

# Good API Design

Study, Improve & Create

Armin Ronacher — <http://lucumr.pocoo.org/>

# Who am I

- Armin Ronacher (*@mitsuhiko*)
- Founder of the Pocoo Team
- we do Jinja2, Werkzeug, Flask, Sphinx, Pygments etc.

# What is an API?

ap·pli·ca·tion pro·gram·ming in·ter·face (*abbr.:* **API**)

*noun* Computing

*an interface implemented by a software program that enables it to interact with other software.*

# API Requirements

A Gentlemen's Agreement

# A Good API

- Easy to learn
- Usable, even without a documentation
- Hard to misuse
- Powerful and easy to extend

# A Good API

- Easier to use than to re-implement equal functionality
- Consistent
- Abstract interface that does not limit performance and scaling

# Bad Examples

Learn from other's mistakes

# Bad Examples

- Windows API
- Java's IO System
- POSIX and the C standard library
- Parts of the Python Standard Library



# Windows API

- **Task:**
  - execute an application
  - wait for it to close
  - continue doing what you were doing

# How it Works

```
SHELLEXECUTEINFO shinfo;
memset(&shinfo, 0, sizeof(SHELLEXECUTEINFO));
shinfo.cbSize = sizeof(SHELLEXECUTEINFO);
shinfo.hwnd = calling_window_handle;
shinfo.lpVerb = "open";
shinfo.lpFile = "notepad.exe";
shinfo.lpParameters = "\"C:\\Path\\To\\File.txt\"";
shinfo.nShow = SW_NORMAL;
shinfo.fMask = SEE_MASK_NOCLOSEPROCESS;
int rv = ShellExecuteEx(&shinfo);
if (rv)
    WaitForSingleObject(shinfo.hProcess, INFINITE);
```

# The Problems

- Ugly :-)
- Put size of struct into struct
- No defaults at all
- Huge Security Problem
- Platform specific

# Expected API

```
const char *args[3];  
args[0] = "notepad.exe";  
args[1] = "C:\\Path\\To\\File.txt";  
args[2] = NULL;  
ShellExecuteAndWait(args);
```

# Read Textfile into String

- **Task:**
  - Open a textfile
  - Read whole contents
  - return string decoded from UTF-8
  - may raise an IO exception but nothing else (checked exceptions FTW?)

# How it Works

```
import java.io.*;

public class ReadFile {
    public static String readFile(String filename)
        throws IOException {
        InputStreamReader r;
        int read;
        try {
            r = new InputStreamReader(
                new FileInputStream(filename), "UTF-8");
        }
        catch (UnsupportedEncodingException uee) {}
        try {
            StringBuffer buf = new StringBuffer();
            char tmp[] = new char[1024];
            while ((read = r.read(buf, 0, 1024)) > 0)
                buf.append(tmp, 0, read);
        }
        finally {
            r.close();
        }
        return buf.toString();
    }
}
```

# The Problems

- Requires dealing with explicit remembering of the number of chars read
- requires three classes (StringBuilder, InputStreamReader, FileReader)
- requires catching of exception that can't happen (UTF-8 is required to be supported)

# Expected API

```
import java.io.*;

public class ReadFile {
    public static String readfile(String filename)
        throws IOException {
        return new File(filename).getStringContents("UTF-8");
    }
}
```



# POSIX / C

- An amazing example of how an API can limit performance
- Also an astonishing example of how security can be affected by bad design decisions -> `getc()` / `sprintf()` etc.
- **Task:**
  - Get current working directory

# The Naive Way

```
int
main(void)
{
    char *buffer[1024];
    getwd(buffer);
    printf("Current working dir: %s\n", buffer);
}
```

# Slightly Improved

```
int
main(void)
{
    char *buffer[1024];
    getcwd(buffer, 1024);
    printf("Current working dir: %s\n", buffer);
}
```

# Still wrong, why?

- `curwd()` -> same problem as `getc()`
- `getcwd()` -> however might return a NULL pointer on errors which not many people know.
- When NULL and `errno ERANGE` you have to call again with higher buffer size.

