BINARY PYTHON

introducing native code

Armin @mitsuhiko Ronacher

Hi I'm Armin

Hailing from ice cold Vienna Austria

(where stores are closed on sundays)

Rottenegg
Gemeinde St. Gotthard

Windpassing

SENTRY

I <3 AND BUILD OPEN SOURCE

Werkzeug, Jinja, Flask, Sentry, …

OUR HEART BEATS FOR Python

But we also have other things we need to interface with …

C / C++ / RUST

WHY DO WE HAVE NATIVE CODE?

SPEED

Functionality

NECESSITY

IMPORTING NATIVE MODULES

Import System

- package/mylib.so
- package/mylib.pyd
- package/mylib.dylib

Local Development

- lib/lib.c -> package/_lib.so
- `python setup.py build`
- `pip install --editable . -v`

Build for Distribution

- lib/lib.c -> build/…/_lib.so
- `python setup.py bdist_wheel`

WHY HANDROLL

Many Systems

Many Developers

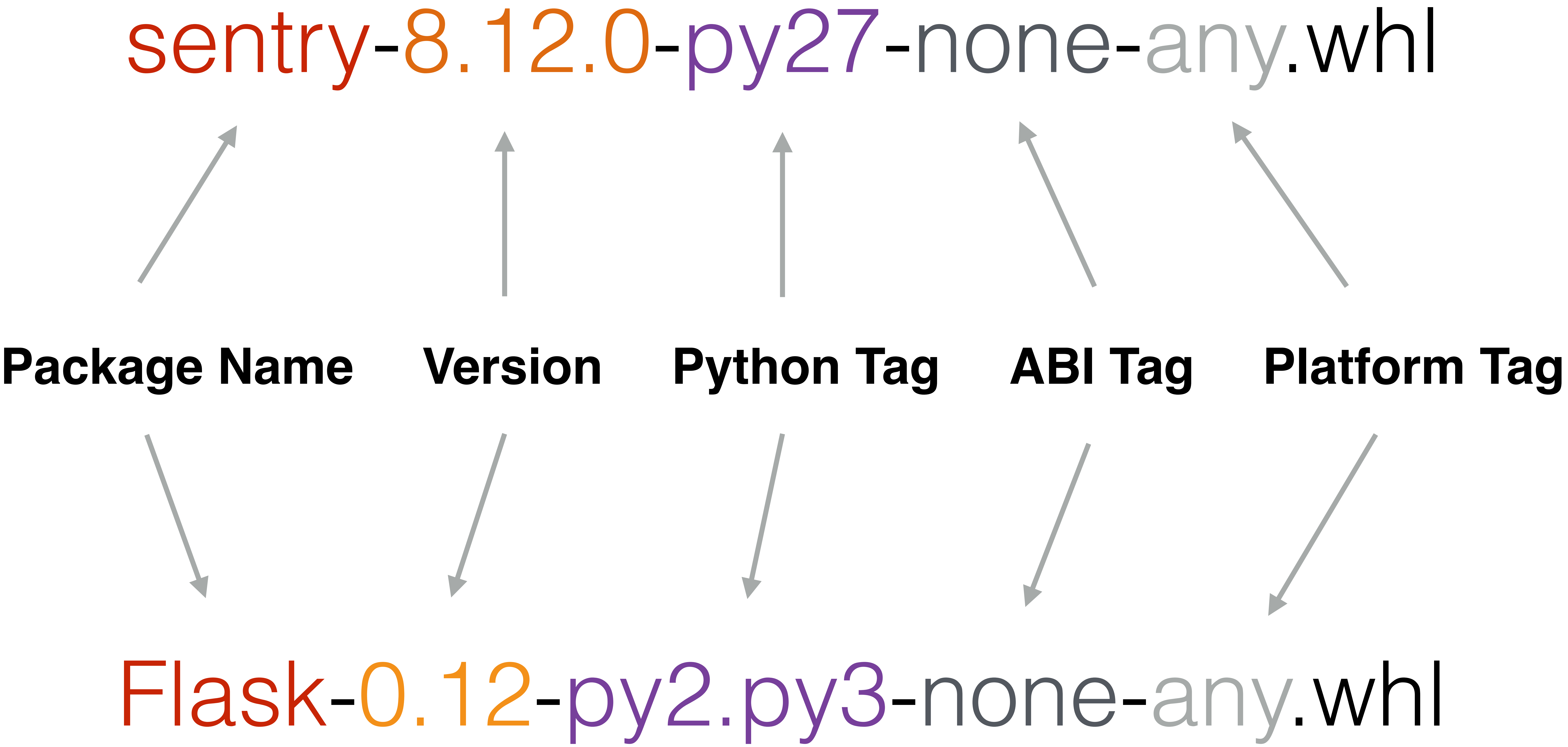
Run “everywhere”

