django

Building Django

Jacob Kaplan-Moss

Colorado Software Summit
October 19-24th, 2008

Welcome!

My name is Jacob Kaplan-Moss; I’m one of the lead developers of Django and the former lead developer at the Lawrence Journal-World, where Django originated. This is a bit of a new topic for me: instead of talking about how Django works, I’ll be looking at why Django’s built the way it is. I'll look at the history of Django, the philosophies guiding its development, and dissect the choices we made as we developed Django.

First, a little bit about me.

At my day job, I’m a software architect at Whiskey Media, makers of fine websites. That’s a new gig — until recently I was the lead developer at the Lawrence Journal-World, where Django was originally developed.

Today, though, I’m firmly wearing my Django hat. I’ve been working on Django since before it was released to the public — I helped release it — and today I’m one of the two lead developers, or BDFLs, or the project. I’ll talk a bit more about what a BDFL does later on in the talk.
Django’s had an interesting history which has informed many future decisions.

As a side-note, this picture here is of the Lawrence Journal-World newsroom on a local election night circa 1950. If you examine the picture closely, you can actually see an early form of database-driven journalism in the back of the room.

Unlike much of the software we use today, Django didn’t come out of Silicon Valley, or in fact any of the usual suspect breeding grounds.

No, Django was originally developed in Kansas...

... which actually isn’t nearly as bad as you think.
No, Django comes out of Lawrence, KS, a fantastic little town smack in the middle of flyover country. It’s got a vibrant downtown...

... a couple of awesome locally-owned bookstores (and associated shop cats) ...

... a really fantastic local music scene ... 

... one of the best microbreweries in the country ...
... and is home to KU, 2008’s NCAA basketball champions. We’re #1, baby!

I didn’t know any of that when I moved there, however — I went to Lawrence to work here, at the local newspaper.

And specifically I went to work with these guys, Adrian & Simon.

You might know Adrian from his work on ChicagoCrime and/or EveryBlock; he’s probably the most highly renowned of a new breed of “database journalists.” Simon “Stinky” Willison is another one of those guys who’s somehow always ahead of the curve. Just a few examples: he started writing about AJAX years before the acronym was coined; his `getElementsByTagName` library was the inspiration for jQuery, a leading JavaScript framework; and Simon is one of the premier evangelists responsible for the widespread adoption of OpenID.

I moved halfway across the country to work with these guys, and it was the best decision I ever made.
The flagship project of the online group in those early days was this site, lawrence.com. It’s a site dedicated to the local entertainment scene in Lawrence, and it kicks the crap out of anything else I’ve ever seen.

We’ve got all the cool buzzwords:

... MP3 downloads of local music ...

... RSS feeds for every band, venue, blog, etc ...
... podcasts ...

... plus a weekly print edition that’s actually generated from the website content instead of the other way around, which is much more common in traditional print media circles. Lawrence.com may have in fact invented the concept; I’m certainly not aware of anything similar that predates it.

Now, in the beginning this was all written in PHP. This did not work out very well.
So Adrian and Simon switched to Python. I’ll talk more about that decision later on.

When I joined in the summer of 2004, I got a look at what Adrian and Simon had been developing for the last year or so: a content management system for media companies. It doesn’t sound all that sexy, but it really was — it was light, fast, and clean, putting a premium on clear, structured data and easy-to-use interfaces.

At the time, we just called it “The CMS.” We didn’t really think we were building a “framework” — we were just focused on building a CMS for our properties.

Still, we did have some guiding philosophies that guided us in those early days.
First, we needed to make web development fast. We were a department of three supporting an entire newsroom and a number of websites, some quite large. On top of that, when you work for a news company, you have to work on a journalist’s schedule: when a story breaks, you don’t get much time to develop software.

We discovered very quickly that web development is often boring and tedious, so we wanted tools that would automate the repetitive tasks easily.

We were also pretty obsessed with best practices — web standards, accessibility, etc. — and wanted a set of tools that would help us do the right thing with a minimum of effort.
Finally there’s this. Let me explain:

When you print a newspaper, as soon as it rolls off the press it’s done. You don’t get to revise, change, or expand it — the day’s paper is what it is. Online, however, “the ink is never dry” — you can always tweak, improve, change, or expand features. This can lead to some big problems as a software developer: if every thing you develop is a feature that’s used continuously (as opposed to a one-off), you have a potentially deadly increase in incremental maintenance. This means that ongoing, improving tools have to be virtually maintenance-free.

