

01.03-Magic-Commands

November 10, 2024

1 IPython Magic Commands

The previous chapter showed how IPython lets you use and explore Python efficiently and interactively. Here we'll begin discussing some of the enhancements that IPython adds on top of the normal Python syntax. These are known in IPython as *magic commands*, and are prefixed by the % character. These magic commands are designed to succinctly solve various common problems in standard data analysis. Magic commands come in two flavors: *line magics*, which are denoted by a single % prefix and operate on a single line of input, and *cell magics*, which are denoted by a double %% prefix and operate on multiple lines of input. I'll demonstrate and discuss a few brief examples here, and come back to a more focused discussion of several useful magic commands later.

1.1 Running External Code: %run

As you begin developing more extensive code, you will likely find yourself working in IPython for interactive exploration, as well as a text editor to store code that you want to reuse. Rather than running this code in a new window, it can be convenient to run it within your IPython session. This can be done with the %run magic command.

For example, imagine you've created a *myscript.py* file with the following contents:

```
# file: myscript.py

def square(x):
    """square a number"""
    return x ** 2

for N in range(1, 4):
    print(f"{N} squared is {square(N)}")
```

You can execute this from your IPython session as follows:

```
In [6]: %run myscript.py
1 squared is 1
2 squared is 4
3 squared is 9
```

Note also that after you've run this script, any functions defined within it are available for use in your IPython session:

```
In [7]: square(5)
Out[7]: 25
```

There are several options to fine-tune how your code is run; you can see the documentation in the normal way, by typing `%run?` in the IPython interpreter.

1.2 Timing Code Execution: `%timeit`

Another example of a useful magic function is `%timeit`, which will automatically determine the execution time of the single-line Python statement that follows it. For example, we may want to check the performance of a list comprehension:

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]
430 µs ± 3.21 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The benefit of `%timeit` is that for short commands it will automatically perform multiple runs in order to attain more robust results. For multiline statements, adding a second `%` sign will turn this into a cell magic that can handle multiple lines of input. For example, here's the equivalent construction with a `for` loop:

```
In [9]: %%timeit
...: L = []
...: for n in range(1000):
...:     L.append(n ** 2)
...:
484 µs ± 5.67 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

We can immediately see that list comprehensions are about 10% faster than the equivalent `for` loop construction in this case. We'll explore `%timeit` and other approaches to timing and profiling code in [Profiling and Timing Code](#).

1.3 Help on Magic Functions: `?`, `%magic`, and `%lsmagic`

Like normal Python functions, IPython magic functions have docstrings, and this useful documentation can be accessed in the standard manner. So, for example, to read the documentation of the `%timeit` magic function, simply type this:

```
In [10]: %timeit?
```

Documentation for other functions can be accessed similarly. To access a general description of available magic functions, including some examples, you can type this:

```
In [11]: %magic
```

For a quick and simple list of all available magic functions, type this:

```
In [12]: %lsmagic
```

Finally, I'll mention that it is quite straightforward to define your own magic functions if you wish. I won't discuss it here, but if you are interested, see the references listed in [More IPython Resources](#).