The cobbler's children have no shoes, or building better tools for ourselves

Alex Gaynor - PyCon 2016

This talk is about us, as software developers, writing tools for ourselves
About me

- Director of the Python Software Foundation
- Open source contributor
  - Django, PyPy, CPython, pyca/cryptography, etc.
- Washington, D.C. resident
- Bagel and deli enthusiast
- US Digital Service employee
A short history of tools

The thrust of this talk is about writing more tools, so I wanted to briefly cover a history of tools, as I remember it, the trajectory of the past is useful to helping us attempt to build the future.
When I first started programming, about the only thing you could assume a software project would have was a version control system. Back then it wasn’t git either. Issue trackers were pretty common, but not universal!
Gradually, continuous integration became pretty common. A bot that automatically ran all your tests whenever code changed on trunk.
In the past few years, code review has become increasingly common, now many OSS projects and companies have some form of formal code review process. Usually supported by a tool like Gerrit, Phabricator, or Github.
It's also basically expected now that for a project of any serious size, you can automatically deploy to production, using something like fabric or chef or maybe a service like Heroku.
Emerging trends

In 2016, most “healthy” projects have all the things I just described; they’re all great ideas and everyone should do them. Not quite universal yet, but pretty close.

There’s also a set of tools that I see becoming more popular, but aren’t universal yet.
Possibly the most powerful feature Travis CI introduced was the ability to automatically run all your tests on a pull request, in addition to running them on master. Now you know your tests pass before you merge a patch.

It's pretty common to see this practice in OSS, largely because of Travis CI, but it's not quite as common at company's yet. It's possible to configure Jenkins and Buildbot to do this, so I think folks would be well served to adopt this.
Linting

(flake8, bandit, flake8-import-order, etc.)

Running “lints” which check for basic style conformance are also becoming more popular. You may think of these as just tools that check that your code base is properly PEP8, but there are also more tailored linters, for example bandit, which checks for bad security practices.

One note I’ll make about linters is that other communities, such as Rust and Go’s, have been moving away from style-checking-linters towards auto-formatters. Instead of flake8 telling you what’s wrong, go fmt will simply fix it for you. This is super powerful, and I hope this pattern get’s uptake in the Python community.
coverage.py has been around for a while, but integrating regular coverage measurements into your CI pipeline is an exciting new trend. This means whenever you merge a PR you can be informed about what impact it’ll have on your overall coverage, and how well tested the new lines of code in the PR are.
You’re a large company, you have a bunch of projects that all interoperate, you’re often not sure which code base implements some functionality, or you want to find all places a certain function is used.

Livegrep gives you nearly instantaneous regex search over a large amount of code, helping you to manage the complexity of working with multiple projects.
By analyzing the blame information on this pull request, we identified @reaperhulk, @public and @alex to be potential reviewers.

github.com/facebook/mention-bot
Ok. So those were tools I see getting adoption now, and I imagine in a PyCon or two, all of those will be as common as the first set of tools I mentioned.

The other thing that's becoming more standard is a workflow around small branches, send a pull request, get code reviewed, and then someone merges your PR.

The rest of the material in this talk basically assumes you have these processes and tools.
So the rest of this talk is about how to build better tools for your specific development team, and you’re specific development process. As developers, we have the ability to write computer programs, but too often our own teams work on a hodgepodge of undocumented and manual processes. There’s all sorts of simple and easy to create tools that your team might want for it’s own process.
This is because my central thesis is that, whenever possible, you should encode the process your team follows into tooling, rather than implement it by hand.

This is because:
- Automation scales. Human enforced process tends to break down as your team gets larger.
- Changing the process becomes an act of sending a pull request, rather than updating a prose document, or trying to inform your whole company that something is changing. This has the benefit of giving you something concrete to discuss whether a change is good, and contributing to the question of scale.
Good news. If we're going to implement some tools, lots of our existing tools have APIs we can leverage to integrate with them.

The examples I'm going to use for the rest of the talk all use Github and its APIs, but these concepts should translate to any other issue tracking/code review system; I'm just using Github because it's familiar to most folks and publicly accessible.

So what kinds of APIs do we have available?
Issues

- Create an issue
- Add/remove labels
- Add a comment
- Assign to someone

Some of the operations we might want to perform an issue are:
- Create a new one: if we have a program that knows there's a problem, we can automatically file an issue for it. Lots of places I see just send emails. An issue gives us all the tracking and other tools we might want
- Mess with labels or assignee to keep stuff organized
- Leave comments to add additional information or context
Pull requests

- Send a PR
- Assign a PR
- Add/remove labels
- Leave a code review
- Add a commit status

