)]`. The feedback represents the best text that could be input into the field and may be omitted, including the containing round brackets.

5.2.2 Inline text field

An inline text field, a single line gap for inputting text which can be inserted into text without getting put into a new line, can be created by inserting `[inlineTextField(name)]number[/inlineTextField(feedback)]` into the task with `number` being the size of the gap and maximum input size in characters, for example `[inlineTextField(mySmallField)]10[/inlineTextField(correct)]`. Alternatively it can be inserted like `[inlineTextField(name)]number,number[/inlineTextField(feedback)]` with the first number being the maximum input size and the second number being the visual size. The feedback represents the best text that could be input into the field and may be omitted, including the containing round brackets.

5.2.3 Text area

A text area, a multi-line text input, can be created by inserting `[textArea(name)]number[/textArea(feedback)]` into the task with `number` being maximum number of input characters, for example `[textArea(myLimitedArea)]100[/textArea(First correct line\\nsecond correct line)]`.
If the limit is omitted, such as in `[textArea(myUnimitedArea)][/textArea]`, there is no limit (at least no limit imposed by us) to the input size. The feedback represents the best text that could be input into the area and may be omitted, including the containing round brackets. Regular line breaks or `\n` can be used in this feedback.

5.2.4 Checkboxes

A set of checkboxes, where multiple answers can be selected, can be created by inserting `[checkBoxes(name)]answer1,answer2,...[/checkBoxes(feedback)]` into the task. The answers for the checkboxes are thus separated by commata and can not contain them.
As checkboxes contain a variable amount of selected answers, all selected answers are returned together as a string separated by `HGCheckBoxSeparator` (which is currently set to `"HGCHECKBOXSEPARATOR"`). If no option was selected, this will be am empty string.
The feedback represents the best choices that could be input into the checkboxes and may be omitted, including the containing round brackets. They are written as the comma separated correct answers. If you do not want to deal with the separation issue, making a new checkBox for each option is deemed an acceptable solution, often even a preferable one.
Alternatively, by inserting such boxes like
`[*checkBoxes(name)]answer1:alias1,answer2:alias2...[/checkBoxes(feedback)]` an aliased checkBox-Set can be created. `HGGetter`, `HGSetter` and the feedback will then operate with the aliases while the answers are shown to the user. This is especially useful when the answers are supposed to contain LaTeX. For separating the answers and the aliases, the last colon is considered. Therefore the aliases can't contain colons.

5.2.5 Radio buttons

A set of radio buttons, where at most one answer can be selected, can be created by inserting `[radioButtons(name)]answer1,answer2,...[/radioButtons(feedback)]` into the task. The answers for the radio buttons are thus separated by commata and can not contain them.
The `HGGetter` returns the content of the selected option or an empty string if no option was selected. The feedback represents the best choice that could be input into the radio buttons and may be omitted, including the containing round brackets. It is written as the correct answer.
Alternatively, by inserting such buttons like `[*radioButtons(name)]answer1:alias1,answer2:alias2...[/radioButtons(feedback)]` an aliased radioButtonSet can be created. `HGGetter`, `HGSetter` and the feedback will then operate with the aliases while the answers are shown to the user. This is especially useful when the answers are supposed to contain LaTeX. For separating the answers and the aliases, the last colon is considered. Therefore the aliases can't contain colons.

5.2.6 Selection Inputs(experimental)

A selection input, a drop down menu with answers, can be created by inserting
`[selection(name)]answer1,answer2,...[/selection(feedback)]` into the task. The answers for the selection are thus separated by commata and can not contain them.
The `HGGetter` returns the content of the selected option. Note that by default the first option will be selected. The feedback represents the best choice that could be input into the selection and may be omitted, including the containing round brackets. It is written as the correct answer.
Alternatively, by inserting such a selection like
`[*selection(name)]answer1:alias1,answer2:alias2...[/selection(feedback)]` an aliased selection can be created. `HGGetter`, `HGSetter` and the feedback will then operate with the aliases while the answers are shown to the user. This is especially useful when the answers are supposed to contain LaTeX. For separating the answers and the aliases, the last colon is considered. Therefore the aliases can't contain colons.

