switowski

# Creating Magic Functions in IPython - Part 2

08 Feb 2019 8 minutes read #Python #IPython

In the previous post, I explained what the magic functions are and why they are cool. We have also created a **line magic** function that interprets mathematical formulas written in Polish notation. Today, we will talk about **cell magic** functions.

## Cell magics in IPython #

Cell magics are similar to line magics, except that they work on cells (blocks of code), not on single lines. IPython comes with a few predefined ones and most of them will let you interpret code written in a different programming language. Need to run some Python 2 code, but IPython is using Python 3 by default? No problem, just type `%%python2`, paste/type the code and run it:

```
PYTHON
In [1]: print 'hello there'
  File "<ipython-input-1-202d533f5f80>", line 1
    print 'hello there'
                      ^
SyntaxError: Missing parentheses in call to 'print'. Did 

# But!

In [2]: %%python2
   ...: print 'hello there'
   ...:
   ...:
hello there
```

You can also run code written in Ruby, Bash, JavaScript, and other languages. And those different blocks of code can interact with each other, for example, you can run some JavaScript code and send variables back to Python.

## Writing a cell magic function

Now, let's try to write our own cell magic function. I initially wanted to continue with the example of Polish notation from the first part of the series. So I started writing a function that translates all the mathematical operations in a block of code into a Polish notation form. Unfortunately, I quickly realized that if I want to write a good example (not some half-assed code that works only for `+` and `-`), I would have to write a proper interpreter. And that would no longer be a simple example[1]. So this time, we are going to do something different.

One of the new features that came in Python in version 3.5 are **type hints**. Some people like them, some people don't (which is probably true for *every* new feature in *every* programming language). The nice thing about Python type hints is that they are not mandatory. If you don't like them - don't use them. For fast prototyping or a project that you are maintaining yourself, you are probably fine without them. But for a large code base, with plenty of legacy code maintained by multiple developers - type hints can be tremendously helpful!

As you are probably starting to guess, our cell magic function will check types for a block of code. Why? Well, with IPython, you can quickly prototype some code, tweak it and save it to a file using the %save or %%writefile magic functions (or simply copy and paste it, if it's faster for you). But, at the time of writing this article, there is no built-in type checker in Python. The mypy library is a *de facto* static type checker, but it's still an external tool that you run from shell ( `mypy filename.py` ). So let's make a helper that will allow us to type check Python code directly in IPython!

This is how we expect it to work:

```python
In [1]: %%mypy
   ...: def greet(name: str) -> str:
   ...:     return f"hello {name}"
   ...: greet(1)
   ...:
   ...:
Out[1]: # It should print an error message, as 1 is not a
```

To achieve this, we will simply call the `run` function from `mypy.api` (as suggested in the documentation) and pass the `-c PROGRAM_TEXT` parameter that checks a string.

Here is the code for the type checker:

```python
from IPython.core.magic import register_cell_magic

@register_cell_magic('mypy')
def typechecker(line, cell):
    try:
        from mypy.api import run
    except ImportError:
        return "'mypy' not installed. Did you run 'pip in

    args = []
    if line:
        args = line.split()

    result = run(['-c', cell, *args])

    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]
```

Let's go through the code, given that there are a few interesting bits:

```python
@register_cell_magic(mypy)
def typechecker(line, cell):
```

We start by defining a function called `typechecker` and registering it as a cell magic function called `%%mypy` . Why didn't I just define a function called `mypy` instead of doing this renaming? Well, if I did that, then **our** `mypy` function would shadow the `mypy` module. In this case, it probably won't cause any problems. But in general, you should avoid shadowing variables/functions/modules, because one day, it will cause you a lot of headache.

```python
try:
    from mypy.api import run
```

```
except ImportError:
    return "`mypy` not found. Did you forget to run `pip 
```

Inside our function, we first try to import the `mypy` module. If it's not available, we inform the user that it should be installed, before this magic function can be used. The nice thing about importing `mypy` in the `typechecker` function is that the import error will show up only when you run the magic function. If you put the import at the top of the file, then save the file inside IPython startup directory, and you **don't** have `mypy` module installed, you will get the `ImportError` every time you start IPython. The downside of this approach is that you are running the import code every time you run the `typechecker` function. This is something that you should avoid doing, if you care about the performance, but in case of our little helper, it's not a big problem.

