Hi.
Open Source
Requests
HTTP for Humans

```python
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf-8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json
{u'private_gists': 419, u'total_private_repos': 77, ...}
```
$ curl http://httpbin.org/get?test=1
{
  "url": "http://httpbin.org/get",
  "headers": {
    "Content-Length": "",
    "Connection": "keep-alive",
    "Accept": "**/*",
    "User-Agent": "curl/7.21.4 ...",
    "Host": "httpbin.org",
    "Content-Type": ""
  },
  "args": {
    "test": "1"
  },
  "origin": "67.163.102.42"
}
Et Cetera

- Legit: Git Workflow for Humans
- Envoy: Subprocess for Humans
- Tablib: Tabular Data for Humans
- Clint: CLI App Toolkit
- Autoenv: Magic Shell Environments
- OSX-GCC-Installer: Provokes Lawyers

275+ More
I am not a data scientist.

I’d like to be, but the APIs are painful.
What do we have in common?
We’re Makers.

We craft experiences & interfaces.
Developers!

Developers, Developers, Developers.
People are going to be spending two or three hours a day with these machines — more than they spend with a car.

— Steve Jobs, 1983
Software design must be given at least as much consideration as we give automobiles today — if not a lot more.

— Steve Jobs, 1983
That worked.
Beautiful Interfaces.

Today, beautiful applications abound.

- Industrial Design
- Web Interfaces
- iOS, Android, Mobile Apps
- Desktop Clients & Applications
Hackers are the real Makers.
Developers spend 8+ hours a day with APIs.

Why are they treated differently?
Everything is a remix*. 

* APIs Rule Everything Around Us.
How?
Step I: Have an Issue.
A Real, Tangible Problem.

You can’t solve a problem properly if you don’t experience it firsthand.
Example: GitHub

Over two million people collaborating.

- GitHub wasn’t built for the developer community at large.
- Resonated with millions of developers.
- They themselves happen to be developers.
Other’s Success

- Gumroad, built for the founder.
- 37 Signals product, build for the team.
- Ruby on Rails, by Rubyists for Rubyists.
Optimization

What drives your decisions?

- Feature driven development?
- Profit driven development?
- Growth driven development?
- Problem driven development.
pragmatic |pragˈmatɪk|, adj:

Dealing with things sensibly and realistically in a way that is based on practical rather than theoretical considerations
We know Ruby...

```ruby
require 'net/http'
require 'uri'

uri = URI.parse('https://api.github.com/user')

http = Net::HTTP.new(uri.host, uri.port)
http.use_ssl = true

req = Net::HTTP::Get.new(uri.request_uri)
req.basic_auth('username', 'password')

r = http.request(req)

puts r
```
Python’s net/http?

http/url/lib/2
Several hours later...
import urllib2

gh_url = 'https://api.github.com/user'

req = urllib2.Request(gh_url)

password_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_manager.add_password(None, gh_url, 'user', 'pass')

auth_manager = urllib2.HTTPBasicAuthHandler(password_manager)
opener = urllib2.build_opener(auth_manager)

urllib2.install_opener(opener)

handler = urllib2.urlopen(req)

print handler.read()
import re

class HTTPForcedBasicAuthHandler(HTTPBasicAuthHandler):

    auth_header = 'Authorization'
    rx = re.compile('(?:.*,)*[ \t]*([^ \t]+)[ \t]+' + 'realm=(\"\'\')(.*?)/\2', re.I)

    def __init__(self, *args, **kwargs):
        HTTPBasicAuthHandler.__init__(self, *args, **kwargs)

    def http_error_401(self, req, fp, code, msg, headers):
        url = req.get_full_url()
        response = self._http_error_auth_reqed('www-authenticate', url, req, headers)
        self.reset_retry_count()
        return response

http_error_404 = http_error_401
Admit it.
You’d leave and never come back.
This is a serious problem.

HTTP should be as simple as a print statement.
APIs For Humans
Let’s Break it Down.

What is HTTP at its core?

- A small set of methods with consistent parameters.
- HEAD, GET, POST, PUSH, PUT, PATCH, DELETE, &c.
- They all accept Headers, URL Parameters, Body/Form Data.
Enter Requests.
HTTP for Humans.
import requests

url = 'https://api.github.com/user'
auth = ('username', 'password')

r = requests.get(url, auth=auth)
print(r.content)
Achievement Unlocked!

- A small set of methods with consistent parameters.
- HEAD, GET, POST, PUSH, PUT, PATCH, DELETE, &c.
- They all accept Headers, URL Parameters, Body/Form Data.
Requests Success

- Python is a language built for Humans.
- Why should HTTP be non-trivial?
- I explored and discovered what I really needed, and built it.
- I had a real problem that I solved for myself.
Requests Success

• At first, Requests was far from powerful.

• But, it deeply resonated with people.

• Features grew over time, but the API was never compromised.

• Quite popular (48,000,000 downloads).
Developers spend 8+ hours a day with APIs.

Build for yourself—a developer.
Step II: Respond.
Write the README.
• Before any code is written, write the README — show some examples.

• Write some code with the theoretical code that you’ve documented.
Achievement Unlocked!

• Instead of engineering something to get the job done, you interact with the problem itself and build an interface that reacts to it.

• You discover it. You respond to it.
Sculptures, Etc.

- Great sculptures aren’t engineered or manufactured—they’re discovered.
- The sculptor studies and listens to the marble. He identifies with it.
- Then, he responds.
- Setting free something hidden that inside all along.
Responsive Design

• It’s not about a design that will “work” on a phone, tablet, and desktop.

• It’s about making something that identifies itself enough to respond to the environment it’s placed in.

• Free of arbitrary constraints.
Readme-Driven Development?

Responsive API Design.
Step III: Build.
Responsive Design

- Once you discover the API: build it.
- Write all the code necessary to make exactly what you documented happen.
- Complex code? Layer your API.
- “Porcelain” layer is documented.
The API is all that matters.

Everything else is secondary.
Do unto others as you would have them do to you?

Build tools for others that you want to be built for you.
CONSTRAINTS
FOSTER
CREATIVITY
Open Source
All The Things!
Build for Open Source

- Components become concise & decoupled.
- Concerns separate themselves.
- Best practices emerge (e.g. no creds in code).
- Documentation and tests become crucial.
- Code can be released at any time.
Recommended Tools

- Anaconda: Python distribution for humans and scientists (“it just works”)
- Heroku: turnkey machine resources.
- Conda Buildpack for Heroku.
Simplicity is always better than functionality.

— Pieter Hintjens