Sam Vidovich

127 Followers

3 Cool Python Libraries that will Save You Time and Effort

Sam Vidovich Aug 30 · 7 min read

Hey look, I'm using the Python logo. Again. Really it's just for a good-looking thumbnail.

Hey again, valued reader. How are you? Good! Good to hear. I am genuinely invested in your well being, you know. That's why I'm writing this article! Last time, we talked about a couple of tips to writing performant code in Python. Today we're going to talk about using open-source libraries when you code to save time. This isn't something to do out of laziness; this is something to do because it will save you a lot of time, effort and heart-ache.

When you find yourself getting caught up in something mechanical, take a moment and think whether this problem is already solved in a library. If it is, then it's likely that several *thousand* people are already using it. That means that it's battle-tested and well-known — more-so than you'll likely achieve by yourself.

Not only that, but these libraries are really cool and simple to use. I've saved the best for last, so, you know. Fork over them 'average read times'.

JMESPath

Pronounced "James-Path", this is a library that helps query JSON. It's super simple. You can find it here!

Situation: You're dealing with a really deep JSON document or a dictionary. Sometimes the keys might not be there. Let's suppose that it looks something like this:

```
test_dictionary = {
    'level_1': {
        'level_2_a': {
            'level_3_a': 'some_string'
        },
```

```
        'level_2_b': {
            'level_3_b': 'a_different_string'
        }
    }
}
```

now let's assume that any of the levels below level 1 could just be null. Then, if you wanted to get to the deepest portion safely, you'll have to do something like this:

```
level_3_a = test_dictionary.get('level_1', {}).get('level_2_a',
{}).get('level_3_a', '')
```

And that sucks. It's ugly code: difficult to read, and if something goes wrong with it, figuring out *what* went wrong sucks.

Instead, let's try JMESPath:

```
expression = jmespath.compile('level_1.level_2_a.level_3_a')
result = expression.search(test_dictionary)
```

JMESPath gives us access to the sort of 'JavaScript style' object-access of JSON or dictionary objects that we're craving. It makes the code simpler and more testable. It's also safe — if any of the paths don't exist, JMESPath's search function will return None instead of exploding. Save yourself time and headache, and parse JSON and deep dictionaries with JMESPath.

Inflection

Inflection is a library based on something from Ruby. It helps you process your strings, because string processing logic can be difficult to write, and there can be a lot of corner cases you miss the first time around. You can find it here!

Let's go through some situations, shall we?

Situation: You have sets of words that you want to turn into titles for articles or for books. They come from random sources, and are all ugly and not quite in title form.

```
words = {'the_last_mimzy', 'The-Return-Of-The-King',
'ConanTheBarbarian'}
```

Trying to write custom code that detects defects in these messed up titles would be time consuming and have lots of edge cases, especially as the count of titles or the number of sources you're pulling from expands. Offload it to inflection:

```
words = {'the_last_mimzy', 'The-Return-Of-The-King',
'ConanTheBarbarian'}
titleized_words = [inflection.titleize(word) for word in words]
print(titleized_words)

>>> ['The Last Mimzy', 'The Return Of The King', 'Conan The
Barbarian']
```

Instead of dozens of lines (or even hundreds, depending on how dirty your data is) of code that needs tons of testing, we solve our issue in one line with inflection. Not bad, right?

Situation: You've got names of variables or datapoints coming in that were generated in another language or system that you want to use, but you need them to conform to PEP standards, that is, you need them to be *snake cased*. That is, suppose you have some datapoints with names like this:

```
foreign_names = {'SystemMemoryUsage', 'systemCPU',
'webContainerCGroup'}
```

Like in our 'movie / book title' situation, in order to do this yourself, you'd need to write several parsers for these, and even then, several tests, with plenty of corner cases. Instead, offload it to inflection:

```
foreign_names = {'SystemMemoryUsage', 'systemCPU',
'webContainerCGroup'}

snake_cased_words = [inflection.underscore(name) for name in
foreign_names]

print(snake_cased_words)
>>> ['system_memory_usage', 'system_cpu', 'web_container_c_group']
```

Perfect, just what we needed! We don't have to get in a big string fistfight where we chunk the strings out, lower them, blah blah… just one function call and life is good.

Those are two examples of where inflection can save you time. There are quite a few more uses for this library. It's a small library — you should check out the docs here to get more ideas on how to use it!

more-itertools

I promised the best for last. If saving time on JSON parsing and string transformations doesn't get you excited, then boy have I got something for you. more-itertools is a seriously heavy-hitter, and any time you're doing something where you're iterating a list, set, or other Python iterable, you should stop and see whether your problem has been solved by more-itertools. It's really that powerful. You can find the source here. more-itertools has several categories of functions to help you program, I'll demonstrate a few.