So, flash forward to March 2005, when there was a confluence of a couple of formative events.

The first was that the annual Python Conference (held in Washington, DC in 2005). That year they held a “Python success stories” session where folks were invited to give five-minute demos of what Python had done for them. Adrian demoed our CMS, building a little blog tool in a couple of minutes. It was a well-timed demo: a major theme of that conference was the rather sorry state of web development in Python, so Adrian’s talk was a huge hit — we had all sorts of folks coming up after and asking how they could get their hands on the software.
Around the same time, this happened: Ruby on Rails was coming to prominence, and was seriously shaking up web development (for the better!) I think it’s important not to understate Rails’ effects on web development in general: it really has had a major, positive effect on our entire industry.

However, at the time, all we could think was, “what have they got that we don’t?”

So we started fooling around with the idea of releasing Django as Open Source. Here’s an early (joke) version of our website.

July 2005 was our IPO, and we started getting users, developers, and contributors right away. I won’t talk much about the development of Django between then and now, but suffice to say that over these last three years Django’s grown by leaps and bounds.

We’ve just released Django 1.0, which represents a stake in the ground, a stable version with internals and techniques we’re really happy with. Future releases of Django will be rigorously backwards-compatible, and that’s a big deal.
OK, on to the meat of this presentation: the choices we were faced with along the way, and our reasons for making them as we did.

The first big decision was simply the choice to release Django as Open Source, instead of trying to sell it as commercial software.

Now, Adrian and I are both pretty devoted Open Source fans, but the choice wasn’t up to us — it was up to the management at the Lawrence Journal-World. So we prepared a series of arguments in favor of Open Source.

First, Open Source code is higher quality. This is mostly because you’ve got all these people looking at your code and using it in ways you couldn’t anticipate; they find lots and lots of bugs.
That’s Linus Torvalds, creator of Linux, as formulated by Eric S. Raymond in *The Cathedral and the Bazaar*.

There’s also a related effect of Open Source: it’s major impetus to get your code right, because you don’t want to be shamed in front of your peers.

We argued that releasing Django would help us hire: we’d have a pool of developers we could hire out of with great confidence, since they’d be part of the existing community.

Releasing Open Source code would help put us “on the map”, and that would help again with hiring, and also with our fledgling commercial software business.
Finally, we argued that our entire business was based around Open Source — Linux, PostgreSQL, Python, ... — and that this was our opportunity to give back to the community that had given us so much.

Surprisingly, this argument was the one that seemed to carry the most weight with our management. A nice testament to how clueful this small company can be.

How’d it work out?

Turns out — to my great surprise, I promise! — that all of these arguments came true.
This highly scientific graph shows clearly that our code quality has increased dramatically since releasing Django.

Seriously, though — by any conceivable metric, Django’s much, much better for being Open Source. We’ve got a high level of test coverage, when before releasing Django we didn’t even have a test system, let alone any tests. We’ve got some major performance improvements given by the community, as well as all sorts of great features.

Hiring has also been made much, much easier. This is a shot of some of the folks working over at the J-W these days; the awesome thing is that each of them came to the J-W already knowing Django, which means the “start up cost” (training, etc.) of new employees is dramatically lower.

There’s no question that Django has helped keep the Journal-World on the map. It’s lead directly to sales of Ellington, the Journal-World’s award-winning CMS for news/media companies.
We’ve also been able to give something really big back to the Python community. Recall that when we released Django, the Python web development state of the art was pretty woeful. These days, Django is hot, and it’s bringing lots of people into the greater Python community.

We’ve also been held up as an example of how successful Open Source projects ought to be run, which is fantastic. This is Guido van Rossum, creator and BDFL of Python, and he says the following of Django:

> Django is [...] an exemplary example of a good open source project, run by people who really understand community involvement.

— Guido van Rossum
Creator of Python


The next choice we made right at the beginning was the choice of the BSD license for our code. Depending on where you fall within the FLOSS community, this might come as a surprising choice, so I’ll look at it a bit.
This next bit is a little bit “inside baseball” — I’m going to geek out on software licensing for a bit. If you don’t care about this subject, you might want to use this time to check your email or tweet about how boring I am.

There’s a lot of misunderstanding about what software licensing is, and how it fits together with copyright. Here’s a crash course.

First term: “Copyright.”