5.2.7 Source code editors

An editor can be created by inserting `[editor(name)]language[/editor(feedback)]` into the task with `language` being the language used for highlighting and automatic formatting, such as for example `[editor(myPythonEditor)]python[/editor(first correct line\\nsecond correct line)]`. The feedback represents the best text that could be input into the editor and may be omitted, including the containing round brackets. Regular line breaks or `\n` can be used in this feedback.
The used editor is the Ace editor with the available languages visible in this github page.

5.2.8 Variable Table(experimental)

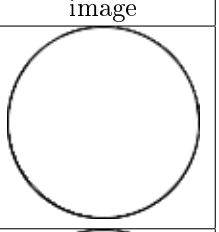
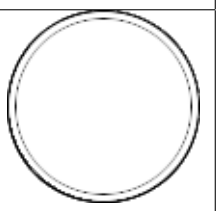
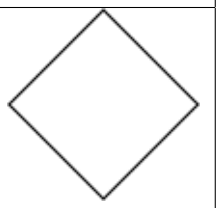
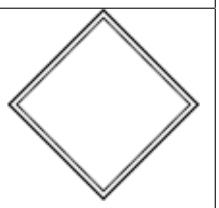
A variable table, a table with text inputs and the option to change size inside the test, can be created by inserting `[variableTable(name)]xSize,ySize,default[/variableTable(feedback)]` into the task, with `xSize` and `ySize` being the default dimensions of the table and `default` being the default entry.
Feedback is again optional and is given in the form
`row 1 column 1, row 2 column 1, row 3 column 1; row 1 column 2, ... , row 3 column 3`, meaning column for column with columns being separated by `;` and cells in a column being separated by `,`. This means, that the entries can not contain commata or semicolons.
The `HGGetter` returns a string with the data given row for row, with the rows being separated by `HGVTableRowSeparator` (which currently has the value `"HGJSVARIABLETABLEROWSEPARATOR"`) and the cells being separated by `HGVTableColumnSeparator` (which currently has the value `"HGJSVARIABLETABLECOLUMNSEPARATOR"`). The `HGSetter` takes the data in the same format.
Preferably the functions `HGGetEntries` and `HGSetEntries` should be used. `HGGetEntries` returns an array of arrays of values, which can be indexed like `values[x][y]` to get the content of the selected cell, which is given as a string. This is a copy of the value array and can be changed without changing the table. `HGSentries(values)` takes just such an array as its argument.
Whenever you set the entries, make sure they form a full rectangle. Furthermore the size should not be 0 in any direction.
It should be noted, that the entries are still internally saved when the table is shrunk, as to allow for recovery of entries after accidental clicking of a shrinking button, however, this additional data is not returned by either `HGGetter` or `HGGetEntries` and will be deleted when setting the data (including when reloading the page).

5.2.9 Drawing canvas

A canvas with integrated drawing tools can be created by inserting
`[drawingCanvas(name)]width,height,stackDepth[/drawingCanvas(feedback)]` into the task.
The current drawing tools only consist of a pencil-type brush (completely opaque circle) and a text stamp with color selection and a size slider with clearing- and undo/redo-capabilities.
The width and height may be omitted, but only both at a time, such as in `[drawingCanvas][/drawingCanvas]` or `[drawingCanvas]20[/drawingCanvas]`, at which point the size will default to 400px×400px. Keep in mind, that a large canvas may cause performance issues. Similarly the stack depth, regarding the undo-stack, may be omitted, in which case it is not limited, such as in `[drawingCanvas]300,500[/drawingCanvas]`. The feedback represents the best picture that could be input into the area and may be omitted, including the containing round brackets. The picture is given as the relative path (from execution) of the picture. *png*, *jpg*, *gif* and *bmp* should be supported. This picture will not be rescaled to fit the canvas dimensions and simply be drawn into the upper left corner.
In case the automatic processing of such a picture is desired, the `HGGetter` returns a dataURL containing the picture as supplied by the regular `toDataURL()` function of canvas elements.

