JupyterKernel

Jupyter kernel written in GAP

1.3

23 February 2019

Markus Pfeiffer

Manuel Martins

Markus Pfeiffer
Email: markus.pfeiffer@morphism.de
Homepage: https://markusp.morphism.de/
Address: School of Computer Science
North Haugh
St Andrews
Fife
KY16 9SX
Scotland

Manuel Martins
Email: manuelmachadomartins@gmail.com
Homepage: http://github.com/mcmartins

Contents

1 Introduction **3**
- 1.1 GAP Jupyter Kernel . . . 3
- 1.2 Installation . . . 3
- 1.3 How it works . . . 4
- 1.4 Code Highlighting and Indentation . . . 4
- 1.5 Known Limitations and Caveats . . . 4
- 1.6 Feedback . . . 5

2 Jupyter Kernel **6**
- 2.1 Functions . . . 6

3 Jupyter Renderables **7**
- 3.1 Handlers for Jupyter requests . . . 7

4 Jupyter Utility Functions **9**
- 4.1 Functions . . . 9
- 4.2 Additional Utility Functions . . . 9

Index **10**

Chapter 1

Introduction

1.1 GAP Jupyter Kernel

This package provides a so-called *kernel* for the Jupyter interactive document system. This kernel is implemented in GAP.

1.2 Installation

This package requires Jupyter to be installed on your system, which on most Python installations can be achieved by issuing the following command.

Example

```
> pip install --user notebook
```

Further instructions can be found at `https://jupyter.org/install`. This package requires the GAP packages IO, uuid, ZeroMQInterface, crypting, and json, all of which are distributed with GAP, and some of which require compilation. To compile ZeroMQInterface you need to install ZeroMQ, for details please refer to the ZeroMQInterface manual. JupyterKernel itself does not contain any kernel code that needs to be compiled. It is necessary to register JupyterKernel with your Jupyter installation. Registering the GAP jupyter kernel system-wide works as follows:

Example

```
> python setup.py install
```

or registering for your user only

Example

```
> python setup.py install --user
```

If GAP is not in your PATH, then you have to set the environment variable `JUPYTER_GAP_EXECUTABLE` to point to your GAP executable for Jupyter to be able to execute GAP, and the script `jupyter-kernel-gap` that is distributed with this package in the directory `bin/` needs to be in your path. To start Jupyter run:

Example

```
> jupyter notebook
```

Then JupyterKernel should show up in your Jupyter installation as "GAP 4".

1.3 How it works

1.3.1 Kernel Startup

This section gives a short explanation how the process of executing GAP as a kernel by Jupyter works to help with debugging issues. Jupyter registers kernels using json files in various directories. You can list which kernel specifications are installed by executing the following command

Example

```
> jupyter kernelspec list
Available kernels:
python2        /usr/local/lib/python2.7/site-packages/ipykernel/resources
gap-4          /usr/local/share/jupyter/kernels/gap-4
```

If there is no line containing the `gap-4` kernel, something went wrong with `setup.py`. You can try to manually install the file `kernel.json` which is in the `etc` directory of the JupyterKernel package by copying it. Better yet, you should report this issue on the issue tracker giving wich operating system you are using, your version of Jupyter and GAP, and which commands you tried to execute. What happens when Jupyter wants to start a GAP kernel is that it tries to execute the small script `jupyter-kernel-gap` (which is distributed with the JupyterKernel package), which in turn executes GAP, loading the package and then running the kernel. This script currently has to be in your `PATH` environment variable, too.

1.3.2 Kernel operation

The communication between the Jupyter frontend and GAP happens through ZeroMQ streams as documented here, encoded as `JSON` dicts. After entering code into a cell and instructing Jupyter to execute that code, the jupyter frontend sends the code to the GAP session where it is executed by using the GAP function `READ_ALL_COMMANDS`, resulting values of the execution are rendered using `ViewString` and sent back to the Jupyter frontend. In principle, rich rendering of content, as exemplified in the function `JUPYTER_DotSplash` is possible. Tab-completion is handled by the GAP function `JUPYTER_completion`, and inspection is handled by `JUPYTER_Inspect`. Changing these functions, one can change the behaviour of Tab-completion and inspection to improve user experience.

