Metakernel Documentation

Release 0.30.1

Metakernel contributors

Sep 11, 2023

CONTENTS

A Jupyter kernel base class in Python which includes core magic functions (including help, command and file path completion, parallel and distributed processing, downloads, and much more).

See Jupyter's docs on wrapper kernels.

Additional magics can be installed within the new kernel package under a *magics* subpackage.

- **Basic set of line and cell magics for all kernels.**
  - Python magic for accessing python interpreter.
  - Run kernels in parallel.
  - Shell magics.
  - Classroom management magics.
- Tab completion for magics and file paths.
- Help for magics using ? or Shift+Tab.
- Plot magic for setting default plot behavior.

CHAPTER

ONE

KERNELS BASED ON METAKERNEL

- matlab_kernel, https://github.com/Calysto/matlab_kernel
- octave_kernel, https://github.com/Calysto/octave_kernel
- calysto_scheme, https://github.com/Calysto/calysto_scheme
- calysto_processing, https://github.com/Calysto/calysto_processing
- java9_kernel, https://github.com/Bachmann1234/java9_kernel
- xonsh_kernel, https://github.com/Calysto/xonsh_kernel
- calysto_hy, https://github.com/Calysto/calysto_hy
- gnuplot_kernel, https://github.com/has2k1/gnuplot_kernel
- spylon_kernel, https://github.com/mariusvniekerk/spylon-kernel
- wolfram_kernel, https://github.com/mmatera/iwolfram
- sas_kernel, https://github.com/sassoftware/sas_kernel
- pysysh_kernel, https://github.com/Jaesin/psysh_kernel
- calysto_bash, https://github.com/Calysto/calysto_bash

. . . and many others.

1.1 Installation

You can install metakernel through *pip*:

pip install metakernel –upgrade

1.2 API Reference

1.2.1 Metakernel

class metakernel.**MetaKernel**(***kwargs: Any*)

The base MetaKernel class.

Create a configurable given a config config.

Parameters

config
[Config] If this is empty, default values are used. If config is a `Config` instance, it will be used to configure the instance.

parent
[Configurable instance, optional] The parent Configurable instance of this object.

Notes

Subclasses of Configurable must call the `__init__()` method of `Configurable` *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config=config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

classmethod run_as_main(**args*, ***kwargs*)

Launch or install a metakernel.

Modules implementing a metakernel subclass can use the following lines:

if __name__ == '__main__':
MetaKernelSubclass.run_as_main()

makeSubkernel(*kernel*)

Run this method in an IPython kernel to set this kernel's input/output settings.

set_variable(*name*, *value*)

Set a variable to a Python-typed value.

get_variable(*name*)

Lookup a variable name and return a Python-typed value.

repr(*item*)

The repr of the kernel.

get_usage()

Get the usage statement for the kernel.

get_kernel_help_on(*info*, *level=0*, *none_on_fail=False*)

Get help on an object. Called by the help magic.

handle_plot_settings()

Handle the current plot settings

get_local_magics_dir()

Returns the path to local magics dir (eg ~/.ipython/metakernel/magics)

get_completions(*info*)

Get completions from kernel based on info dict.

do_execute_direct(*code*, *silent=False*)

Execute code in the kernel language.

do_execute_file(*filename*)

Default code for running a file. Just opens the file, and sends the text to do_execute_direct.

do_execute_meta(*code*)

Execute meta code in the kernel. This uses the execute infrastructure but allows JavaScript to talk directly to the kernel bypassing normal processing.

When responding to the %%debug magic, the step and reset meta commands can answer with a string in the format:

"highlight: [start_line, start_col, end_line, end_col]"

for highlighting expressions in the frontend.

initialize_debug(*code*)

This function is used with the %%debug magic for highlighting lines of code, and for initializing debug functions.