# How to use that API ...

```
char *
get_current_working_directory(void)
{
    size_t bufsize = 1024;
    char *buffer = malloc(bufsize);
    while (1) {
        char *rv = getcwd(buffer, bufsize);
        if (rv)
            return rv;
        if (errno == ERANGE) {
            char *tmp = realloc(buffer, (size_t)(bufsize *= 1.3));
            if (!tmp)
                goto abort_error;
            buffer = tmp;
        }
        else
            goto abort_error;
    }
}

abort_error:
    free(buffer);
    return NULL;
}

int
main(void)
{
    char *cwd = get_current_working_directory();
    printf("Current working dir: %s\n", buffer);
    free(cwd);
}
```

# Things to learn

- That API was nice and simple for the time
- Then very long path names came around
- Also that API was designed for different memory areas for early efficiency reasons (stack versus heap)

# Not limited to getcwd

- All syscalls on POSIX can be interrupted (simplified by BSD)
- calls to open/close/read etc. have to be checked for EINTR
- Who checks for EINTR?

# EINTR

```
mitsuhiko at nausicaa in ~  
$ python  
Python 2.7 (r27:82508, Jul 3 2010, 21:12:11)  
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin  
Type "help", "copyright", "credits" or "license" for more  
information.  
>>> import sys  
>>> sys.stdin.read()  
^Z  
[1]+  Stopped                  python
```

```
mitsuhiko at nausicaa in ~ exited 146 running python  
$ fg  
python  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IOError: [Errno 4] Interrupted system call
```



# Python Standard Library

- Just a few examples:
  - `Cookie.Cookie`
  - `cgi.parse_qs`

# Cookie

- Nearly impossible to extend, requires use of undocumented APIs
- Was necessary when browsers started supporting the HttpOnly flag
- Discards all cookies if a part of a cookie is malformed (bad)
- You don't want to see the code...

# Just in Case

```
class _ExtendedMorsel(Morsel):
    _reserved = {'httponly': 'HttpOnly'}
    _reserved.update(Morsel._reserved)

    def __init__(self, name=None, value=None):
        Morsel.__init__(self)
        if name is not None:
            self.set(name, value, value)

    def OutputString(self, attrs=None):
        httponly = self.pop('httponly', False)
        result = Morsel.OutputString(self, attrs).rstrip('\t ;')
        if httponly:
            result += '; HttpOnly'
        return result

class _ExtendedCookie(SimpleCookie):

    def _BaseCookie__set(self, key, real_value, coded_value):
        morsel = self.get(key, _ExtendedMorsel())
        try:
            morsel.set(key, real_value, coded_value)
        except CookieError:
            pass
        dict.__setitem__(self, key, morsel)

def unquote_header_value(value, is_filename=False):
    if value and value[0] == value[-1] == '"':
        value = value[1:-1]
        if not is_filename or value[:2] != '\\\\':
            return value.replace('\\\\', '\\').replace('\\"', '"')
    return value

def parse_cookie(header):
    cookie = _ExtendedCookie()
    cookie.load(header)
    result = {}
    for key, value in cookie.iteritems():
        if value.value is not None:
            result[key] = unquote_header_value(value.value)
    return result
```

# cgi.parse\_qs

- Depending on the (user controlled input) you get different types back
- Might be a string, might be a list
- Useless interface for any stable real-world code.
- That function can't be used, use `cgi.parse_qsl` instead.

# Become a Designer

Because every programmer is an API designer

# Basic Principles

What you always have to keep in mind

# General Rules

- Start building applications with the API
- Think in terms of APIs
- Even if you will always be the only programmer on that thing
  - because you should never assume you will be [*success, handing over maintenance etc.*]

# Implementation vs Interface

- Interface must be independent of implementation
- Don't let implementation details leak into the API (exceptions, error codes, etc.)



# Implementation vs Interface

```
>>> from cStringIO import StringIO
>>> from pickle import load
>>> load(StringIO('Foo'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: o
>>> load(StringIO('d42'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> load(StringIO("S'foo'\n"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
EOFError
```

# Performance and Scaling

- Bad decisions limit performance
  - make things immutable or document them to be immutable
  - Account for concurrency that are not threads or processes
  - Be reentrant

# Performance and Scaling

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE.utf-8')
'de_DE.utf-8'
>>> locale.atof('42,42')
42.42
>>> locale.setlocale(locale.LC_ALL, 'en_US.utf-8')
'en_US.utf-8'
>>> locale.atof('42.42')
42.42
```

# Be consistent and nice

- Consistent naming
- Follow naming rules of platform
  - PEP 8
  - If you develop library for twisted etc. follow theirNamingRules.
  - Don't go down the DSL road

# Be consistent and nice