that rules out most already existing solutions. SAD

DISTRIBUTING

Python Wheels

- ★ .py files are portable
- ★ .pyc files are generated on install
- ★ wheel is largely universal



sentry-8.12.0-py27-none-any.whl

Package Name Version Python Tag ABI Tag Platform Tag

Flask-0.12-py2.py3-none-any.whl

Binaries

★ Platform Specific

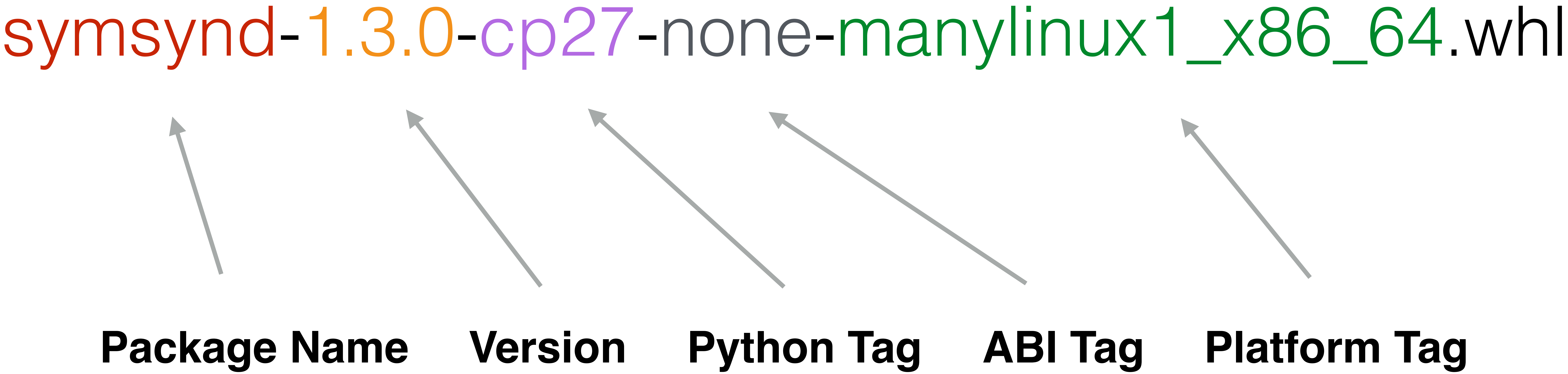
★ libc specific :(

★ might link against system libraries

★ typically cannot compile on installation time

Binary Wheels

- ★ “easy” on OS X
- ★ trivial on Windows
- ★ limited support on Linux (*manylinux1*)