Return the empty string if highlighting is not supported.

do_function_direct(*function_name*, *arg*)

Call a function in the kernel language with args (as a single item).

restart_kernel()

Restart the kernel

do_execute(*code*, *silent=False*, *store_history=True*, *user_expressions=None*, *allow_stdin=False*)

Handle code execution.

https://jupyter-client.readthedocs.io/en/stable/messaging.html#execute

post_execute(*retval*, *code*, *silent*)

Post-execution actions

Handle special kernel variables and display response if not silent.

do_history(*hist_access_type*, *output*, *raw*, *session=None*, *start=None*, *stop=None*, *n=None*, *pattern=None*, *unique=False*)

Access history at startup.

https://jupyter-client.readthedocs.io/en/stable/messaging.html#history

do_shutdown(*restart*)

Shut down the app gracefully, saving history.

https://jupyter-client.readthedocs.io/en/stable/messaging.html#kernel-shutdown

do_is_complete(*code*)

Given code as string, returns dictionary with 'status' representing whether code is ready to evaluate. Possible values for status are:

'complete' - ready to evaluate 'incomplete' - not yet ready 'invalid' - invalid code 'unknown' - unknown; the default unless overridden

Optionally, if 'status' is 'incomplete', you may indicate an indentation string.

Example:

return {'status'
['incomplete',] 'indent': ' ' * 4}

https://jupyter-client.readthedocs.io/en/stable/messaging.html#code-completeness

do_complete(*code*, *cursor_pos*)

Handle code completion for the kernel.

https://jupyter-client.readthedocs.io/en/stable/messaging.html#completion

do_inspect(*code*, *cursor_pos*, *detail_level=0*, *omit_sections=()*)

Object introspection.

https://jupyter-client.readthedocs.io/en/stable/messaging.html#introspection

clear_output(*wait=False*)

Clear the output of the kernel.

Display(**objects*, ***kwargs*)

Display one or more objects using rich display.

Supports a *clear_output* keyword argument that clears the output before displaying.

See https://ipython.readthedocs.io/en/stable/config/integrating.html?highlight=display#rich-display

Print(**objects*, ***kwargs*)

Print *objects* to the iopub stream, separated by *sep* and followed by *end*.

Items can be strings or *Widget* instances.

Write(*message*)

Write message directly to the iopub stdout with no added end character.

Error(**objects*, ***kwargs*)

Print *objects* to stdout, separated by *sep* and followed by *end*.

Objects are cast to strings.

Error_display(**objects*, ***kwargs*)

Print *objects* to stdout is they area strings, separated by *sep* and followed by *end*. All other objects are rendered using the Display method Objects are cast to strings.

reload_magics()

Reload all of the line and cell magics.

register_magics(*magic_klass*)

Register magics for a given magic_klass.

send_response(**args*, ***kwargs*)

Send a response to the message we're currently processing.

This accepts all the parameters of `jupyter_client.session.Session.send()` except `parent`.

This relies on `set_parent()` having been called for the current message.

call_magic(*line*)

Given an line, such as "%download http://example.com/", parse and execute magic.

get_help_on(*expr*, *level=0*, *none_on_fail=False*, *cursor_pos=-1*)

Get help for an expression using the help magic.

parse_code(*code*, *cursor_start=0*, *cursor_end=-1*)

Parse code using our parser.

class metakernel.**Magic**(*kernel*)

Base class to define magics for MetaKernel based kernels.

Users can redefine the default magics provided by Metakernel by creating a module with the exact same name as the Metakernel magic.

For example, you can override %matplotlib in your kernel by writing a new magic inside magics/matplotlib_magic.py

get_completions(*info*)

Get completions based on info dict from magic.

metakernel.**option**(**args*, ***kwargs*)

Return decorator that adds a magic option to a function.

1.3 Information

1.3.1 Line Magics

%activity

(poll, classroom response, clicker-like activity)

This magic will load the JSON in the filename.

Examples: %activity /home/teacher/activity1 %activity /home/teacher/activity1 new %activity /home/teacher/activity1 edit

%cd

This line magic is used to change the directory of the notebook or console.

Note that this is not the same directory as used by the %shell magics.

Example: %cd ..

%connect_info

This line magic will show the connection information for this language kernel instance. This information is only necessary if you are interested in making additional connections to the running kernel.

Example: %connect_info

Paste the given JSON into a file, and connect with:

```
$> ipython <app> --existing <file>
```

or, if you are local, you can connect with just:

```
$> ipython <app> --existing %(key)s
```

or even just: $> ipython –existing

if this is the most recent Jupyter session you have started.

%conversation

%dot

This line magic will render the Graphiz CODE, and render it as an image.