Any creative material you create (in the United States, at least) automatically gets copyright protection. This means that since you own the material, you also own the “rights” to make “copies,” and can determine who is and is not permitted to make copies, and under what circumstances they may make derivative works (modifications). The law provides for very substantial penalties for illegal copies — as much as $150,000 per illegal copy!

The courts classify software as a creative material, and includes making digital copies in copyright protection. This means that if you write a bit of code and I make an edited version without your explicit explicit permission, I’ve broken the law. Notice again that this happens automatically — you don’t need to register your code, or even mark it as copyright. “All rights reserved” is the default.
Second term: “copyleft”

This is a play on the word “copyright” and basically describes the idea of using copyright law to actually *remove* restrictions on copying and modifying content. It does this through *licensing* — granting permission to other parties to make copies and derivative works under certain explicit circumstances.

There’s a few important things to note about copyleft:

» It’s not anti-copyright. In fact, copyleft licenses get all their power from Copyright: violating a copyleft license is the same as violating an “All rights reserved” work, and is subject to the same penalties. In other words, copyleft gets its teeth from copyright.

» It’s not “giving away” code. The creator maintains ownership of the original work — as well as derivative versions — but she simply grants free usage under certain conditions. It’s not clear, in fact, if giving away copyright work is even possible in the United States; there doesn’t seem to be any legal way to actually renounce ownership. The best you can do is an incredibly permissive license.

» Legally, there’s no difference between a copyleft license and a traditional software license. That is, when you buy Photoshop, what you actually are buying from Adobe is a *license* to use Photoshop under certain circumstances and with certain restrictions. Although there are many differences in content between that license and a copyleft one, the law recognizes them equally.

» There’s no formal definition of “copyleft” or “open source” licenses. The closest we get — and the definition I use — is OSI’s *Open Source Definition* and their list of approved *Open Source licenses*. 
Given this understanding, let’s talk a bit about some different licenses. There’s a lot of different ways to think about software licenses; one way is to classify licenses along “proprietary ... public” and “unlimited ... limited reuse” axes. This does a good job of pointing out some differences between some popular licenses and concepts.

» Down in the bottom left we’ve got traditional closed source software. Completely proprietary (only available to a select few), and highly limited in terms of reuse. There’s variation among closed source, of course, with internal, company only code being the most closed and limited, and commercial software made available as source more public and more reusable.

» Microsoft has tried a sort of middle ground called “shared” source with some of the OS code: the code is still owned by Microsoft, and reuse is still very, very limited, but developers can look at and in some cases even download and modify the source. It’s not “open”, but it is “open-er”.

» All the way on the other end of the spectrum we’ve got so-called “public domain” code. Again, it’s not clear that US copyright law actually recognizes this category, but if it did it’d be all the way up here in the top-right: the reuse is completely unlimited, and absolutely anyone can use the code.

» Now, all the way over here we’ve got the most popular Open Source licenses: the GNU General Public License, or GPL. The GPL is fully public — anyone can use GPL code, for any reason — but it places a few very important restrictions on reuse, which I’ll talk about in a bit. So we could think of the GPL license as a “strong” copyleft license: it’s open, but it’s also restricted.

» Finally, we’ve got Django’s choice, the Berkeley Systems Distribution license, or just the BSD. The BSD is as public as the
GPL, but places almost no restrictions on reuse. More on that later, too.

So let’s talk about the GPL a bit. When most people think of “open source” or “free software”, the GPL is the license that comes to mind.

It’s actually a very long license, with a lot of complexity in it. This isn’t to say that the complexity isn’t needed, but it does visually demonstrate that the GPL has a number of restrictions to reuse.

The most important restriction, as articulated by the Free Software Foundation (the group that owns and puts out the GPL) is this one: the GPL requires that all released derivations be released under the GPL as well.

Source: [http://www.gnu.org/licenses/gpl-faq.html#WhyUseGPL](http://www.gnu.org/licenses/gpl-faq.html#WhyUseGPL)
In other words, the GPL is a “share alike” license. It says “you can use this code as long as you share your version under the same terms as mine.”

You can see how this is a desirable thing, I hope, and in fact the GPL is probably primarily responsible for foiling efforts to co-opt or even kill Linux. It’s a hugely important license, and the world is better for it.

However, the GPL is sometimes more negatively referred to as “viral”, and this is a good way of describing it: anything that the GPL touches must also be released as GPL’ed code, and that’s a pretty big limitation on reuse. It makes it difficult to use in some circumstances.

