switowski

Creating Magic Functions in IPython - Part 3

15 Feb 2019 8 minutes read #Python #IPython

So far in this series, we have covered three different decorators: `@register_line_magic` (in part1), `@register_cell_magic` and `@register_line_cell_magic` (in part2). Which is enough to create any type of magic function in IPython. But, IPython offers another way of creating them - by making a **Magics** class and defining magic functions within it.

Magics classes

Magics classes are more powerful than functions, in the same way that a class is more powerful than a function. They can hold state between function calls, encapsulate functions, or offer you inheritance. To create a Magics class, you need three things:

- Your class needs to inherit from `Magics`
- Your class needs to be decorated with `@magics_class`
- You need to register your magic class using the `ipython.register_magics(MyMagicClass)` function

In your magic class, you can decorate functions that you want to convert to magic functions with `@line_magic`, `@cell_magic` and `@line_cell_magic`,

Writing a magics class

To show how the magics class works, we will create another version of `mypy` helper. This time, it will allow us to run type checks on the previous cells. This is how we expect it to work:

PYTHON

```
In [1]: def greet(name: str) -> str:
   ...:     return f"hello {name}"

In [2]: greet('tom')
Out[2]: 'hello tom'

In [3]: greet(1)
Out[3]: 'hello 1'

In [4]: %mypy 1-2
Out[4]: # Everything should be fine

In [4]: %mypy 1-3
Out[4]: # It should report a problem on cell 3
```

Here are a few assumptions about the `%mypy` function:

- It should accept all the parameters that the `mypy` command accepts
- It should accept the same range parameters that `%history` command accepts, but **only from the current session**. I usually don't reference history from the previous sessions anyway and it will make parsing arguments slightly easier. So `1`, `1-5`, and `1 2 4-5` are all valid arguments, while `243/1-5` or `~8/1-~6/5` are not.

- The order of arguments doesn't matter (and you can even mix ranges with `mypy` arguments), so we can call our function in the following ways:
  - `%mypy --ignore-imports 1 2 5-7`
  - `%mypy 1-3`
  - `%mypy 2 4 5-9 --ignore-imports`
  - `%mypy 2 4 --ignore-imports 5-9`

With that in mind, let's write the code. The main class looks like this:

```python
from IPython.core.magic import Magics, magics_class, line
import re

# The class MUST call this class decorator at creation ti
@magics_class
class MypyMagics(Magics):
    @line_magic
    def mypy(self, line):
        try:
            from mypy.api import run
        except ImportError:
            return "'mypy' not installed. Did you run 'pi

        if not line:
            return "You need to specify cell range, e.g.

        args = line.split()
        # Parse parameters and separate mypy arguments fr
        mypy_arguments = []
        cell_numbers = []
        for arg in args:
            if re.fullmatch(r"\d+(-\d*)?", arg):
                # We matched either "1" or "1-2", so it's
                cell_numbers.append(arg)
            else:
                mypy_arguments.append(arg)

        # Get commands from a given range of history
        range_string = " ".join(cell_numbers)
        commands = _get_history(range_string)

        # Run mypy on that commands
        print("Running type checks on:")
        print(commands)

        result = run(["-c", commands, *mypy_arguments])

        if result[0]:
            print("\nType checking report:\n")
            print(result[0])  # stdout

        if result[1]:
            print("\nError report:\n")
            print(result[1])  # stderr

        # Return the mypy exit status
        return result[2]


ip = get_ipython()
ip.register_magics(MypyMagics)
```

We have the `MypyMagics class` (that inherits from `Magics`) and in it, we have the `mypy` line magic that does the following:

- checks if `mypy` is installed
- if there were no arguments passed - it returns a short information on how to use it correctly.

- parses the arguments and splits those intended for `mypy` from the cell numbers/ranges. Since `mypy` doesn't accept arguments that look like a number (`1`) or range of numbers (`1-2`), we can safely assume that all arguments that match one of those 2 patterns, are cells.
- retrieves the input values from the cells using the `_get_history` helper (explained below) as a string, and prints that string to the screen, so you can see what code will be checked.
- runs the `mypy` command, prints the report and returns the exit code.

At the end, we need to remember to register the `MypyMagics` class in IPython.

We are using one helper function on the way:

PYTHON

```
def _get_history(range_string):
    ip = get_ipython()
    history = ip.history_manager.get_range_by_str(range_s
    # history contains tuples with the following values:
    # (session_number, line_number, input value of that l
    # We only need the input values concatenated into one
    # with trailing whitespaces removed from each line
    return "\n".join([value.rstrip() for _, _, value in h
```

I told you before, that when writing a class, we can put our helper function inside, but I'm purposefully keeping this one outside of the `MypyMagics`. It's a simple helper that can be used without any knowledge about our class, so it doesn't really belong in it. So, I'm keeping it outside and using the naming convention to suggest that it's a private function.

