Rewriting Reddit

December 6, 2005

Original link

Translations: Srpsko-Hrvatski

2012 note: This article was first published in 2005. After it was published, Django launched a RemovingTheMagic project to address some of my criticisms (though personally I still find it unusable), web.py inspired FriendFeed's tornado.web and Google's gae.webapp and others (though I still prefer web.py), and this article led to a permanent surge in Reddit traffic that still hasn't really stopped growing.

Over at reddit.com, we rewrote the site from Lisp to Python in the past week. It was pretty much done in one weekend. (Disclosure: We used my web.py library.) The others knew Lisp (they wrote their whole site in it) and they knew Python (they rewrote their whole site in it) and yet they decided liked Python better for this project. The Python version had less code that ran faster and was far easier to read and maintain.

The idea that there is something better than Lisp is apparently inconceivable to some, judging from comments on the reddit blog. The Lispers instead quickly set about trying to find the *real* reason behind the switch.

One assumed it must have been divine intervention, since "there seems to be no other reason for switching to an inferior language." Another figured something else must be going on: "Could this be…a lie? To throw off competition? It's not as though Paul Graham hasn't hinted at this tactic in his essays…" Another chimed in: "I decided it was a prank." Another suggested the authors simply wanted more "cut corners, hacks, and faked artisanship."

These were, of course, extreme cases. Others assumed there must have been outside pressure. "Either libraries or hiring new programmers I guess." Another concluded: "some vc suit wants a maintainable-by-joe-programmer product. I hope he pays you millions."

The Lisp newsgroup, `comp.lang.lisp`, was upset about the switch that they're currently planning to *write a competitor* to reddit in Lisp, to show how right they are or something.

The more sane argued along the lines of saying Lisp's value lies in being able to create new linguistic constructs and that for something like a simple web app,

this isn't necessary, since the constructs have been already built. But even this isn't true. web.py was built pretty much from scratch and uses all sorts of "new linguistic constructs" and — even better — these constructs have syntax that goes along with them and makes them reasonably readable. Sure, Python isn't Perl 6, so you can't add arbitrary syntax, but you can often find a clever way to get the job done.

Python, on the other hand, has problems of its own. The biggest is that it has dozens of web application frameworks, but none of them are any good. Pythonists are well aware of the first part but apparently not of the second, since when I tell them that I'm using my own library, the universal response is "I don't think Python needs another web application framework". Yes, Python needs fewer web application frameworks. But it also needs one that doesn't suck.

The framework that seems most promising is Django and indeed we initially attempted to rewrite Reddit in it. As the most experienced Python programmer, I tried my best to help the others out.

Django seemed great from the outside: a nice-looking website, intelligent and talented developers, and a seeming surplus of nice features. The developers and community are extremely helpful and responsive to patches and suggestions. And all the right goals are espoused in their philosophy documents and FAQs. Unfortunately, however, they seem completely incapable of living up to them.

While Django claims that it's "loosely coupled", using it pretty much requires fitting your code into Django's worldview. Django insists on executing your code itself, either through its command-line utility or a specialized server handler called with the appropriate environment variables and Python path. When you start a project, by default Django creates folders nested four levels deep for your code and while you can move around some files, I had trouble figuring out which ones and how.

Django's philosophy says "Explicit is better than implicit", but Django has all sorts of magic. Database models you create in one file magically appear someplace else deep inside the Django module with a different name. When your model function is called, new things have been added to its variable-space and old ones removed. (I'm told they're currently working on fixing both of these, though.)

Another Django goal is "less code", at least for you. But Django is simply full of code. Inside the django module are 10 different folders and inside each of those are a few more. By the time you actually build a site in the Django tutorial, you've imported `django.core.meta`, `django.models.polls`, `django.conf.urls.defaults.*`, `django.utils.httpwrappers.HttpResponse`,

and `django.core.extensions.render_to_response`. It's not clear how anyone is supposed to remember all that, especially since there appear to be no guiding principles for what goes where or how it's named. Three of these are inserted automatically by the start scripts, but you still need to memorize such names for every other function you want to use.

But Django's most important problem is that its developers seem incapable of designing a decent API. They're clearly capable Python programmers — their code uses all sorts of bizarre tricks. And they're clearly able to write code that works — they have all sorts of interesting features. But they can't seem to shape this code into something that other people can use.

Their APIs are ugly and regularly missing key features: the database API figures out queries by counting underscores but has no special syntax for JOINs, the template system requires four curly braces around every variable and can't do any sort of computation, the form API requires 15 lines to process a form and can't automatically generate the template.

I tried my best to fix things — and the Django community was extremely supportive — but the task simply dwarfed me. I just couldn't do it mentally, let alone with the time constraints of having to actually build my own application for my own startup.

And so, Lisp and Django found wanting, we're left with web.py. I'd like to say that web.py learned from these mistakes and was designed to avoid them, but the truth is that web.py was written long before all this and managed to avoid them anyway.

The way I wrote web.py was simple: I imagined how things should work and then I made that happen. Sometimes making things just work takes a lot of code. Sometimes it only takes a little. But either way, that fact is hidden from the user — they just get the ideal API.

So how should things work? The first principle is that code should be clear and simple. If you want to output some text, you call web.output. If you want to get form input, you call web.input. There's nothing particularly hard to remember.

The second principle is that web.py should fit your code, not the other way around. Every function in web.py is completely independent, you can use whichever ones you want. You can put your files wherever you like, and web.py will happily follow along. If you want a piece of code to be run as a web app, you call web.run, you don't put your code in the magical place so that web.py can run you.

The third principle is that web.py should, by default, do the right thing by the Web. This means distinguishing between GET and POST properly. It means

simple, canonical URLs which synonyms redirect to. It means readable HTML with the proper HTTP headers.

And that, as far as I'm concerned, are pretty much all the principles you need. They seem pretty simple and obvious to me and I'm even willing to fudge on some of them, but no other Python web app framework seems to even come close. (If you know of one, tell me and I'll happily recant. I don't want to be in this business.) Until then, it looks like I'm forced to do that horrible thing I'd rather not do: release one more Python web application framework into the world.