Example: %dot graph A { a->b };

%download

This line magic will download and save a file. By default it will use the same filename as the URL. You can give it another name using -f.

Examples: %%download http://some/file/from/internet.txt -f myfile.txt %%download http://some/file/from/program.ss

Options:

-f –filename use the provided name as filename [default: None]

%edit

This line magic will open the file in the next cell, and allow you edit it.

This is a shortcut for %load, and appending a "%%file" as first line.

Example: %edit myprogram.ss

%get

This line magic is used to get a variable.

Examples: %get x

%help

This is MetaKernel Python. It implements a Python interpreter.

%html

This line magic will send the CODE to the browser as HTML.

Example: %html This is underlined!

%include

This line magic will get the contents of a file and include it in this cell evaluation.

You can have multiple %include's at the beginning of a cell, and they will be included in order.

Examples: %include myprog.py %include myprog1.py myprog2.py

%install

Example: %install calico-spell-check

%install_magic

This line magic will copy the file at the URL into your personal magic folder.

Example: %install_magic http://path/to/some/magic.py

%javascript

This line magic will execute the CODE on the line as JavaScript in the browser.

Example: %javascript console.log("Print in the browser console")

%jigsaw

This line magic will allow visual code editing or generation.

Examples: %jigsaw Processing %jigsaw Python %jigsaw Processing –workspace workspace1 –height 600

Options:

-h –height set height of iframe [default: 350] -w –workspace use the provided name as workspace filename [default: None]

%kernel

This line magic will construct a kernel language so that you can communicate.

Example:

```
%kernel bash_kernel BashKernel -k bash
```

Use `%kx` or `%%kx` to send code to the kernel.

Also returns the kernel as output.

Options:

-k –kernel_name kernel name given to use for execution [default: default]

%kx

This line magic will send the CODE to the kernel for execution.

Returns the result of the execution as output.

Example:

```
%kernel ls -al
```

Use `%kernel MODULE CLASS [-k NAME]` to create a kernel.

Options:

-k –kernel_name kernel name given to use for execution [default: None]

%latex

This line magic will display the TEXT on the line as LaTeX.

Example: %latex $x_1 = \dfrac{a}{b}$

%load

This line magic will get the contents of a file and load it into the next cell.

Example: %load myprog.py

%ls

This line magic is used to list the directory contents.

Examples: %ls . %ls ..

Options:

-r –recursive recursively descend into subdirectories [default: False]

%lsmagic

This line magic will list all of the available cell and line magics installed in the system and in your personal magic folder.

Example: %lsmagic

%macro

This line macro will execute, show, list, or delete the named macro.

Examples: %macro renumber-cells

```
%%macro test
print "Ok!"

%macro -l all

%macro -d test
```

Options:

-s –show show macro [default: False] -l –list list macros [default: False] -d –delete delete a named macro [default: False]

%magic

This line magic shows all of the install magics, either from the system magic folder, or your own private magic folder.

%matplotlib

This line magic will set (and reload) the items associated with the matplotlib backend.

Also, monkeypatches the IPython.display.display to work with metakernel-based kernels.

Example: %matplotlib notebook

```
import matplotlib.pyplot as plt
plt.plot([3, 8, 2, 5, 1])
plt.show()
```

%parallel

Example:

```
%parallel bash_kernel BashKernel
%parallel bash_kernel BashKernel -k bash
%parallel bash_kernel BashKernel -i [0,2:5,9,...]
```

cluster_size and cluster_rank variables are set upon initialization of the remote node (if the kernel supports %set).

Use %px or %%px to send code to the cluster.

Options:

-i –ids the machine ids to use from the cluster [default: None] -k –kernel_name arbitrary name given to reference kernel [default: default]

%plot

This line magic will configure the plot settings for this language.

Examples: %plot qt –format=png %plot inline -w 640

Note: not all languages may support the %plot magic, and not all options may be supported.

Options:

-h –height Plot height in pixels -w –width Plot width in pixels -r –resolution Resolution in pixels per inch -b –backend Backend selection [default: inline] -f –format Plot format (png, svg or jpg). -s –size Pixel size of plots, "width,height"

%pmap

This line magic will apply a function name to all of the arguments given one at a time using a dynamic load balancing scheduler.