1.4 Code Highlighting and Indentation

JupyterKernel provides a GAP mode with code highlighting and indentation. This mode is installed as a notebook extension and registers 'text/x-gap' as a MIME type.

1.5 Known Limitations and Caveats

Currently the support of the GAP system for alternative frontends is a work in progress. In particular, certain outputs that are printed by GAP cannot be captured by the Jupyter frontend and will not show up. If you happen to notice problems of this kind, feel free to report them on the issue tracker. or suggest a solution via a pull-request.

1.6 Feedback

For bug reports, feature requests and suggestions, please use our issue tracker.

Chapter 2

Jupyter Kernel

A `Jupyter Kernel` is an object that can handles the Jupyter Protocol.

2.1 Functions

2.1.1 JUPYTER_LogProtocol

▷ `JUPYTER_LogProtocol(`*`filename`*`)` (function)

Opens a file that is used to log all jupyter protocol messages.

2.1.2 JUPYTER_UnlogProtocol

▷ `JUPYTER_UnlogProtocol(`*`arg`*`)` (function)

Closes the protocol log.

Chapter 3

Jupyter Renderables

A `JupyterRenderable` is an object that can be rendered by Jupyter. JupyterRenderables are component object that have to contain at least the components `data` and `metadata`.

These components are themselves GAP records which can contain different representations of an object to be rendered. The record component name is the MIME-Type of the representation and the content is the representation itself.

Example

```
render := JupyterRenderable(
      rec( text\/plain := "Integers",
           text\/html := "$\mathbb{Z}$" )
    , rec( ) );

render2 := JupyterRenderable(
      rec( ("image/svg+xml") := "<svg></svg>"
    , rec( ("image/svg+xml") := rec( width := 500, height := 500 ) ) );
```

3.1 Handlers for Jupyter requests

3.1.1 IsJupyterRenderable (for IsObject)

▷ `IsJupyterRenderable(`*arg*`)` (filter)

Returns: `true` or `false`

JupyterRenderable

3.1.2 (for IsComponentObjectRep and IsJupyterRenderable)

▷ `(`*arg*`)` (filter)

Returns: `true` or `false`

3.1.3 JupyterRenderable (for IsObject, IsObject)

▷ `JupyterRenderable(`*data, metadata*`)` (operation)

Returns: A new JupyterRenderable

Basic constructor for JupyterRenderable

3.1.4 JupyterRender (for IsObject)

▷ `JupyterRender(`*arg*`)` (operation)

Method that provides rich viewing experience if code used inside Jupyter

3.1.5 JupyterRenderableData (for IsJupyterRenderable)

▷ `JupyterRenderableData(`*arg*`)` (attribute)

Accessor for data in a JupyterRenderable

3.1.6 JupyterRenderableMetadata (for IsJupyterRenderable)

▷ `JupyterRenderableMetadata(`*arg*`)` (attribute)

Accessor for metadata in a JupyterRenderable

Chapter 4

Jupyter Utility Functions

4.1 Functions

4.1.1 JUPYTER_print

▷ `JUPYTER_print(`*arg*`)` (function)

Jupyter printing

4.1.2 JUPYTER_Complete

▷ `JUPYTER_Complete(`*arg*`)` (function)

This function is called when the user presses Tab in a code cell and produces a list of possible completions. It is passed the current code in the cell, and the curser position inside the code.

4.2 Additional Utility Functions

4.2.1 ISO8601Stamp

▷ `ISO8601Stamp(`*arg*`)` (function)

Current date and time as ISO8601 timestamp. Don't trust this function.

Index

IsJupyterRenderable
    for IsObject, 7
ISO8601Stamp, 9

JupyterRender
    for IsObject, 8
JupyterRenderable
    for IsObject, IsObject, 7
JupyterRenderableData
    for IsJupyterRenderable, 8
JupyterRenderableMetadata
    for IsJupyterRenderable, 8
JUPYTER_Complete, 9
JUPYTER_LogProtocol, 6
JUPYTER_print, 9
JUPYTER_UnlogProtocol, 6