If you are using Python 3.6 or higher, you can catch the `ModuleNotFoundError` error instead of `ImportError`. `ModuleNotFoundError` is a new subclass of `ImportError` thrown when a module can't be located. I want to keep my code compatible with lower versions of Python 3, so I will stick to the `ImportError`.

PYTHON

```
args = []
if line:
    args = line.split()

result = run(['-c', cell, *args])
```

Note that the function used for defining a cell magic must accept both a `line` and `cell` parameter. Which is great, because this way, we can actually pass parameters to `mypy`! So here, we are passing additional arguments from the `line` parameter to the `run` function. Here is how you could run our magic function with different settings:

PYTHON

```
In [1]: %%mypy --ignore-missing-imports --follow-imports 
   ...: CODEBLOCK
```

which is equivalent to running the following command in the command line: `mypy --ignore-missing-imports --follow-imports error -c 'CODEBLOCK'`.

The rest of the code is quite similar to the example from the documentation.

## Testing time

Our cell magic function is ready. Let's save it in the IPython startup directory (what's IPython startup directory?), so it will be available next time we start IPython. In my case, I'm saving it in a file called:

SHELL

```
~/.ipython/profile_default/startup/magic_functions.py
```

Now, let's fire up IPython and see if it works:

PYTHON

```
In [1]: %%mypy
   ...: def greet(name: str) -> str:
   ...:     return f"hello {name}"
   ...: greet('Bob')
   ...:
   ...:
Out[1]: 0
```

```
In [2]: %%mypy
   ...: def greet(name: str) -> str:
   ...:     return f"hello {name}"
   ...: greet(1)
   ...:
   ...:

Type checking report:

<string>:3: error: Argument 1 to "greet" has incompatible

Out[2]: 1
```

Great, it works! It returns 0 (which is a standard UNIX exit code for a successful command) if everything is fine. Otherwise, it reports what problems have been found.

How about passing some additional parameters?

PYTHON

```
In [3]: %%mypy
   ...: 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[3]: 1

# Ok, this can happen (https://mypy.readthedocs.io/en/lat
# Let's ignore this error

In [4]: %%mypy --ignore-missing-imports
   ...: import flask
   ...:
   ...:
Out[4]: 0
```

Passing additional parameters also works!

Great, we created a nice little helper function that we can use for checking, if the type hints are correct in a given block of code.

## Line and cell magic function

There is one more decorator that we didn't discuss yet: `@register_line_cell_magic`. It's nothing special - especially now that you know how line magic and cell magic works - so there is no need for a separate article. IPython documentation explains this decorator very well:

PYTHON

```
@register_line_cell_magic
def lcmagic(line, cell=None):
    "Magic that works both as %lcmagic and as %%lcmagic"
    if cell is None:
        print("Called as line magic")
        return line
    else:
        print("Called as cell magic")
        return line, cell
```

If you run `%lcmagic`, this function won't receive the `cell` parameter and it will act as a line magic. If you run `%%lcmagic`, it will receive the `cell` parameter and - optionally - the `line` parameter (like in our last example with `%%mypy`). So you can check for the presence of `cell` parameter and based on that, control if it should act as a line or cell magic.

## Conclusion

Now you know how to make a **line magic** and a **cell magic** functions and how to combine them together into a **line and magic** function. There is still one more feature that IPython offers - the **Magics class**. It allows you to write more powerful magic functions, as they can, for example, hold state in between calls. So stay tuned for the last part of this article!

Image from: Pexels

1. Writing a translator is still a great exercise! I recently followed the Let's Build A Simple Interpreter series, where you would build a Pascal interpreter in Python, and it was a really fun project for someone who never studied the compilers. So, if you are interested in this type of challenge, that blog can help you get started. ↩

### Don't miss new articles

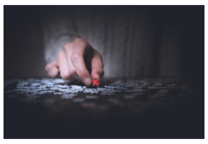
Your name (optional)

Your email

Subscribe

No spam, unsubscribe with one click.

## Similar posts



**IPython Extensions Guide**

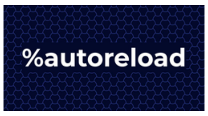
What are IPython extensions, how to install them, and how to write and publish your own extension?

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 1

Next: Creating Magic Functions in IPython - Part 3

0 comments

Write Preview Aa

Sign in to comment

© 2022 Sebastian Witowski. All rights reserved.
Built with ♥, sweat, tears, 11ty, and other technologies.