Currently, the args are provided as a Python expression (with no spaces).

You must first setup a cluster using the %parallel magic.

Examples:

```
%pmap function-name-in-language range(10)
%pmap function-name-in-language [1,2,3,4]
%pmap run_experiment range(1,100,5)
%pmap run_experiment ["test1","test2","test3"]
%pmap f [(1,4,7),(2,3,5),(7,2,2)]
```

The function name must be a function that is available on all nodes in the cluster. For example, you could:

```
%%px
(define myfunc
   (lambda (n)
     (+ n 1)))
```

to define myfunc on all machines (use %%px -e to also define it in the running notebook or console). Then you can apply it to a list of arguments:

```
%%pmap myfunc range(100)
```

The load balancer will run myfunc on the next available node in the cluster.

Note: not all languages may support running a function via this magic.

Options:

-s –set_variable set the variable with the parallel results rather than returning them [default: None]

%px

Example:

```
%px sys.version
%px -k scheme (define x 42)
%px x
%px cluster_rank
```

cluster_size and cluster_rank variables are set upon initialization of the remote node (if the kernel supports %set).

Use %parallel to initialize the cluster.

Options:

-s –set_variable set the variable with the parallel results rather than returning them [default: None] -e –evaluate evaluate code in the current kernel, too. The current kernel should be of the same language as the cluster. [default: False] -k –kernel_name kernel name given to use for execution [default: None]

%python

This line magic will evaluate the CODE (either expression or statement) as Python code.

Note that the version of Python is that of the notebook server.

Examples: %python x = 42 %python import math %python x + math.pi

%reload_magics

Example: %reload_magics

This line magic will reload the magics installed in the system, and in your private magic folder.

You only need to do this if you edit a magic file. It runs automatically if you install a new magic.

%restart

This line magic will restart the connection to the language kernel.

Example: %restart

Note that you will lose all computed values.

%run

kernel

This magic will take the code in FILENAME and run it. The exact details of how the code runs are determined by your language.

The –language LANG option will prefix the file contents with "%%LANG". You may also put information in the cell which will appear before the contents of the file.

Examples: %run filename.ss %run -l python filename.py

```
%kx calysto_scheme.kernel CalystoScheme
%run --language kx filename.ss
%run --language "kx default" filename.ss
```

Note: not all languages may support %run.

Options:

-l –language use the provided language name as kernel [default: None]

%scheme

This line magic will evaluate the CODE (either expression or statement) as Scheme code.

Examples: %scheme (define x 42) %scheme (import "math") %scheme (+ x + math.pi)

%set

This line magic is used to set a variable to a Python value.

Examples: %set x 42 %set x [1, 2, 3]

%shell

This line command will run the COMMAND in the bash shell.

Examples: %shell ls -al %shell cd

Note: this is a persistent connection to a shell. The working directory is synchronized to that of the notebook before and after each call.

You can also use "!" instead of "%shell".

1.3.2 Cell Magics

%%activity

a JSON structure

This magic will construct a Python file from the cell's content, a JSON structure.

Example: %%activity /home/teacher/activity1 {“activity”: “poll”, “instructors”: [“teacher01”], “results_file”: “/home/teacher/activity1.results”, “items”: [{“id”: “…”, “type”: “multiple choice”, “question”: “…”, “options”: [”…”, …] }, …] }

In this example, users will load /home/teacher/activity1

%%brain

for a calysto.simulation.

Requires calysto.

Examples:

robot.forward(1)

%%conversation

%%debug

This cell magic will step through the code in the cell, if the kernel supports debugging.

Example: %%debug

```
(define x 1)
```

%%dot

This cell magic will send the cell to the browser as HTML.

Example: %%dot

```
graph A { a->b };
```

%%file

This cell magic will create or append the cell contents into/onto a file.

Example: %%file -a log.txt This will append this line onto the file “log.txt”

Options:

-a –append append onto an existing file [default: False]

%%help

This is MetaKernel Python. It implements a Python interpreter.

%%html

This cell magic will send the cell to the browser as HTML.

Example: %%html

```
<script src="..."></script>

<div>Contents of div tag</div>
```

%%javascript

This cell magic will execute the contents of the cell as JavaScript in the browser.