symsynd-1.3.0-cp27-none-manylinux1_x86_64.whl

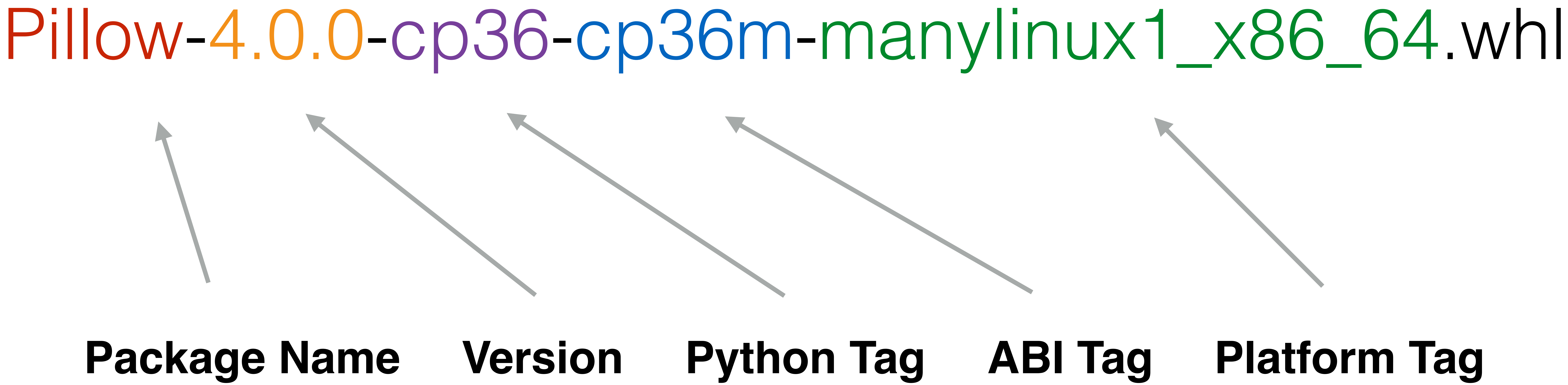
Package Name

Version

Python Tag

ABI Tag

Platform Tag



Pillow-4.0.0-cp36-cp36m-manylinux1_x86_64.whl

Package Name Version Python Tag ABI Tag Platform Tag

ALL THE TAGS

Python Tag

- ★ any python version
- ★ any 2.x / 3.x version
- ★ a specific version
- ★ a specific Python implementation (*cpython*, *pypy*, ...)

ABI Tag

- the Python Interpreter ABI version (eg: UC2 vs UC4)

Platform Tag

- identifies the platform
- eg: 32bit Intel OS X, x86_64
- platform can be complex.
  (eg: *manylinux1_x86_64*)

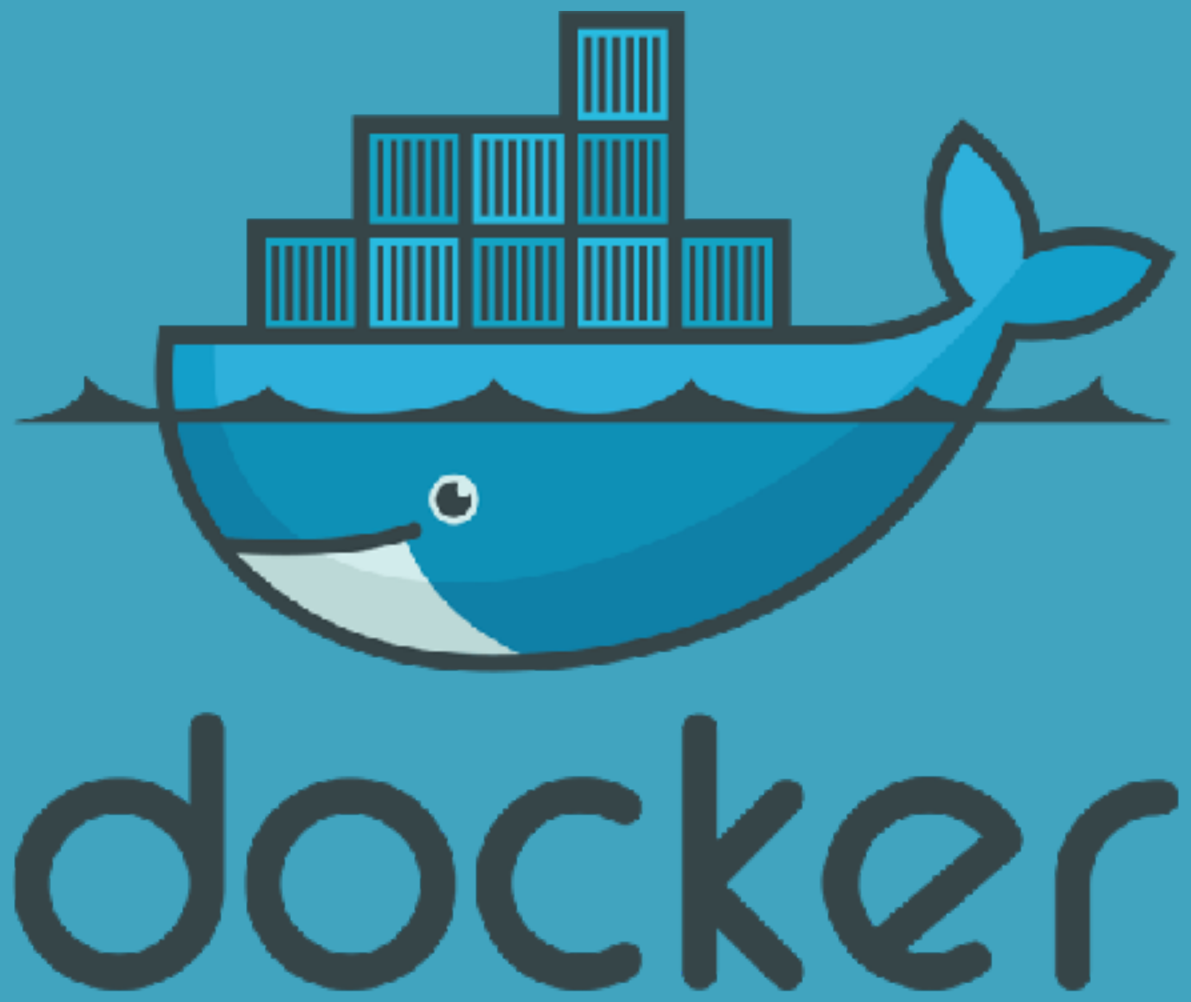
WTF IS MANYLINUX1?