5.2.10 Graph Editor(experimental)

A graph editor can be inserted like `[graphEditor(name)]optionA, optionB, ...[/graphEditor]`. For feedback a feedbackOnly-element or a custom script are recommended. The first option is the directedness of the graph. It can be set to 'directed', 'undirected' or 'any'. The following options set all available types of nodes. These types are:

name	image
simpleCircle	
doubleCircle	
simpleDiamond	
doubleDiamond	

The graph can be refitted (meaning the view is scrolled and zoomed so all nodes are visible) with `HGRefit` Furthermore the graph editor has the methods `HGGetGraph` and `HGSetGraph` which can be used to export and import graphs in an easily usable format (unlike the string used by the `HGSetter` and the `HGGetter`). The data used in these methods is in the form of an object containing two properties - 'nodes' and 'edges', both of which are arrays containg objects. The `HGGetGraph` also contains an 'edgeList' property, which is however not used for the `HGSetGraph`. This property is an object with the two properties "forwards" and "backwards", each of which is a two-dimensional array with the entry 'edgeList["forwards"][a]' being the array of all numbers n, such that 'edges[n]' is an edge going from 'nodes[a]' to some other node and 'edgeList["forwards"][a]' being the array of all numbers n, such that 'edges[n]' is an edge going from some node to 'nodes[a]'. Bidirectional edges will therefore appear in two entries. Note that this property may be lost through changes to the data and must be maintained by the user, as well as the rest of the data not being changed when this is changed. edgeList is purely meant for examining the graph and not editing it.
An object from the 'nodes' property will most importantly have the properties 'id', 'label' and 'type'. The ids will be integers continuously counting upwards from 0 up to the number of nodes minus 1. These must be kept unique and non-negative for import into a graph editor. The labels are the displayed names of the nodes. The types correspond to the types mentioned above and use the same strings. The labels and types are freely controllable by the students, the ids are not.
The properties 'x' and 'y' are also attached and represent the position of the nodes in the editor, however these can be omitted in newly added nodes. An object from the 'edges' property will most importantly have the properties 'from', 'to', 'label', 'bidirectional'. The from and to properties are the ids of the nodes connected by this edge and as such are integers. The label is the displayed name of the edge. The bidirectional property is a boolean which describes, whether the given edge is bidirectional. For undirected graphs this is always true and in directed graphs it is always false.
Calling `setGraph(getGraph())` may cause internal changes, as 'backward'-edges are converted and nodes are renumbered, but should not adversely affect any provided functionalities.

The controls of the graph editor are as follows:

- Double-clicking in empty space will spawn a new node.
- Nodes can be dragged around by holding the left mouse button.
- The whole view can be dragged around by holding the left mouse button outside a node.
- The view can be zoomed by using the scroll wheel.
- Nodes can be selected by left-clicking them.
- Shift-clicking a node with another node selected will draw an arrow from the selected node to the clicked node (which can be the same).
- Double-clicking any node or edge allows the inputting of the label.
- Right-clicking on a node or edge allows changing of the type or deleting.

5.2.11 Custom inputs

For making a custom input these steps should be followed:

- Give the new input an **attribute** `HGInput`. The value of this attribute should be different to any other given input and will be used to get this input via HGInput.
- Give the new input `HGGetter` and `HGSetter` with the same mode of operation as for all inputs. If these steps are taken, the values from this new input will automatically be saved and loaded.