Things we can do with a pull request:
- Send one. We can have a program change our code somehow and turn it into a pull request
- More organizational stuff
- We can have a bot perform some sort of automatic code review, link a linter on steroids
- Github has the idea of commit statuses, so we can mark something as passing/failing so people know whether it’s safe to merge a PR or not.
Cool, let's look at some examples of how this works
To get started, let's install github3.py, which a great set of Python bindings to Github's API, by Ian Cordasco.

```bash
$ pip install github3.py
```
import github3

gh_client = github3.login(
    os.environ['GITHUB_USERNAME'],
    os.environ['GITHUB_PASSWORD'],
)

We get ourselves a github client, I’ve stored the username/password for this account in the OS environment. Small note: for any sort of production tools, don’t use your own account’s username/password. Create a second account to run your bots on and create single-use keys for that account, with just the permissions you need for whatever you’re working on.
repo = gh_client.repository("django", "django")

Now we get a repository object. This, not surprisingly, represents a repo on Github, and has methods for most of the API calls we'll want to make.
Ok. So let's say something we want to do is scan our websites to see if any of them have an HTTPS certificate that's going to expire soon, and if it going to expire soon, we want to file a ticket, so someone can investigate and get a fresh cert.
def get_expiration_date(host):
    ssl_context = ssl.create_default_context()
    with closing(socket.socket(socket.AF_INET, socket.SOCK_STREAM)) as sock:
        sock = ssl_context.wrap_socket(sock, server_hostname=host)
        sock.connect((host, 443))
        expiration = ssl.cert_time_to_seconds(sock.getpeercert()["notAfter"])
    return datetime.datetime.fromtimestamp(expiration)

Ok, so we start with a simple function to get the expiration date of an SSL certificate. Relatively straightforward, we create a connection to the remote server, get their certificate, and convert the notAfter date to a Python datetime object. Not going to spend a ton of time on the details here, since it’s somewhat orthogonal to our subject.
while True:
    for host in MY_DOMAINS:
        expiration = get_expiration_date(host)
        if expiration - datetime.datetime.now() < CUTOFF:
            file_an_issue(gh_client, host, expiration)
    time.sleep(3600)

Ok, now let’s start putting it together. We’ve loop forever, we want this running always. Now for each domain we have, we get the expiration date. If that expiration minus the current time, is less than some cutoff, say, a month, we file an issue.

Then we sleep for an hour.

Ok, so now we just write the bits to file an issue.
Now we create an issue. It's going to the django repo on the django organization. The title tells us there's a cert expiring soon, and the body lets us know when it expires. We assign it to alex, and attach an ssl-cert label. Now everyone can see there's an issue that needs to be worked.

Small exercise for the reader: right now we'll file an issue every time we go through the loop, or once an hour. We should change this code to not file an issue if there's already an open one.
Auto-labeling

Now let's look at an example of playing with pull requests. Our company has a great security team, but there's a ton of development work going on, and they're not always sure where they need to be looking. We've created a security label to help them know what PRs are security sensitive, but people forget to use it a lot of the time. Let's create a bot to automatically add the security label whenever our issue touches the crypto files.
This bot will leverage github’s webhooks. With a typical API, you make an HTTP request to github’s servers. With a webhook, Github makes an HTTP request to our server. We tell Github what events we care about, and it’ll notify us whenever they happen.
def github_webhook(request):
    event = request.headers.get("X-Github-Event")
    if event != "pull_request":
        return Response(status=200)

    body = json.load(request.stream)
    if body.get("action") not in {"opened", "reopened", "synchronize":
        return Response(status=200)

So we get started with a bit of web hook boilerplate. You’ve got a web server somewhere, running your favorite web framework. Github includes a header that tells us what type of event this is. We only care about pull request events. Then we load the body of the request as JSON. Inside there there’s an action field. We only care about a few cases, we care about opened, reopened, and synchronize. synchronize is the case where someone pushed new commits to a pull request.
We grab the number field out of the request body, and we get the issue with that number. Small conceptual note: in GitHub, pull requests are both issues and PRs. So you see we get a PR object right afterwards using the same number.

`iter_files()` let's us iterate through every file changed in this PR, we check if our crypto file has been changed, and if it has we add a label.

You can imagine going even farther, if only the crypto file is changed, assign a reviewer from the security team.
Other ideas

So these are a few concrete examples of how to build a tool for your team, leveraging these APIs. Here’s a few more examples you might consider.
You've got 100 projects at your company that use Django. When a new django security release comes out, you need to update the requirements.txt for all of them. Create a bot that goes through each of your repos, and sends a pull request increasing Django to the latest version. Now your security team can easily send PRs to all the dev teams without a lot of work.
It’s no secret that reviewing UI changes sucks. Instead build a system of taking screenshots and reviewing them [explain].
You want things to be reviewed by 2 people, including one front-end and one backend engineer. Encode that in a bot that you drive with comments + commit status.
Thanks!
Questions?

https://github.com/alex
https://speakerdeck.com/alex