linux binary compatibility
is fraking terrible

manylinux1

- compile on super old CentOS version
- do not link against fancy libraries
- only use old C++ compilers if at all
- static link all the things you can

where to get ancient CentOS?



FEWER DIMENSIONS

Pillow-4.0.0-cp36-cp36m-manylinux1_x86_64.whl

Python 2 builds:

Versions: 2.7
ABI: cpm + cpmu
Platforms: OS X + 2 Linux

Total: 1 × 2 × 3 = **6**

Python 3 builds:

Versions: 3.3 + 3.4 + 3.5 + 3.6 + 3.7
ABI: cpm
Platforms: OS X + 2 Linux

Total: 5 × 1 × 3 = **15**

21 BUILDS!!!

that's a lot of wheels. SAD

Can we kill tags?

- Python version tag: write Python 2.x and 3.x source compatible code
- ABI Tag: do not link against libpython

- Platform Tag: we can't do anything about this one :(

path to success:

- do not link to libpython
- use cffi
- 2.x/3.x compatible sources
- fuck around with setuptools

setuptools

```python
import os
from distutils.command.build_py import build_py
from distutils.command.build_ext import build_ext


PACKAGE = 'mypackage'

class CustomBuildPy(build_py):
    def run(self):
        build_py.run(self)
        build_mylib(os.path.join(self.build_lib, *PACKAGE.split('.')))

class CustomBuildExt(build_ext):
    def run(self):
        build_ext.run(self)
        if self.inplace:
            build_py = self.get_finalized_command('build_py')
            build_mylib(build_py.get_package_dir(PACKAGE))
```

```python
from wheel.bdist_wheel import bdist_wheel

class CustomBdistWheel(bdist_wheel):
    def get_tag(self):
        rv = bdist_wheel.get_tag(self)
        return ('py2.py3', 'none') + rv[2:]
```

```
from setuptools import setup


setup(
    ...
    cffi_modules=['build.py:my_ffi'],
    install_requires=['cffi>=1.0.0'],
    setup_requires=['cffi>=1.0.0'],
    cmdclass={
        'build_ext': CustomBuildExt,
        'build_py': CustomBuildPy,
        'bdist_wheel': CustomBdistWheel,
    }
)
```