Let's get going with a couple of situations! (I had three but the third one was a little confusing, so I chopped it from the article. The code is still in the gist, though. You might like it.)

Situation: You have a list of dictionaries each with a key in common, and that key is repeated (for example: an id). You want to split it into more than one list based on that common, repeated key. So, you have something like this:

```
sample_chunkable_list_of_dictionaries = [
    {'id': '1', 'datapoint': 'datapoint_1_1'},
    {'id': '1', 'datapoint': 'datapoint_1_2'},
    {'id': '1', 'datapoint': 'datapoint_1_3'},
    {'id': '1', 'datapoint': 'datapoint_1_4'},
    {'id': '2', 'datapoint': 'datapoint_2_1'},
```

```
    {'id': '2', 'datapoint': 'datapoint_2_2'},
    {'id': '2', 'datapoint': 'datapoint_2_3'},
    {'id': '2', 'datapoint': 'datapoint_2_4'},
    {'id': '3', 'datapoint': 'datapoint_3_1'},
    {'id': '3', 'datapoint': 'datapoint_3_2'},
    {'id': '3', 'datapoint': 'datapoint_3_3'},
    {'id': '3', 'datapoint': 'datapoint_3_4'},
]
```

See how the 'id' key is common, and can get repeated? let's split that out into a list of lists, each list containing only the entries from the original that has the same id. All we need to do is use the split_when function from itertools, and provide a comparison function:

```
split_into_chunks = list(
    more_itertools.split_when(
        sample_chunkable_list_of_dictionaries, lambda x, y: x['id']
!= y['id']
        )
    )
```

The formatting here would get really ugly, so I'll show you an image of the result:

Using more-itertools' split_when function

Just what we were looking for — nicely partitioned. I tried to write this routine myself, and it got really ugly, really fast. Having to keep track of positions in your list is a pain, and you're inviting stange off-by-one errors and corner cases. Do yourself a favor and just use split_when, and the similar functions available in the library.

This is one of more-itertools' data **grouping** functions, and it has a ton more for when your data is a little less neat-and-tidy, for when you want to partition, etc. Check those out.

Situation: you have several lists that you want to bring together, but you need the combined list to keep their 'priority' together — you need to round-robin combine them. You could easily write this code, it would probably look something like this:

Writing my own interleave

Hey, it's recursive! And it works! If I use it with some cool values, we get what we expect out of it:

```
need_interleaving = [
    [1, 3, 5],
    [2, 4, 6],
    [10, 11, 12, 13]
]

print(hand_rolled_interleave(need_interleaving))

>>> [1, 2, 10, 3, 4, 11, 5, 6, 12, 13]
```

There are a couple of problems. First of all, it's recursive, which is immediately confusing to try and read. Second of all, it's rather complex, with changing data structures and comprehensions. Third, it's slow on the grand scale. I can use a method called interleave_longest (to ensure that we don't stop once we've exhausted one of our iterables) in a fabulous one-liner instead:

```
need_interleaving = [
    [1, 3, 5],
    [2, 4, 6],
    [10, 11, 12, 13]
]

# The * is a spread operator — it lets me pass the lists in all at
once
print(list(more_itertools.interleave_longest(*need_interleaving))

>>> [1, 2, 10, 3, 4, 11, 5, 6, 12, 13]
```

Cool! So the code isn't recursive, and it's not complex or difficult to read. But what about the performance? let's give it a whirl.

```
(venv) mydriasis@akkad:~/Desktop/article-code/oss-libs$ python more-
itertools-test.py
Interleave with 10 lists of length 100 each took 0.0000069141ms with
more_itertools
Interleave with 10 lists of length 100 each took 0.0003936291ms with
hand-rolled interleave
```

Behold, my goofy hand-rolled method takes *one hundred times longer* than itertools'. Also, because it's recursive, it doesn't scale. If the lists are longer than Python's maximum recursion depth, this just explodes with a RecursionError.

See? It's worth it to use the library. It means we don't have to come up with a regular implementation ourselves, or adjust recursion limiting. This is one of more-itertools' **combining** functions, and there are plenty more where that came from.

At the end of the day, I'm really wow-ed by more_itertools. It has lots of functions that I can see myself using all the time, and they're likely more efficient than anything I could write. Let's be real, I'm not a better programmer than 71 dedicated Python engineers.

Conclusion

Well, here we are again, you and I. At the end of another article that you definitely read this much of. Maybe you enjoyed this overview. Maybe you learned something. Maybe you'll wind up using JMESPath, because parsing JSON sucks. Or maybe you bailed out 30 seconds in, because you saw the top of a code block and it turned you off. No matter what, I hope you enjoyed. As always, you can find the sample code for this article in a fabulous gist. I've been… ehm, programming man. We'll see you next time on *Articles You Didn't Want to Read but Did Anyways Because of the Catchy Title!*

Python Programming Open Source