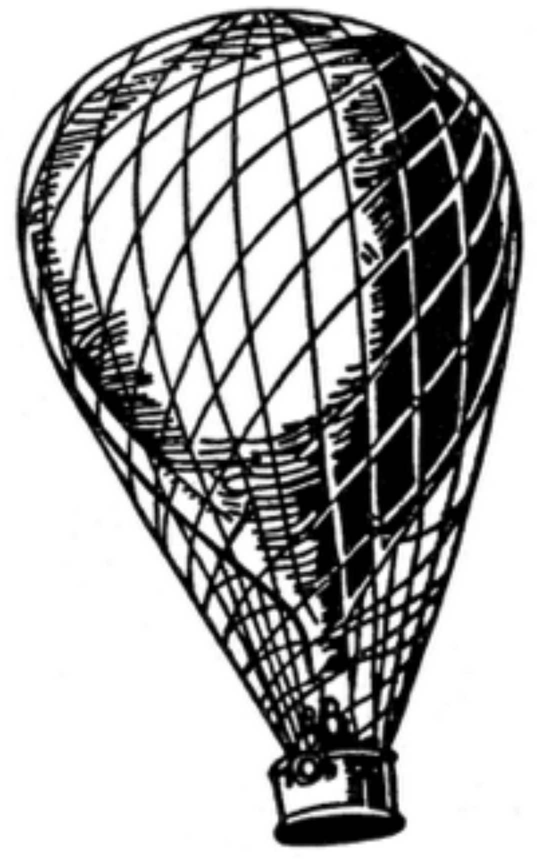

Letters from the Battlefield

Armin *@mitsuhiko* Ronacher

Flask

web development,
one drop at a time

Lektor

SENTRY

I like to review code and design APIs

design for maintainability and security

“if I could do it again …”

so here are some lessons learned

PREFACE

the thing about overengineering

overengineering |ˈōvərˌenjəˈniriNG|
noun
the designing of a product to be more robust or complicated than is necessary for its application

a lot of what's in this talk is often seen as “unnecessary” developers are afraid of complexity and initial overhead but the right solutions were often already created; use them

PROLOGUE

being afraid of changes

afraid |əˈfrād|
adjective
*worried that something undesirable will occur or be done: he was **afraid that** the farmer would send the dog after them*

changes

- developers should never feel afraid of code changes
- developers should not be afraid of the first change
- developers should feel comfortable doing big changes
- developers should not accidentally produce security problems

bite size chunks

- write code so that developers are never overwhelmed
  - neither on making new features
  - nor on changing existing code
- simplifies code review

the goal is to make developers confident and happy

CHAPTER 1

where is the state?

state |stāt|
noun

the particular condition that someone or something is in at a specific time

state in programming

- Most prominent languages are rich in state
- But poor in explicitly managing it
- Most programmers do not know how their own state works
- No rules when mutable state becomes assumed constant state

why is that a problem?

- Most prominent languages are rich in state
- But poor in explicitly managing it
- Most programmers do not know how their own state works

practical example

```
from functools import update_wrapper
from django.conf import settings

def might_debug(f):
    def new_func(*args, **kwargs):
        if settings.DEBUG:
            do_some_debug_stuff()
        return f(*args, **kwargs)
    return update_wrapper(new_func, f)
```

is ‘settings’ mutable?

- it's python, so the answer is yes
- however at which point is it safe to modify them?
- what if people drag out state to an unsafe scope?

decision made

```
from functools import update_wrapper
from django.conf import settings

if settings.DEBUG:
    def might_debug(f):
        def new_func(*args, **kwargs):
            do_some_debug_stuff()
            return f(*args, **kwargs)
        return update_wrapper(new_func, f)
else:
    might_debug = lambda x: x
```

module state in python

- imports are stateful
- module scope is stateful
- this influences code we write in Python
- modules in Python are giant singletons
- the scope of state can be hidden

hidden state

```
from flask import request


def is_api_request():
    return bool(request.headers.get('Authorization'))
```

“Every once a while the error messages are Spanish”

decisions made from hidden state

```
>>> from django.utils.translation import ugettext
>>> ugettext('Hmmmm')
u'Hmmmm'
```

decisions made from hidden state

```
from django.utils.translation import ugettext

class LoginForm(…):
    ERROR = ugettext(u"Could not sign in")
```

decisions made from hidden state

```python
def handle_request(request):
    endpoint, args, kwargs = match_request(request)
    func = import_view_function(endpoint)
    return func(*args, **kwargs)
```

CHAPTER 2

shackle the state!

shackle |ˈSHak(ə)l|
verb

restrain; limit: they seek to shackle the oil and gas companies by imposing new controls.