BUILD MY LIB

```python
import os
import sys
import shutil
import subprocess

EXT = sys.platform == 'darwin' and '.dylib' or '.so'

def build_mylib(base_path):
    lib_path = os.path.join(base_path, '_nativelib.so')
    here = os.path.abspath(os.path.dirname(__file__))
    cmdline = ['make', 'build-ext']
    rv = subprocess.Popen(cmdline, cwd=here).wait()
    if rv != 0:
        sys.exit(rv)
    src_path = os.path.join(here, 'target', 'release', 'libnativelib' + EXT)
    if os.path.isfile(src_path):
        shutil.copy2(src_path, lib_path)
```

build command

build output path

```
develop:
	pip install --editable . -v

build-ext:
	cargo build --release
```

CFFI (*build.py*)

header only

```
import sys
import subprocess
from cffi import FFI

def _to_source(x):
    if sys.version_info >= (3, 0) and isinstance(x, bytes):
        x = x.decode('utf-8')
    return x

my_ffi = FFI()
my_ffi.cdef(_to_source(subprocess.Popen([
    'cc', '-E', '-DPYTHON_HEADER',
    'mynativelib/mynativelib.h'],
    stdout=subprocess.PIPE).communicate()[0]))
my_ffi.set_source('mypackage._nativelib', None)
```

good for typedefs

with source compilation

```
my_ffi = FFI()
my_ffi.cdef(_to_source(subprocess.Popen([
    'cc', '-E', '-DPYTHON_HEADER',
    'mynativelib/mynativelib.h'],
    stdout=subprocess.PIPE).communicate()[0]))

with open('mynativelib/mynativelib.cpp', 'rb') as source:
    my_ffi.set_source(
        'mypackage/_nativelib',
        _to_source(source.read()),
        include_dirs=['mynativelib'],
        extra_compile_args=['-std=c++11'],
        source_extension='.cpp'
    )
```

GITIGNORE

```
mypackage/_nativelib.py
mypackage/*.so
mypackage/*.dylib
build
dist
*.pyc
*.egg-info
```