So, what’s the BSD, then?

Here’s the whole BSD license. Yes, that’s really all there is — it’s a very simple license.
This is really all the BSD license says: you can’t take my code and claim it’s yours. There’s nothing that requires derivative works to be released under the same license, or under any Open Source license, nor is there any protection against the sorts of “stealing” that happens with Linux and other large works.

Given that, why’d we choose the BSD?

“Freedom 0” is the freedom to use software for any purpose whatsoever.

I’ve always been confused on this point: the FSF itself coined this term, but their flagship license, the GPL, prohibits certain forms of reuse. That’s always rubbed me the wrong way. The BSD, on the other hand, is about as close to Freedom 0 as you can get.

We felt strongly on this point: we really wanted Django to be used as widely as possible...

... and “anyone” means “anyone.” This goes along the same lines as Voltaire’s famous line: “I do not agree with a word you say, but I will fight to the death for your right to say it.” We felt the same should apply to software: I may not like what you do with my software, but you should have the right to do it.
Another important thing to keep in mind is that community standards often trump the law: when large companies use Django without giving back, there’s quite a bit of peer pressure for them to do the right thing, and that peer pressure comes at a very low price compared to lawyers’ fees.

Finally, and most importantly, there’s this astute observation by Tim O’Reilly. Anything — anything at all — that limits your adoption raises the chance of dooming you to obscurity. We wanted Django to be successful, which means doing everything to help Django actually get used.


So, how’d it work out?

As we expected, Django’s been adopted widely.
Even by some that for silly reasons refuse to touch GPL’d code — there’s a lot of FUD out there about the GPL, and we’ve managed to avoid that argument (phew!).

Here’s a good example: GAE is a major derivative of Django, and includes quite a bit of modified Django code.

Would Google have used Django if the GPL prevented them from doing everything they wanted? I don’t know, but I suspect that’d have avoided a GPL’d Django.

There’s been a huge amount of ink spilled on software design and especially programming methodology. There are some really good ideas out there in this body of literature, but Django doesn’t subscribe to any of ‘em.
We’re all familiar with Design Patterns, but unless you look carefully you’d be hard-pressed to find any GoF patterns in Django.

MVC is a great idea, but if you’re a hardcore MVC fan, Django’s (mis)use of the MVC pattern will likely anger you greatly. We’ve got models and views and something that sorta looks like a controller... and templates, and URLconf's, and a bunch of other stuff, too.

Likewise, testing is probably the single best thing you can do for your code, and so we have lots and lots of tests. But to we practice Test Driven Development? No.

In fact, we don’t really practice any sort of Methodology With Capital Letters.

Well, why the hell not? These things all have great ideas; why not follow through?

First, much of software capital-M Methodology is simple snake oil. Anyone who claims that their system will “fix” your software development woes is trying to sell you something -- sometimes literally; anyone familiar with the Rational Unified Process?
In fact, there really is “no silver bullet” — no two software problems are alike; no team, company, or person operates the same. Why should we expect that any given methodology should work for any arbitrary situation?

This quote, by the way, comes from Fred Brooks’ wonderful Mythical Man-Month, which is probably still the single best analysis of software development ever written. If you haven’t read it, buy it right now; I’ll wait.

Frankly, most of us are just trying to get a job done. We’ll do just about anything if we think it’ll help. When you lock yourself into a particular process, you’re selling yourself short.

That’s is my former boss, Rob Curley, and this is our unofficial motto. Really, though, this is what drives most of us.

Methodology is most decidedly not cool.

So, how’d that work out for us?
The most value we got out of this skepticism of Process was that we brought this same skepticism to our entire development process. Spend much time on our developer mailing list and you’ll see “what’s the need” ask all the time; it’s probably the #1 phrase in all my email. Every single good idea is immediately illuminated with the cold light of pragmatism. As a result, Django is deeply rooted in the Real World; it doesn’t have a single line of code that didn’t get there because a real developer needed it there to get his or her job done.

This leads to a certain immunity to architecture astronauts.

It does, however, create a risk of reinventing the wheel; we have to be careful to note when something is, in fact, a good idea (so that we can steal it).
Next up: Python!

Python is a high-level, dynamic language. It’s been pretty well proven that higher-level languages are more productive, so that’s a big bonus, but productivity actually isn’t one of main reasons we like Python.