5.3 Outputs

Outputs, in this context, deal just with the data which is sent to ILIAS in the hidden answer gaps and sent back when viewing the results of a test. For displaying something in your task see 5.4.
The five output gaps are defined by their HGIDs:

- `answerX` refers to the answer which is derived from the users inputs. This is the only part which can be graded automatically by ILIAS. The contents should be set with a custom evaluator (5.1.2). The `X` stand for the number of the answer, starting with 0. `answer0` can also simply be addressed as `answer`.
- `raw` refers to the raw inputs made by the user. This should not be accessed directly. For integrated inputs, the contents are saved and loaded automatically. For own inputs see 5.2.11.
- `comments` may be used for adding comments for potential correctors to see. Currently there is no easy way for these to actually see these comments.
- `meta` refers to metadata, such as the random seed for this task and the user. This should not be accessed directly. Instead, add and read from the object `HGMetaData`5.1.4. The saving and loading will then be handled automatically.
- `misc` may be used for anything else.

For accessing these, call `HGOutput(name)` with the name given above (as a string). `HGGetter` and `HGSetter` are properties of all these which should be used to read and write from them.
The `HGGetter` should be called without arguments and returns the contents as a string.
The `HGSetter` should be called with one string which will be written into the output.
Chaining `HGSetter(HGGetter())` should cause no change.
Defining custom outputs is not supported.
Accessing the gaps directly, without the `HGGetter` and `HGSetter`, which utilize Base64 encoding, is not recommended as the correct answers are automatically Base64 encoded on task creation and not doing this may lead to issues with ILIAS.

5.4 Utilities

Utilities refer to all elements which can not be categorized as Inputs or Outputs as in the previous chapters. These do not always contain a `HGGetter` or `HGSetter`. They can be found by their HGUtility attribute using `HGUtility(name)`.

5.4.1 Buttons

A button can be created by inserting `[button(name)]someJavaScriptCode,label[/button]` where the inserted JavaScript code is executed when the button is clicked and the label is displayed on the button. Anything up to the last comma is considered to be the code that is to be executed. The label can therefore not contain commata. Such a button could for example be inserted as `[button(myButton)]myFunction(1,7),Press here[/myButton]`. No manual globalization for any variables used in the contained code is needed (as it is all wrapped into a new function which is globalized automatically).

5.4.2 Text displays

A text display can be created by inserting `[textDisplay(name)]standardContent[/textDisplay]` and can hold formatted text (with linebreaks, tabs etc.).
It is accessed with its' `HGGetter` and `HGSetter` attributes.

5.4.3 Images

An image can be created added to a task by inserting `[image(name)]path[/image]`. With the path being the relative path to the picture on the machine, where conversion is taking place. *png*, *jpg*, *gif* and *bmp* should be supported.
There is also a non-Javascript alternative available in all types of tasks6.1.

5.4.4 VMs(experimental)

HallgrimJS has a rudimentary capability of emulating 32-bit PCs by using v86. This feature is rather experimental and will probably be subject to change, though probably in the farther future.

Such an emulation can be created using `[virtualMachine(name)]bios,vgabios,iso,memory,vmemory[/virtualMachine]` with the first three entries being file-paths (relative to current working directory) and the last two entries being in bytes.
The VM can be accessed from outside by using the attribute `HGToTerminal(command)` to send a command to the terminal (remember to end it with a linebreak for execution) and the attribute `HGRedirectOutput(handler)` to add a handler which will receive output once the terminal goes back to a state of awaiting commands. Alternatively, especially if the terminal isn't expected to go back to the standard resting state, the attribute `HGFromTerminal()` may be used to grab the contents of the current output buffer.
Note, that the output buffer is deleted when a new command is sent and the sent command will be contained in the buffer afterwards. For an example with an ISO containing tcc (a C compiler) see this gitlab page.

5.5 Converters

5.5.1 Markdown and AsciiMath

By calling `HGMDAndAMToHTML(content)` the content, given in Markdown with AsciiMath, will be converted into HTML with LaTeX which can be put on the page as innerHTML.
The LaTeX can then be visualized by calling `HGProcessLatex(element)` with whatever DOM-element the conversion results are contained in.