WRAPPING WITH CFFI

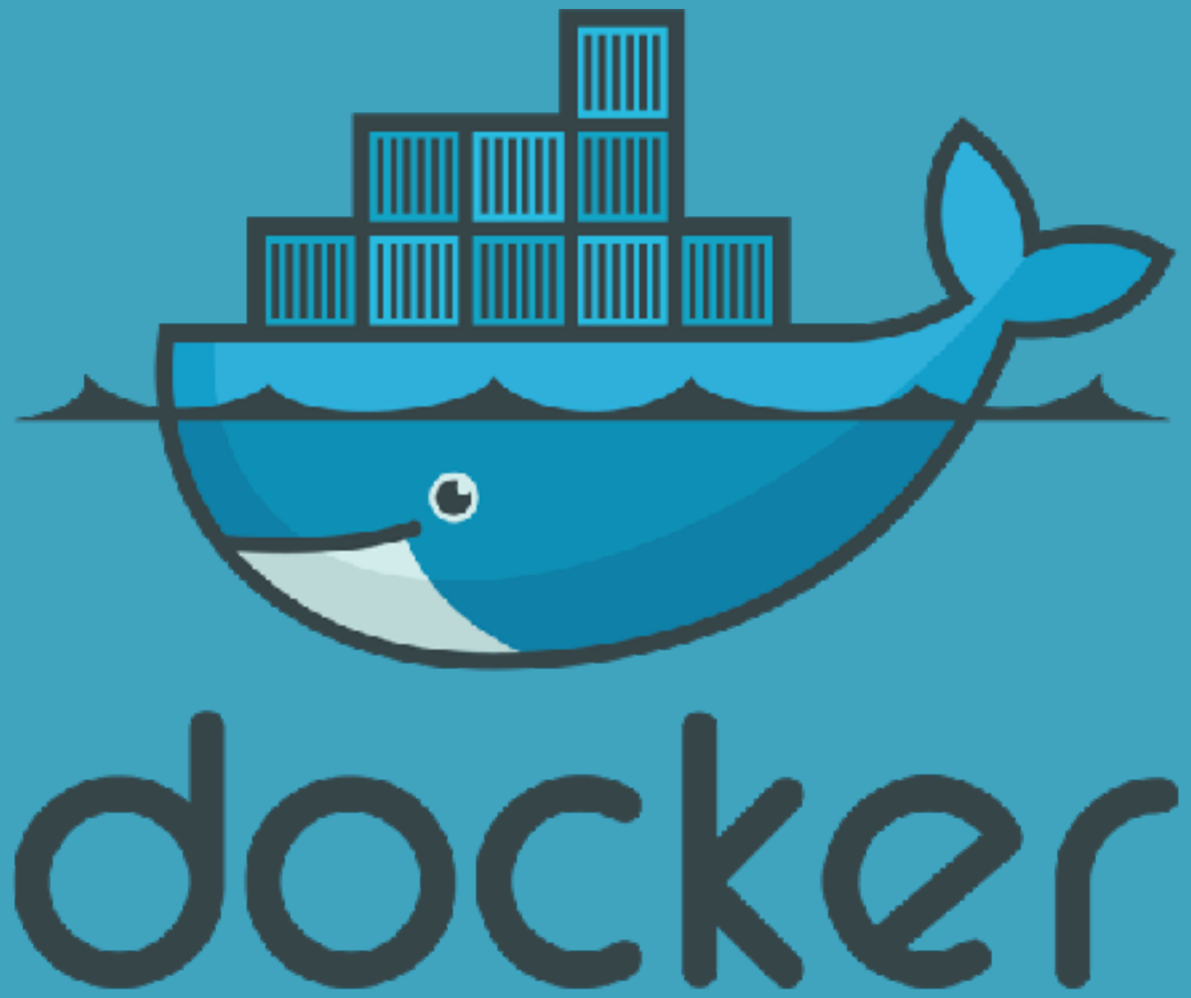
```python
from ._nativelib import ffi as _ffi

_lib = _ffi.dlopen(os.path.join(
    os.path.dirname(__file__), '_nativelib.so'))
_lib.mylib_global_init_if_needed()


class MyObject(object):

    def __init__(self):
        self._ptr = _lib.my_object_new()

    def __del__(self):
        if self._ptr:
            _lib.my_object_free(self._ptr)
            self._ptr = None
```

```python
from ._nativelib import ffi as _ffi, lib as _lib

_lib.mylib_global_init_if_needed()


class MyObject(object):

    def __init__(self):
        self._ptr = _lib.my_object_new()

    def __del__(self):
        if self._ptr:
            _lib.my_object_free(self._ptr)
            self._ptr = None
```

now for building. SO SAD

BASICS

```
$ pip install wheel
$ python setup.py bdist_wheel
```



docker

USEFUL IMAGES

For Python in General

- quay.io/pypa/manylinux1_i686
- quay.io/pypa/manylinux1_x86_64

Things of note

- It's an ancient CentOS (*for instance it has no SNI Support*)
- 32bit builds on on 64bit Docker typically. Use the `linux32` command
- Dockerfile allows you to "cache" steps

How we do it

- travis all the things
- upload artifacts to github releases
- download from there an upload to pypi with twine

what about macOS?

build on travis / locally

- ★ travis better because you can build on old macOS for higher portability
- ★ you can find old SDKs on github!
- ★ Use `MACOS_DEPLOYMENT_TARGET`

```
WHEEL_OPTIONS=
if [ `uname` == "Darwin" ]; then
  WHEEL_OPTIONS="--plat-name=macosx-10.10-intel"
fi
python setup.py bdist_wheel $WHEEL_OPTIONS
```