This is one of these holy wars that geeks like to fight, but I won’t bother. Suffice to say that we immediately fell in love with the way the language is designed. It’s made to be clean, elegant, simple, and powerful. Python’s often been described as “executable pseudocode.”

This is the so-called “Zen of Python”, and it encapsulates the philosophy behind Python. We found this very refreshing and useful, and Django follows these tenants as well.
Most importantly, Python is a language designed for ease of maintenance. I’ve heard that maintenance coding accounts for as much as 80% of the work on any deployed system — that number certainly matches my personal experience. In that context, a language that’s easy to maintain is vitally important.

Really, though, the best thing about Python is the amazingly smart community.
There’s an amazing collection of third-party packages available for Python that do some really fantastic stuff; I can’t imagine web development without them:

» **BeautifulSoup**: parses even the nastiest HTML into a clean tree structure. Vital for web scraping.

» **dateutil**: the best date/time handling library in any language I’ve used. Especially useful for the component that parses “human” date/time strings like “next thursday afternoon” into date/time objects.

» **feedparser**: a complete RSS/Atom parser that understands all the common — and hundreds of uncommon — feed formats and quirks, and simply and cleanly parses them for you. I’ve not encountered a feed that **feedparser** can’t handle.

» **httplib2**: low-level HTTP library with baked-in support for caching/expiration. I use this for all my RESTful API clients.

» **PIL**: the Python Imaging Library. Again, handles just about every image format under the sun. Sensing a theme here?

» **lxml**: Pythonic bindings to **libxml2**; makes XML parsing a breeze.

» **SciPy**: everything you could do with Matlab, but in Python.

The quality and consistency of the Python ecosystem is amazing.

Finally, Python’s simply a joy to work with.
How’d it work out?

Mark Shuttleworth (founder of Canonical, among other bona fides) observed that managing open source communities is a bit like herding cats, and that the Python community is especially feline.

This is an apt point: the Python community is chock-full of some of the smartest folks I’ve ever had the honor of knowing, but getting everyone on the same page working towards common goals is probably impossible. For example: each of those packages I mentioned earlier has at least one alternative or competitor within the Python community.

Is this a bad thing? I don’t know — competition breeds quality, often — but it is something to be aware of.

We’d like very much for Django to fit smoothly into the greater Python ecosystem. Unfortunately, we’ve dropped the ball here; there’s a small but worrying rift between Django and Python. Some now identify as Django developers, instead of Python developers, and that’s a trend that worries me.

It’s hard: because the Python ecosystem is so diverse, figuring out the best way to interact and fit in is difficult. Still, we need to do a better job here.
Probably the most common question I get about Django and Python in general relates to hiring: how hard is it to find a talented Python developer? Isn’t Java or PHP (or whatever) a safer choice because there’s a bigger employee market?

I won’t lie: Python and Django are job-seekers’ markets right now. If you’ve got Python or Django experience you’ll probably have a bunch of great opportunities. So things are awesome on the employee side here.

On the employer side, I’ve found that in practice this means I need to wait a little longer to fill a position. I can’t get a hundred resumes in the day I post a job ad. However, there’s an important flip-side to this: the people who have Python experience are by far the smartest job candidates I’ve dealt with. I’d hire a Python developer for a job using another technology over someone with domain-specific knowledge.

Look at it this way: most people learn PHP because they want to get a job — PHP is right now the lingua franca of the web. Most people learn Java for the same reason, or perhaps because they learned it in school. If people learn C/C++ at all, they probably learn it in school, too.

However, most people learn Python because they love programming.

Which kind of person do you want working for you?

I should note that this same argument applies for other less-common languages like Ruby, and I’d argue it applies double to exotic and obscure languages like Erlang, Haskell, Smalltalk, etc.

Next choice — a controversial one, at least technically — is Django’s use of “magic” behavior to cut down on the amount of boilerplate code you have to write.
Let’s define some terms first, though. I have to say that I’m not a big fan of calling code “magic” — really, what a silly term! — but I think most experienced developers have a gut feeling about what makes code “magical”. Let’s try to be precise.

For me, “magic” comes down to side-effects: code that does “more” than it seems to.

Of course, doing more with less code is a good thing, so it seems there’s both a “black magic” and a “white magic”, right?

I think what differentiates “good magic” from “bad magic” is obviousness: if you’re using clever coding tricks to cut down on the amount of code, how easy is it to poke inside and figure out what’s going on?