stateful APIs suck

- nobody likes stateful APIs
- in particular nobody likes APIs that randomly change behavior

ideal state management

- create scope
  - set up initial working conditions (modify here)
  - execute code
  - clean up state
- destroy scope

prevent access

- If something is not there, say so, not not fall back
- translations should not silently become idempotent calls

raise if accessed in bad scope

```
>>> from flask import request
>>> request.headers
Traceback (most recent call last):
  …
RuntimeError: Working outside of request context.
```

prevent modifications

```
with settings.transaction() as t:
    t.CONFIG_VALUE = 42

settings.close()
```

prevent stupid code

```
>>> settings.transaction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Settings are closed. No more modifications
```

CHAPTER 3

import madness

madness |ˈmadnəs|
noun

the state of being mentally ill, especially severely.

the art of importing

- import all
- upfront
- do not import at runtime
- there be many evil backstabbing dragons

import all stuff

```
from werkzeug.utils import find_modules

def import_all(pkg):
    for module in find_modules(pkg, recursive=True):
        __import__(module)

import_all(__name__.split('.')[0])
```

why?

- importing requires locks; imports can be recursive
- imports have side effects, let's get it done early
- both those things are bad
- once it's imported, it's cached
- after that things become much, much more predictable

circular dependencies

- good luck with that ;-)
- I do not have a good response to this.

CHAPTER 4

make it searchable

search |sərCH|
verb

try to find something by looking or otherwise seeking carefully and thoroughly: I searched among the rocks, but there was nothing

why?

- new developers need to understand context
- when you have conceptional security issues you need to find things
- aids code review

what's ‘searchable’

- assume your only tool is `grep`
- write code so that you can grep/full text search it
- it will be worth it

things that are easily grep-able

- decorators!
- explicit and clear function and class names
- special methods
- avoid funky operator overloads if they do something non-standard

CHAPTER 5

predict common behavior

predict |prəˈdikt|
verb

say or estimate that (a specified thing) will happen in the future or will be a consequence of something: he predicts that the trend will continue

my least favorite code

```
import json
from django.http import HttpResponse

def view_function(request):
    some_data = generate_some_data(…)
    return HttpResponse(json.dumps(some_data),
                        mimetype='application/json')
```

what about this?

```python
from myproject.api import ApiResponse

def view_function():
    some_data = generate_some_data(…)
    return ApiResponse(some_data)
```

why?

- we establish “request context”
- we define a clear common case of “this is the result of an API”
- we can transform and handle data on the way out

what do we gain?

- JSON encode security issues? One clear point to handle it
- Need to support a custom mimetype? Change all in one go
- Instrumentation? One common object

convert common values

```
def handle_request(request):
    rv = dispatch_request(request)
    if isinstance(rv, ApiResponse):
        rv = Response(json.dumps(rv),
                      mimetype='application/json',
                      status=rv.status_code)
    return rv
```

CHAPTER 6

define context

context |ˈkäntekst|
noun
the circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood and assessed

what is context

- runtime context (“scopes”)
- data context (“transfer encodings”)
- security context (“who is the actor?”)

context behavior

- what happens based on context?
- how does data look like?
- how does context influence what is happening?

examples of scoped context

- current language
- current http request
- current authenticated user
- current access restrictions

implied context

```
>>> from myapp.i18n import ugettext, set_language
>>> with set_language("en_US"):
...     ugettext("Sign in")
...
u"Sign in"
>>> with set_language("de_DE"):
...     ugettext("Sign in")
...
u"Anmelden"
```

context for data

- object in string context
- object in HTML context
- object serialization

data in context

```
>>> from markupsafe import Markup, escape
>>> unicode(my_user)
u"Peter Doe"
>>> escape(my_user)
u'<a href="/users/42/">Peter Doe</a>'
>>> Markup("<em>%s</em>") % my_user
u'<em><a href="/users/42/">Peter Doe</a></em>'
>>> print json.dumps(my_user)
{"username": "Peter Doe", "id": 42}
```

CHAPTER 7

prevent misuse

misuse |ˌmisˈyo͞os|
noun
the wrong or improper use of something: a misuse of power.

context for improved security

```
from myapp.db import Model, Query
from myapp.access import get_available_organizations

class Project(Model):
    …

    @property
    def query(self):
        org_query = get_available_organizations()
        return Query(self).filter(
            Project.organization.in_(org_query))
```

automatic escaping

- Template engines escape data automatically by HTML rules
- However HTML is complex in behavior (script tags, attributes etc.)
- It becomes possible to accidentally misuse things
- People will get it wrong, so worth investigating the options

JSON in HTML

- Common case to send JSON to HTML
- Two areas of concern: HTML attributes and `<script>` tags
- How to escape in those. Common case? Can we make one function for both?

example escaping

```
>>> from flask.json import htmlsafe_dumps
>>> print htmlsafe_dumps("<em>var x = 'foo';</em>")
"\u003cem\u003evar x = \u0027foo\u0027;\u003c/em\u003e"
```

result of this exercise

- does not produce any HTML entities
- now works in `<script>` …
- … as well as single quoted attributes
- falls over very obviously in double quoted attributes
- it's pretty clear how it's supposed to work and hard to misuse

think before you act!

Q&A