# IPython-extensions Documentation

***Release 0.1.0***

**The IPython-extensions Development Team**

November 02, 2015

Contents

IPython-extensions requires Python 2.7 or 3.3+.

**See also:**

**Installing IPython** How to install IPython

**Installing the Jupyter notebook** The Notebook, nbconvert, and many other former pieces of IPython are now part of Project Jupyter.

CHAPTER 1

# Installation

## 1.1 Quickstart

If you have `pip`, the quickest way to get up and running with IPython is:

```
$ pip install ipyext
```

## 1.2 Installing the development version

It is also possible to install the development version of IPython-extensions from our Git source code repository. To do this you will need to have Git installed on your system. Then do:

```
$ git clone https://github.com/ipython-contrib/IPython-extensions.git
$ cd IPython-extensions
$ python setup.py install
```

Some users want to be able to follow the development branch as it changes. If you have `pip`, you can replace the last step by:

```
$ pip install -e .
```

This creates links in the right places and installs the command line script to the appropriate places.

Then, if you want to update your IPython at any time, do:

```
$ git pull
```

## 1.3 Dependencies

IPython-extensions relies on a number of other Python packages. Installing using a package manager like pip or conda will ensure the necessary packages are installed. If you install manually, it's up to you to make sure dependencies are installed. They're not listed here, because they may change from release to release, so a static list will inevitably get out of date.

CHAPTER 2

# Magic commands

## 2.1 Line magics

## 2.2 Cell magics

**%%inactive**

Magic to *not* execute a cell.

This magic can be used to mark a cell (temporary) as inactive.

Examples:

```
In [1]: %load_ext ipyext.inactive
'inactive' magic loaded.

In [2]: %%inactive
   ...: print("code not run...")
   ...:
Cell inactive: not executed!
```

**%%writeandexecute**

Writes the content of the cell to a file and then executes it.

This magic can be used to write the content of a cell to a .py file and afterwards execute the cell. This enables to code reuse.

Code is replaced on the next execution (using the needed identifier) and other code can be appended by using the same file name.

Parameters

**-i <identifier>** [str] Surrounds the code written to the file with a line containing the identifier. The use of an identifier enables you to easily overwrite a given code section in the output file.

**<filename>** [str] The file to which the code should be written. Can be specified without an extension and can also include a directory (`dir/file`)

**-d** [(optional)] Write some debugging output. Default: – (no debugging output)

Examples:

```
In [1]: %load_ext ipyext.writeandexecute
'writeandexecute' magic loaded.

In [2]:  %%writeandexecute -i my_code_block functions.py
   ...:  print "Hello world"
   ...:
Hello world
```

In addition to executing (and printing "Hello world"), it would also create a file "functions.py" with the following content

```
# -*- coding: utf-8 -*-


# -- ==my_code_block== --
print "Hello world"

# -- ==my_code_block== --
```

Cell content is transformed, so `%magic` commands are executed, but `get_ipython()` must be available, i.e. the code must be executed in an IPython session.

CHAPTER 3

# What's new in IPython-extensions

This section documents the changes that have been made in various versions of IPython-extensions. Users should consult these pages to learn about new features, bug fixes and backwards incompatibilities. Developers should summarize the development work they do here in a user friendly format.

## 3.1 Release 0.1.0

- Initial release of this package
- Add `%%inactive` to render a cell (temporary) inactive
- Add `%%writeandexecute` to write the a cell to a file and execute it (-> Code reuse)

## C



## I



## W