Coming up with the `_get_history` helper was quite a pickle, so let's talk a bit more about it.

Approach 1: `_ih`

I needed to retrieve the previous commands from IPython, and I knew that IPython stores them in `_ih` list (so, if you want to retrieve, let's say, the first command from the current session, you can just run `_ih[1]`). It sounded easy, but it required some preprocessing. I would first have to translate `1-2` type of ranges into list slices. Then I would have to retrieve all parts of the history, one by one, so for `1 2-3 5`, I would need to call `_ih[1]`, `_ih[2:4]`, `_ih[5]`. It was doable, but I wanted an easier way.

Approach 2: `%history`

My next idea was to reuse the `%history` magic function. While you can't just write `%history` in Python code and expect it to work, there is a different way to call magics as standard functions - I had to use the `get_ipython().magic(<func_name>)` function.

Problem solved! Except that `%history` magic can either print the output to the terminal or save it in a file. There is no way to convince it to *return* us a string. Bummer! I could overcome this problem in one of the following 2 ways:

- Since by default `%history` writes to `sys.stdout`, I could monkey-patch (change the behavior at runtime) the `sys.stdout` and make it save the content of `history` output in a variable. Monkey patching is usually not the best idea and I didn't want to introduce bad practices in my code, so I didn't like this solution.
- Otherwise, I could save the output of `%history` to a file and then read it from that file. But creating files on a filesystem just to write something inside and immediately read it back, sounds terrible. I would need to worry about where to create the file, whether or not the file already exists, then remember to delete it.

> Even with tempfile module that can handle the creation and deletion of temporary file for me, that felt like too much for a simple example.

So the `%history` function was a no-go.

Approach 3: `HistoryManager`

Finally, I decided to peak inside the `%history` and use whatever that function was using under the hood - the HistoryManager from `IPython.core.history` module. `HistoryManager.get_range_by_str()` accepts the same string formats that `%history` function does, so no preprocessing was required. That was exactly what I needed! I only had to clean the output a bit (retrieve the correct information from the tuples) and I was done.

Testing time

Now, that our `%mypy` helper is done (the whole file is available on GitHub) and saved in the IPython startup directory, let's test it:

PYTHON

```
In [1]: def greet(name: str) -> str:
   ...:     return f"hello {name}"
   ...:

In [2]: greet('Bob')
Out[2]: 'hello Bob'

In [3]: greet(1)
Out[3]: 'hello 1'

In [4]: %mypy 1-3  # this is equivalent to `%mypy 1 2 3`
Running type checks on:
def greet(name: str) -> str:
    return f"hello {name}"
greet('Bob')
greet(1)

Type checking report:

<string>:4: error: Argument 1 to "greet" has incompatible

Out[4]: 1

# What about passing parameters to mypy?
In [5]: import Flask

In [6]: %mypy 5
Running type checks on:
import flask

Type checking report:

<string>:1: error: No library stub file for module 'flask
<string>:1: note: (Stub files are from https://github.com,

Out[6]: 1

In [7]: %mypy 5 --ignore-missing-imports
Running type checks on:
import flask
Out[7]: 0
```

Perfect, it's working exactly as expected! You now have a helper that will check types of your code, directly in IPython.

There is only one thing that could make this even better - an **automatic** type checker that, once activated in IPython, will automatically type check your code as you execute it. But that's a story for another article.

Conclusions

This the end of our short journey with IPython magic functions. As you can see, there is nothing *magical* about them, all it takes is to add a decorator or inherit from a specific class. Magic functions can further extend the already amazing capabilities of IPython. So, don't hesitate to create your own, if you find yourself doing something over and over again. For example, when I was working a lot with SQLAlchemy, I made a magic function that converts an sqlalchemy row object to Python dictionary. It didn't do much, except for presenting the results in a nice way, but boy, what a convenience that was, when playing with data!

Do you know any cool magic functions that you love and would like to share with others? If so, you can always send me an email or find me on Twitter!

Image from: pixabay

Similar posts

IPython Extensions Guide

What are IPython extensions, how to install them, and how to write and publish your own extension?

15 Oct 2019

5 Ways of Debugging with IPython

Tips and tricks on how to use IPython as your debugger.

23 Dec 2019

Automatically Reload Modules with %autoreload

Tired of having to reload a module each time you change it? %autoreload to the rescue!

01 Oct 2019

Tags

#11ty #CLI #Conference #Excel #git #IPython #Obsidian #Productivity #Project Management #Python #Slides #Software - Y U so hard?! #Speaking #Tools #VS Code #Writing #Writing Faster Python

Previous: Creating Magic Functions in IPython - Part 2

Next: Wait, IPython Can Do That?!

0 comments

Write Preview Aa

Sign in to comment