```
threading.currentThread()  
unittest.TestCase.assertEqual()  
logging.getLoggerClass()  
logging.getLogger()
```

```
thread.get_ident()  
sys.exc_info  
cgi.parse_multipart()  
urllib.proxy_bypass_environment()
```

```
sys.getfilesystemencoding()  
sys.getdefaultencoding()  
urllib.addurlinfo()  
wave.Wave_read.getnchannels()
```

# Library vs Framework

- A library provides functions, methods and classes to accomplish things.
- A framework might throw meta magic on top of that.

# Library vs Framework

```
def login(environ):
    form = werkzeug.parse_form_data(environ)[1]
    if check_credentials(form['username'],
                        form['password']):
        remember_user(...)

@app.route('/login')
def login():
    if check_credentials(request.form['username'],
                        request.form['password']):
        remember_user(...)
```

# Class Design

Python has classes, so will your code



# Design for Subclassing

- Build your class so that a subclass might improve / change certain behavior
- Provide ways to hook into specific parts of the execution.
- If class is not designed for subclassing, document it as such

# Defaults / Common Use Cases

- Think of the most common use cases, you will have them if you use your API
- Make sure the API provides easy ways to do that
- If you see that your code does things the API should be doing instead, move that specific code over.

# POLS

- An API should not surprise the user (POLS)
- Do not introduce side effects into methods that hint not having side effects.
- getters, properties should never have side effects.
- Metaclasses allow breaking users expectations on so many levels.

# POLS

```
public class Thread implements Runnable {  
    /* Tests whether the current thread has been  
       interrupted. The interrupted status of the thread  
       is cleared by this method.  
  
       In other words, if this method were to be called  
       twice in succession, the second call would return  
       false. */  
    public static boolean interrupted();  
}
```

# Consistent Parameters

- Ordering of parameters is important.
- What you're operating on should always be the first parameter.
- Similar methods should have same ordering of parameters and types.
- If the order is the wrong way round, stick with it! Consistency more important.

# Consistent Parameters

```
char *strcpy(char *dst, const char *src);  
void bcopy(const void *src, void *dst, size_t n);
```

# Interfaces and Strings

“Stringly typed”

# Data structures not Strings

- If users have to parse return values of APIs you are doing something wrong.
- If an implementation detail becomes an interface it prevents future improvements.



# Data structures not Strings

```
>>> import imaplib
>>> srv = imaplib.IMAP4('example.com')
>>> srv.login('username', 'password')
('OK', ['Logged in.'])
>>> srv.list()
('OK', ['(\\HasChildren) "." "Folder"',
        '(\\HasNoChildren) "." "Folder.Subfolder"'])
```

# Other Practical Advice

do away with the global state

# Global State in Python

- `Module globals` -> global state
- `sys.modules` -> global state
- any kind of singleton -> global state

# Don't do this

```
import mylib

@mylib.register('something')
def callback_for_something(args):
    ...

mylib.start_execution()
```

# Do this instead!

```
import mylib

worker = mylib.Worker()

@worker.register('something')
def callback_for_something(args):
    ...

worker.start_execution()
```

# Things to learn from Java

*Classes are a good invention*

# Advantages of Classes

- Create as many objects as necessary
- simplifies tests a lot where exceptions are expected
- no cleanup necessary, GC/refcounting does that for us
- run with more than one configuration, just create one more instance.

# Bad Examples

- Django's global settings module
- Celery used to have this as well, it changed recently for precisely this reason.
- `csv / logging / sys.modules` in the standard library.



# Conclusions

What we learned

# API Design

- Proper API design is what makes people use your library
- An API that is easy to understand lowers the entry barrier for a new programmer
- API design is tough
- Even large companies got it wrong



go ahead and ask :)

Slides at <http://lucumr.pocoo.org/projects/>

# Copyright and Legal

- Slides (c) Copyright 2010 by **Armin Ronacher**
- Licensed under the *Creative Commons Attribution-NonCommercial 3.0 Austria License*
- Some of the slides based on an earlier presentation called "*How to Design a Good API and Why it Matters*" by **Joshua Bloch**