5.6 Interpreters

5.6.1 Python

By calling `HGRunPython(code, outputHandler, errorHandler, timeout)` the given Python 2 code is executed with the output from prints being sent to the outputHandler as a string after execution and any error (such as syntax or arithmetic errors) being sent to the errorHandler as a string. `timeout` is given in milliseconds and an error will be caused after the time runs out. `timeout` is optional and defaults to 10000 and if a nonpositive timeout is given, the computation will be allowed to run indefinately. During execution the tab may freeze and as such a timeout is advisable.
Infinite loops currently will freeze the browser and cause a need to reload the page. In the future this will be solved with a settable timeout.

5.7 Further support capabilities

5.7.1 Integrated random number generator

The function `HGRand()` may be used to generate a random 32-bit integer. It uses a linear congruence generator and should therefore not be used for cryptographic purposes. The original seed, which can also be reset with `HGSRand(newSeed)`, is saved in the metadata-gap and can therefore be used to dynamically generate a task in the browser.

5.7.2 Base64 and ASCII conversion

For any, including UTF8, strings you may use `HGToB64(input)` and `HGFromB64(input)` to convert them from and to Base64. This is automatically done by the getters and setters of the output gaps(5.3).

5.7.3 Anti XSS-functions

When using anything other than the integrated getters and setters, make sure to avoid XSS-attacks (i.e. the ability to insert custom javascript by abusing the output gaps, which could translate into an attack on the correctors etc.).
`HGCleaner(content)` will generate the DOM-structure from the given HTML and replace all tags outside of a whitelist with a warning. The new HTML is then returned and should be safe to insert anywhere as innerHTML.
`HGCompEnc(content)` encodes the comparators $<$ and $>$ to avoid any insertion of HTML tags. Note that this may break display in pre- or code-tags.

5.7.4 HGMode

The global variable `HGMode` contains the execution state of the task as a string:

- "ANSWERING" - the task is currently being answered and there were no previous answers
- "CONTINUING" - the task is currently being answered and there were previous answers (caused by interruptions and multiple runs)
- "CORRECTING" - the results of the task are currently being viewed

5.8 Feedback

The content of the feedback can be inserted directly into the tags, as described in the sections for the Inputs, or by setting the contents of the inputs with your own code. It should be noted, that this text is subject to the changes described in the custom markdown section6 and might have to be enclosed by verb-tags6.6. In the first of these cases feedback generally concerns the best possible solution, but the second variant allows for regular feedback under consideration of the answers given by the students. Keep in mind, that all the code may be accessible to the student and therefore also the feedback solutions, which are extremely visible given access to the browser console.

Feedback button

A so-called feedback button can be inserted into a task like `[feedbackButton(name)][/feedbackButton]`. When a student presses this button all input fields will be filled with the provided feedback and they will, in some way, be blocked from answering. Inside ILIAS upon viewing the results of a student who used these solutions an alert will pop up to inform the viewer. For grady, the `HGEvaluator` is automatically replaced by one, which replaces the answer by the fact, that the solutions have been viewed. Note that this happens before any load handler you may register when the page is loaded again. You can check whether `HGMetaData["viewedSolution"]` is equal to `"yes"` to find out, whether the button has been pressed at any point. Since this is saved in the metadata, it persists between viewings but is only accessible in a new instance after loading, in the load handler.

Feedback-only elements

In conjunction with the feedback buttons, feedback-only elements can be used. These only display after a feedback button has been pressed. This can, for example, be used to display additional pictures for clarification without misusing something like a drawing canvas.
To insert such elements, simply enclose the part that is to be hidden like `[feedbackOnly(name)]content[/feedbackOnly]`. The content should be able to contain any other tags. Be aware, that the placement of this forces a line break.

