switowski

Creating Magic Functions in IPython - Part 1

01 Feb 2019 5 minutes read #Python #IPython

IPython magic functions

One of the cool features of IPython are magic functions - helper functions built into IPython. They can help you easily start an interactive debugger, create a macro, run a statement through a code profiler or measure its' execution time and do many more common things.

> Don't mistake **IPython** magic functions with **Python** magic functions (functions with leading and trailing double underscore, for example `__init__` or `__eq__`) - those are completely different things! In this and next parts of the article, whenever you see a **magic function** - it's an IPython magic function.

Moreover, you can create your own magic functions. There are 2 different types of magic functions.
The first type - called **line magics** - are prefixed with `%` and work like a command typed in your terminal. You start with the name of the function and then pass some arguments, for example:

```
PYTHON
In [1]: %timeit range(1000)
255 ns ± 10.3 ns per loop (mean ± std. dev. of 7 runs, 10
```

My favorite one is the %debug function. Imagine you run some code and it throws an exception. But given you weren't prepared for the exception, you didn't run it through a debugger. Now, to be able to debug it, you would usually have to go back, put some breakpoints and rerun the same code. Fortunately, if you are using IPython there is a better way! You can run `%debug` right after the exception happened and IPython will start an interactive debugger for that exception. It's called *post-mortem debugging* and I absolutely love it!

The second type of magic functions are **cell magics** and they work on a block of code, not on a single line. They are prefixed with `%%`. To close a block of code, when you are inside a cell magic function, hit `Enter` twice. Here is an example of `timeit` function working on a block of code:

```
PYTHON
In [2]: %%timeit elements = range(1000)
   ...: x = min(elements)
   ...: y = max(elements)
   ...:
   ...:
52.8 µs ± 4.37 µs per loop (mean ± std. dev. of 7 runs, 1
```

Both the line magic and the cell magic can be created by simply decorating a Python function. Another way is to write a class that inherits from the `IPython.core.magic.Magics`. I will cover this second method in a different article.

Creating line magic function

That's all the theory. Now, let's write our first magic function. We will start with a `line magic` and in the second part of this tutorial, we will make a `cell magic`.

What kind of magic function are we going to create? Well, let's make something useful. I'm from Poland and in Poland we are use Polish notation for writing down mathematical operations. So instead of writing `2 + 3`, we write `+ 2 3`. And instead of writing `(5 - 6) * 7` we write `* - 5 6 7`[1].

Let's write a simple Polish notation interpreter. It will take an expression in Polish notation as input, and output the correct answer. To keep this example short, I will limit it to only the basic arithmetic operations: `+`, `-`, `*`, and `/`.

Here is the code that interprets the Polish notation:

```python
def interpret(tokens):
    token = tokens.popleft()
    if token == "+":
        return interpret(tokens) + interpret(tokens)
    elif token == "-":
        return interpret(tokens) - interpret(tokens)
    elif token == "*":
        return interpret(tokens) * interpret(tokens)
    elif token == "/":
        return interpret(tokens) / interpret(tokens)
    else:
        return int(token)
```

Next, we will create a `%pn` magic function that will use the above code to interpret Polish notation.

```python
from collections import deque

from IPython.core.magic import register_line_magic


@register_line_magic
def pn(line):
    """Polish Notation interpreter

    Usage:
    >>> %pn + 2 2
    4
    """
    return interpret(deque(line.split()))
```

And that's it. The `@register_line_magic` decorator turns our `pn` function into a `%pn` magic function. The `line` parameter contains whatever is passed to the magic function. If we call it in the following way: `%pn + 2 2`, `line` will contain `+ 2 2`.

To make sure that IPython loads our magic function on startup, copy all the code that we just wrote (you can find the whole file on GitHub) to a file inside IPython startup directory. You can read more about this directory in the IPython startup files post. In my case, I'm saving it in a file called:

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

(name of the file doesn't matter, but the directory where you put it is important).

Ok, it's time to test it. Start IPython and let's do some *Polish* math:

```
In [1]: %pn + 2 2
Out[1]: 4

In [2]: %pn * - 5 6 7
Out[2]: -7

In [3]: %pn * + 5 6 + 7 8
Out[3]: 165
```

Perfect, it works! Of course, it's quite rudimentary - it only supports 4 operators, it doesn't handle exceptions very well, and given that it's using recursion, it might fail for very long expressions. Also, the `queue` module and the `interpret` function will now be available in your IPython sessions, since whatever code you put in the `magic_function.py` file will be run on IPython startup.
But, you just wrote your first magic function! And it wasn't so difficult!

At this point, you are probably wondering - *Why didn't we just write a standard Python function?* That's a good question - in this case, we could simply run the following code:

```
In [1]: pn('+ 2 2')
Out[1]: 4
```

or even:

```
In [1]: interpret(deque('+ 2 2'.split()))
Out[1]: 4
```

As I said in the beginning, magic functions are usually helper functions. Their main advantage is that when someone sees functions with the `%` prefix, it's clear that it's a magic function from IPython, not a function defined somewhere in the code or a built-in. Also, there is no risk that their names collide with functions from Python modules.

Conclusion

I hope you enjoyed this short tutorial and if you have questions or if you have a cool magic function that you would like to share - drop me an email or ping me on Twitter!

Stay tuned for the next parts. We still need to cover the **cell magic** functions, **line AND cell magic** functions and **Magic** classes.

Image from: Pexels

---

1. It's a joke. We don't use *Polish notation* in Poland 😉 ↵

Similar posts

Creating Magic Functions in IPython - Part 2

Continue the magic functions journey and create a cell magic function that checks type hints in IPython.

08 Feb 2019

5 Ways of Debugging with IPython

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

23 Dec 2019

IPython Extensions Guide

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

15 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:
__str__ vs. __repr__

Next:
Creating Magic Functions in IPython - Part 2

0 comments

Write Preview

Sign in to comment