**Python Magic Methods You Might Not Have Heard About**

**There are many less-known Python magic methods—let’s find out what they do and how we can use them in our code**



Photo by [Aaron Huber](https://unsplash.com/@aahubs?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

Python’s magic methods — also known as *dunder* (double underscore) methods — can be used to implement a lot of cool things. Most of the time we use them for simple stuff, such as constructors…

**Iterator Length**

We all know the \_\_len\_\_ method that you can use to implement len() function on your container classes. What if you want to get length of a class object that implements an iterator, though?

<https://gist.github.com/MartinHeinz/61ab69c21290107433f7ef5ff62fc6c6#file-length_hint-py>



All you need to do is implement the \_\_length\_hint\_\_ method, which is also present on builtin iterators (but not generators) as you can see above. Additionally, as you can see here, it also supports dynamic length changes. With that said though—as the name suggests—it's really just a *hint* and can be wholly inaccurate: for the list iterator you will get exact results, for other iterators not necessarily. However, even if it's not accurate, it can be very helpful for optimizations as explained in the [PEP 424](https://peps.python.org/pep-0424/) which introduced it a while back.

**Meta Programming**

The bulk of the magic methods you rarely see being used are related to meta-programming and while meta-programming is something you probably don’t have to use every day, there are some handy tricks you can use it for.

One such trick is using \_\_init\_subclass\_\_ as a shortcut to extend the functionalities of a base class without having to handle metaclasses:

Here we use it to add keyword argument to a base class, which can be set when defining the child. In the real world use cases, you would probably use this in a situation where you want to process the provided argument, rather than just assigning to an attribute.

While this might seem very obscure and rarely useful, you probably encountered it many times already, as it can be used when building an API where users subclass your parent class like in [SQLAlchemy models](https://docs.sqlalchemy.org/en/14/orm/inheritance.html) or [Flask Views](https://github.com/pallets/flask/blob/9b44bf2818d8e3cde422ad7f43fb33dfc6737289/src/flask/views.py#L162).

Another metaclass magic method that you might find use for is \_\_call\_\_. This method allows you to customize what happens when you *call* a class instance:

Funnily, you can use this to create class that cannot be called:

This can be useful if you have a class that only have static methods and therefore there’s no good reason to create instances of said class.

Another similar use case that comes to mind is singleton pattern — a class that can have at most single instance:

Here we demonstrate this by implementing a global logger class that there can be only one instance of. The concept might look a little complicated, but this implementation is pretty simple — the Singleton class holds a private \_\_instance —if there's none, it gets created and assigned to the attribute, if it already exists, it just gets returned.

Now, let’s say you have a class and you want to create an instance of it without invoking \_\_init\_\_. The \_\_new\_\_ magic method can help with that:

There are situations where you might need to bypass the usual process of creating instance and the above code shows how you can do that. Instead of calling Document(...) we invoke Document.\_\_new\_\_(Document) which creates a bare instance without invoking \_\_init\_\_. Because of that, the instance attribute(s) - in this case text - aren't initialized, to fix that, we can use setattr function (which - by the way - is also a magic method - \_\_setattr\_\_).

You might be wondering why would you ever want to do that. One example would be implementing alternative constructor, such as this:

Here we define from\_file method which serves as a constructor by first creating instance with \_\_new\_\_ and then configuring it without invoking \_\_init\_\_.

Next meta-programming related magic method we will take a look at here is \_\_getattr\_\_. This method gets called when normal attribute access fails. This can be leveraged to delegate access/calls to missing methods to another class:

Let’s suppose that we want to define custom implementation of string with some extra functions such as custom\_operation above. We however, don't want to re-implement every single string method such split, join, capitalize, and so on. Therefore, we use \_\_getattr\_\_ to call these existing string methods in case they're not found on our class.

While this works great for normal methods, notice that in the example above the operations such as concatenation provided by magic method \_\_add\_\_ doesn't get delegated. So, if we wanted those to work as well, then we would have to re-implement them.

**Introspection**

The final meta-programming related magic method we will try out is \_\_getattribute\_\_. This one looks very similar to the previous \_\_getattr\_\_. There's however a slight difference — as already mentioned \_\_getattr\_\_ gets invoked only when attribute lookup fails, while \_\_getattribute\_\_ is invoked *before* attribute lookup is attempted.

You can therefore use \_\_getattribute\_\_ to control access to attributes, or you can for example create a decorator that logs every attempt to access instance attribute:

The logger decorator function start by taking note of the original \_\_getattribute\_\_ method of the class it decorates. It then replaces it with custom method that first logs the name of the attribute being accessed before calling the original \_\_getattribute\_\_ method.

**Magic Attributes**

So far, we’ve talked only about magic methods, but there are also quite a few magic variables/attributes in Python. One of them is \_\_all\_\_:

This magic attribute can be used to define which variables and function get exported from a module. In the example we create a Python module in .../some\_module/ with single file (\_\_init\_\_.py). In this file we define 2 variables and one function of which we export only 2 ( func and some\_var). If we then try to import contents of some\_module in other Python program we only get the 2 exported ones.

Be aware, though, that the \_\_all\_\_ variable only effects the \* import shown above, you're still able to import the un-exported functions and variables with imports like import some\_other\_var from some\_module.

Another double underscore variable (module attribute) that you might have seen is \_\_file\_\_. This variable simply identifies path to the file it's accessed from:

And combining the \_\_all\_\_ and \_\_file\_\_, you can for example load all modules in a folder:

And one last we will try out is \_\_debug\_\_ attribute. This one - obviously - can be used for debugging, but more specifically it can be used to better control assertions:

If we run this piece of code normally with python example.py, we will see the "debugging logs" printed out, however if we use python3 -O example.py, the optimization flag (-O) will set \_\_debug\_\_ to false and strip out the debugging messages. Therefore, if you run your code with -O in production environment, you won't have to worry about forgotten print calls left from debugging, as they will be all stripped out.

**Hidden and Undocumented**

All of the above methods and attributes might be somewhat unknown, but they’re all in Python docs. There are however a couple that are not clearly documented and/or somewhat hidden.

You can for example run the following code to discover a couple new ones:

Besides these, there are quite a few more as listed in Python bug tracker [BPO 23639](https://github.com/python/cpython/issues/67827). As pointed out there though, most of them are implementation details or private names which should not be accessed. So them being not documented is probably for the best.

**Making Your Own?**

Now, with so many magic methods and attributes, could you actually make your own? Well, you could, but you should not.

The double underscore names are reserved for future extensions of the Python language and should not be used for your own code. If you decide to use such name in your code anyway, then you’re running a risk of them getting added to the Python interpreter in the future, which would more than likely break your code.

**Closing Thoughts**

In this article we looked at the lesser known magic methods and attribute that I find useful or interesting, there are however more of them listed in docs that might be useful for you. Most of them can be found in [Python Data Model docs](https://docs.python.org/3/reference/datamodel.html#special-method-names). If you however want to dig deeper, you can try searching for "\_\_" in Python docs, which will turn up [many more methods and attributes](https://docs.python.org/3/search.html?q=__&check_keywords=yes&area=default) to explore and play with.