5.8.1 Accessing original solutions for feedback

The original solution given by the student may be useful for giving specific hints in the feedback.
To access these the functions `HGInputs,HGUtilities` are provided. These work like `HGInput,HGUtility` but, instead of providing the element corresponding to the current instance, these provide the list of all fitting elements. To retrieve the original solution these would be indexed with `[0]`.
Note that there is no special version of `HGOutput`, as there is currently no plan to use the answer gaps provided as part of the feedback, since base64-encoded solutions just aren't a nice read.

5.9 Some advice for new JavaScript coders

JavaScript should not be to difficult to grasp if you are familiar with Java or C, but there are a few unusualities, especially when developing with hallgrimJS.

- When you declare a new variable, be sure to use the `var` keyword, such as `var myNum = 5`, to declare it as a local instead of a global variable.
- Since the JavaScript you are writing is written into a string in a python module, you may need some double- or even triple-escaping. For example, to store a string with a line break you may have to write `var myString = "First line\\nSecond line"`, which, as far as python is concerned, will actually be `First line\nSecond line` internally and which the the Execution in Javascript will then convert into the wanted line break.
- These scripts all run locally and are extremely easy to attack by any student with just a regular browser.
- Using `console.log(toLog)` and the built-in console of your browser of choice is very handy for debugging. You can even send more complicated objects, and not only strings, to the console for inspection.

6 Custom Markdown

HallgrimJS supports regular Markdown (using mistune). In general, regular HTML tags can be inserted into the task as well.
There are some customizations which have been applied to the markdown in HallgrimJS.

6.1 Images

An image can be created added to a task by inserting `[image]path[/image]`. With the path being the relative path to the picture on the machine, where conversion is taking place. *png*, *jpg*, *gif* and *bmp* should be supported. There is also a JavaScript alternative available in JavaScript tasks5.4.3.

6.2 LaTeX

LaTeX formulas can be written in Hallgrim just as in ILIAS by enclosing them in double square brackets, for example:

```
[[\\sum_{i=1}^n i = \\frac{n(n+1)}{2}]]
```

Unless raw strings are used (`r'a raw string'`) some symbols, including \, must be escaped.

6.3 AsciiMath

AsciiMath can be written with `$` as delimiter on both sides. It will then be converted to LaTeX.

6.4 Syntax highlighting

For syntax highlighting, the pygments name of a language has to be put on the first line of the code block. It can be enabled for each code block individually by appending `_copy` to the language name. For example:

````
```java_copy
class Car {
    private float price;
    private String manufacturer;
    public void getPrice() {
        return price;
    }
}
```
````

6.5 Tables

Tables are enclosed in [*table*] and [/*table*] with cells in a row being separated by commata and semicollata separating rows. A table could therefore look like this:

```
[table]row1column1,row1column2,row1column3;row2column1,...[/table]
```

Tables have often been found compatible with other tags being inserted inside, although no guarantee is given for this.

6.6 Verbatim text

By enclosing text like `[verb]text[/verb]` it will not be affected by any features in this section. However, this does not influence the processing of javaScript tasks.

7 Question parametrization

The python script for a task is fully executed before the relevant fields are extracted and a task is created. Therefore, by using randomization, HallgrimJS may produce variations of a task.
One can add an integer value to the key `'instances'` in `'meta'`, for example

```
meta = {
    'author': 'John Doe',
    'title': 'A very simple task, but 80 of them',
    'type': 'gap',
    'instances': 80,
}
```

Now by using `hallgrimJS` with the argument `gen` and the flag `-p` one can generated as many instances as specified in meta:

```
hallgrim gen -p parametrized_gap_task.py
```

All these tasks will be contained in one output file.
Note that variables are not cleared between executions when making batches of tasks (i.e. with `-p`). As such a construct like this:

```
try:
    runCounter
except NameError:
    runCounter = -1

runCounter = runCounter + 1
```

could be used to retain the run number or even transfer data between runs. This functionality is not guaranteed, as it depends on the used libraries, which may change.