Example: %%javascript

```
element.html("Hello this is <b>bold</b>!")
```

%%kx

This cell magic will send the cell to be evaluated by the kernel. The kernel must have been created use the

Returns the result of the execution as output.

Example:

```
%%kernel bash
ls -al
```

Use `%kernel MODULE CLASS [-k NAME]` to create a kernel.

Options:

-k –kernel_name kernel name given to use for execution [default: None]

%%latex

This cell magic will display the TEXT in the cell as LaTeX.

Example: %%latex $x_1 = \dfrac{a}{b}$

```
$x_2 = a^{n - 1}$
```

%%macro

This cell macro will learn the macro in the cell. The cell contents are just commands (macros or code in the kernel language).

Example: %%macro test print "Ok!"

```
%macro test
Ok!
```

%%pipe

The pipe cell will "pipe" the contents of a cell through a series of function calls. All of the functions must be defined in the language, and the kernel must support the `do_function_direct` method.

Example: %%pipe f1 | f2 | f3 CELL CONTENTS

```
is the same as issuing:

f3(f2(f1("CELL CONTENTS")))
```

%%processing

This cell magic will execute the contents of the cell as a Processing program. This uses the Java-based Processing language.

Example:

```
%%processing
setup() {
}
draw() {
}
```

%%px

Example:

```
%%px
(define x 42)
```

Use %parallel to initialize the cluster.

Options:

-s –set_variable set the variable with the parallel results rather than returning them [default: None] -e –evaluate evaluate code in the current kernel, too. The current kernel should be of the same language as the cluster. [default: False] -k –kernel_name kernel name given to use for execution [default: None]

%%python

This cell magic will evaluate the cell (either expression or statement) as Python code.

Unlike IPython's Python, this does not return the last expression. To do that, you need to assign the last expression to the special variable "retval".

The -e or –eval_output flag signals that the retval value expression will be used as code for the cell to be evaluated by the host language.

Note that the version of Python is that of the notebook server.

Examples: %%python x = 42

```
%%python
import math
retval = x + math.pi

%%python -e
retval = "'(this is code in the kernel language)"

%%python -e
"'(this is code in the kernel language)"
```

Options:

-e –eval_output Use the retval value from the Python cell as code in the kernel language. [default: False]

%%scheme

This cell magic will evaluate the cell (either expression or statement) as Scheme code.

The -e or –eval_output flag signals that the retval value expression will be used as code for the cell to be evaluated by the host language.

Examples: %%scheme (define x 42)

```
%%scheme
(import "math")
(define retval (+ x math.pi))

%%scheme -e
(define retval "this = code")

%%scheme -e
"this = code"
```

Options:

-e –eval_output Use the retval value from the Scheme cell as code in the kernel language. [default: False]

%%shell

This shell command will run the cell contents in the bash shell.

Example: %%shell cd .. ls -al

Note: this is a persistent connection to a shell. The working directory is synchronized to that of the notebook before and after each call.

You can also use "!!" instead of "%%shell".

%%show

This cell magic will put the contents or results of the cell into the system pager.

Examples: %%show This information will appear in the pager.

```
%%show --output
retval = 54 * 54
```

Options:

-o –output rather than showing the contents, show the results [default: False]

%%time

Put this magic at the top of a cell and the amount of time taken to execute the code will be displayed before the output.

Example: %%time [code for your language goes here!]

This just reports real time taken to execute a program. This may fluctuate with number of users, system, load, etc.

%%tutor

Online Python Tutor.

Defaults to use the language of the current kernel. 'python' is an alias for 'python3'.

Examples:

a = 1 b = 1 a + b

[You will see an iframe with the pythontutor.com page including the code above.]

public class Test { public Test() { } public static void main(String[] args) { int x = 1; System.out.println("Hi"); } }

Options:

-l –language Possible languages to be displayed within the iframe. Possible values are: python, python2, python3, java, javascript

CHAPTER

TWO

API REFERENCE

Documentation for the functions included in Jupyter Kernel.

CHAPTER

THREE

INSTALLATION

How to install Jupyter Kernel.

CHAPTER

FOUR

INFORMATION

Other information about Jupyter Kernel.

PYTHON MODULE INDEX

m

metakernel, 3

INDEX