In the case of Python, the mechanism by which magic is expressed is the metaclass. I’m not really going to go into what metaclasses are. Read the Python documentation, or read The Art of the Metaclass Protocol if you’re a language weenie.

Tim Peters — also the author of the Zen of Python, mentioned earlier — is one of the “founding fathers” of the Python language, and he points out that the vast majority of developers have no cause to think about or use metaclasses.
We do.
Let’s talk about why that is.

Face it: web development is boring. Take form processing, for just one example.

It ought to be fun!

From a software design perspective, APIs ought to match behavior, not internals — witness the spread of DSLs — “magic” helps hide implementation details behind a function facade.
So...

One danger is taking it too far.

This book? It teaches Ruby to folks who learned Ruby on Rails first. RoR heavily changes the core behavior of Ruby to the point that if you learn Rails first and then try to use “vanilla” Ruby, you’ll be confused. I speak from experience here.

More pointedly, Django most decidedly did take magic too far in the early years. It made Django very hard to understand, even for experienced Python developers.

So we “removed the magic” — this was our first really big refactoring, and it led to the balance we strike now between obviousness and “cleverness”.

I removed the magic and all I got was this lousy t-shirt.
The downside of using “tricks” is that it can make the internals messy. Put another way, nice public APIs often come at the cost of messy internals. This is a constant struggle for us: recently we rewrote our ORM to work the same but have cleaner internals. It’s an important step to take.

But of course “magic” lets you do a lot more with far less code.

Four years ago I wrote a little toy to track newspaper endorsements of presidential candidates. I wrote it in Python, but using raw CGI. It ran around 5000 LOC. This year’s version: 474 lines, and it’s a lot more powerful.

Better APIs make for better documentation.

This leads to something we’ve called “DDD” — Documentation Driven Development. Write the documentation first, then write the code.

This is good.
We chose to be a “full stack” web framework: Django provides all the basics, right in the box. ORM, template language, controller dispatch, etc.

The alternative is a “best of breed” framework that cherry-picks the best components from the ecosystem and integrates them.

Why’d we choose to do it all ourselves?

Mostly, there wasn’t really a “breed” to select the “best of” at the time.

So that leads to the obvious questions: would you do it differently now?
Best of breed frameworks can also feel “glued together,” with a definite lack of consistency among the various pieces. Not all of them do, of course, but it’s a danger.

So...

We get more control over our bits. We don’t have to push other package maintainers to do what we want; we just do it.

We have to be careful, however, not to repeat Zope’s mistakes. Zope was the first Python web framework — really one of the first web frameworks, actually — and although they did some amazing work, they ended up a very separate community from the Python community since they basically reinvented every wheel possible.

We need to not do this.
And this is why. Bill Joy was talking about companies, but the same is true for communities. It’s a lot easier for open source communities to poach innovation from elsewhere, but if you’re isolated from the larger community the smart people will still stay away.

This is why you have to be friendly and inclusive.

SQLAlchemy, Beaker, … It’s a struggle to keep up.

Another choice: “sensible defaults.” This means providing defaults for nearly every configuration value; you shouldn’t have to fill out a crazy config file just to get started.

The choice of terms is important; we don’t call it “opinionated software” since that’s often an excuse for opinionated developers.
Ego is dangerous. You have to be very careful to keep ego out of the equation, otherwise it drives your development.

Face it: you can’t.
So you have to provide options.

So....

Like I said, it’s very hard to predict what your community wants, so...
You have to listen to your users. Sounds simple; it’s not.

Another choice: we try to emphasize “one obvious way” over “plugable components”. That is, we try to provide a single way to accomplish any single task instead of emphasizing the plugability of different pieces.

Why?

Here’s one reason: these are all the ORMs for Python that I’m aware of (I’m sure there’s more, actually). If you’re starting a project with plugable ORMs, the first thing you have to do is choose one.

See, the thing is that choices are very good for experts...
But not so good for beginners. New users have no frame of reference for these kind of choices; they want (need) to be guided to the easiest choices.

Thing is, there are always more beginners than experts. Especially when you’re trying to foster adoption.

A well-known quip.

Here’s a better way of putting it, from Barry Schwartz’s *The Paradox of Choice*. 
So...

Django is highly consistent and orthogonal. APIs “match” across different areas, and things that work in one area work in others.

This lets us make complicated things easy: we can gloss over the difficult choices for newbies.

However, it can be dangerous: we have to be careful not to reinvent the wheel.
We set our own pace — we’re not bound to the pace of included packages. This is a double-edged sword.

Django isn’t a democracy.

Actually, most open source projects aren’t. However, it’s worth looking at why that is.

So we have a position we call the “BDFL” — Benevolent Dictator for Life.

This is taken from the Python community where Guido van Rossum, the creator of Python, is affectionately known as the Benevolent Dictator for Life.

The thing to know is that it’s tongue it cheek: Guido rarely flexes his power, and that’s key. He only speaks up to end holy wars.
But why have one at all?

Fact is that design — visual or software or whatever — isn’t exactly collaborative.

Let me show you a visual example: admin interface screens.

This is a tool called “lipsaAdmin”. OK looking, I suppose, but cluttered and messy — look at those dates!

Here’s another tool: ActiveScaffold. Yuck.
Here’s another: Streamlined. Meh.

Here’s a common one: phpMyadmin. Yikes!

Here’s Django.

Why does it look so good? Because it’s designed by fiat.

The same applies to software design: important APIs are designed by fiat.

Frankly, the BDFL system works very well for Python, so why should we do anything different?
So...

We can be slow to accept new committers. New addition is by consensus of all existing developers; this means we can mostly speak with a single voice, but it means a very high bar for committers.

If we’re not good at working with non-committing contributors, this can lead to slow development.

We have to delegate whenever possible. I’ll note that when we do, we delegate BDFL-style: the person who writes the code gets the design fiat.
Dictatorships have great quality control.

And it leads to consistency: Adrian and I have the final word, and that means we can ensure a really high degree of consistency.

In the last few minutes I want to talk about the choices we’re facing now, and the ones I think we’ll face in the future.

This is a big one: do we want to go after the long tail of web development — all those millions of tiny sites — or do we want to try to chase down huge deployments? The two might not be compatible.

Let me explain what I mean.
From the Alexa top 25 websites. Do you know what these have in common?

They’re written in PHP. PHP is far from the new hotness of web frameworks — in fact, we frameworky folks like to trash on it a good bit. But it’s by far the most common choice in the top 25.

Rails’ biggest deployment? #643.

Django? #1295, or thereabouts.
Let’s look at this another way.

We just held our first Django conference in California. One of the keynoters was Cal Henderson, the head architect of Flickr (#19). He gave a hilarious talk about why he “hates” Django — find it on YouTube; it’s awesome. I want to cherry-pick a few slides from his presentation.

Here’s how fast you can develop using most tools: it starts off hard, and you steadily get better over time.

Here’s frameworks: it starts off very easy, but at some point you hit a tough and you have to learn some really difficult stuff to move on. Eventually you end up in the same place.

Most apps, however, don’t go very far beyond a first prototype.

Given that, do we care about those big apps that have a productivity trough?
This is somewhat related to “enterprise”
development: really big installations — #2 Yahoo
comes to mind — often shy away from tools like
Django.

But is this really a competition? Do we care about
numbers?

Here’s another thing I’m thinking about right
now.
A common meme these days is that “Rails doesn’t scale” — 4.7 million hits! This mostly comes out of a single developer working on a single site (Twitter) complaining.

For the record, it’s simply not true. Yes, Rails is slower than a lot of tools, but it’s certainly “fast enough”; Rails simply has the same productivity trough that all frameworks do, so you need to learn a lot about Rails before you can scale it up.

Still, there’s now a pretty major backlash against Rails. This is good for us, obviously, but I really feel for the Rails developers. Backlash sucks, and they don’t deserve it.

What will be the backlash against Django? It hasn’t happened yet, so how can we defend against it?

Again, keep out the ego. I think the Rails community set themselves up for personal attack by being personally invested in their software. Hopefully we can avoid personal attacks when the inevitable happens.
Most importantly, though, I think we do a really good job inviting criticism and taking it well.

This is a photo from DjangoCon; at the end of the conference Adrian and I spent a good deal of time taking questions, criticism, and feature requests from the community. Being open to outside suggestions and especially criticism is very important.

Of course you’d like criticism to be constructive, but you have to respond well even to nasty criticism.

Here’s the main takeaway, I think: make conscious decisions. Don’t just stumble into a choice; analyze it, and make a conscious choice.

Thank you!

<jacob@jacobian.org>