PATTERNS

Library Design

```c
#ifndef MYLIB_H_INCLUDED
#define MYLIB_H_INCLUDED

#ifdef __cplusplus
extern "C" {
#endif

typedef void mylib_type_t;

mylib_type_t *mylib_type_new(void);
void mylib_type_free(mylib_type_t *self);

#ifdef __cplusplus
}
#endif

#endif
```

```
#include "mylib.h"

class Type {
    Type();
    ~Type();
};

mylib_type_t *mylib_type_new()
{
    Type *rv = new Type();
    (mylib_type_t *)rv;
}

void mylib_type_free(mylib_type_t *self)
{
    if (self) {
        Type *t = (Type *)self;
        delete t;
    }
}
```

Error Handling

```c
typedef struct mylib_error_t {
    int code;
    char *msg;
};

void mylib_error_free(mylib_error_t *err)
{
    if (err) {
        free(err->msg);
        free(err);
    }
}
```

```c
int mylib_do_stuff(int a, int b, mylib_error_t **err_out)
{
    if (a + b > 255) {
        mylib_error_t *err = malloc(mylib_error_t);
        err->msg = strdup("Adding those chars overflows");
        err->code = MYLIB_CHAR_OVERFLOW;
        *err_out = err;
        return -1;
    }

    return a + b;
}
```

```python
special_errors = {}


def invoke_with_exc(func, *args):
    err = _ffi.new('mylib_error_t **')
    try:
        rv = func(*(args + (err,)))
        if not err[0]:
            return rv
        cls = special_errors.get(err[0].code, RuntimeError)
        raise cls(_ffi.string(err[0].msg).decode('utf-8', 'replace'))
    finally:
        if err[0]:
            _lib.mylib_error_free(err[0])
```

```
try:
    rv = invoke_with_exc(_lib.mylib_do_stuff, arg1, arg2)
except DefaultError as e:
    print 'An error happened: %s' % e
else:
    print 'The result is %r' % rv
```

CONCLUSIONS

how painful is it?

it's pretty bad. SAD

but when it works it keeps working. LOVE IT

what do we use it for?

C/C++

Native Symbolication

Rust

Javascript Source Maps

Q&A

```python
def rustcall(func, *args):
    err = _ffi.new('lsm_error_t *')
    rv = func(*(args + (err,)))
    if not err[0].failed:
        return rv
    try:
        cls = special_errors.get(err[0].code, SourceMapError)
        exc = cls(_ffi.string(err[0].message).decode('utf-8', 'replace'))
    finally:
        _lib.lsm_buffer_free(err[0].message)
    raise exc
```

```rust
use std::mem;
use std::panic;

fn silent_panic_handler(_pi: &panic::PanicInfo) {
    /* don't do anything here */
}

#[no_mangle]
pub unsafe extern "C" fn mylib_init() {
    panic::set_hook(Box::new(silent_panic_handler));
}
```

```rust
unsafe fn set_err(err: Error, err_out: *mut CError) {
    if err_out.is_null() {
        return;
    }
    let s = format!("{}\x00", err);
    (*err_out).message = Box::into_raw(s.into_boxed_str()) as *mut u8;
    (*err_out).code = err.get_error_code();
    (*err_out).failed = 1;
}
```

```
unsafe fn landingpad<F: FnOnce() -> Result<T> + panic::UnwindSafe, T>(
    f: F, err_out: *mut CError) -> T
{
    if let Ok(rv) = panic::catch_unwind(f) {
        rv.map_err(|err| set_err(err, err_out)).unwrap_or(mem::zeroed())
    } else {
        set_err(ErrorKind::InternalError.into(), err_out);
        mem::zeroed()
    }
}
```

```
macro_rules! export (
    ($n:ident($($an:ident: $aty:ty),*) -> Result<$rv:ty> $body:block) => (
        #[no_mangle]
        pub unsafe extern "C" fn $n($($an: $aty,)* err: *mut CError) -> $rv
        {
            landingpad(|| $body, err)
        }
    );
);
```

```
export!(lsm_view_dump_memdb(
    view: *mut View, len_out: *mut c_uint, with_source_contents: c_int,
    with_names: c_int) -> Result<*mut u8>
{
    let memdb = (*view).dump_memdb(DumpOptions {
        with_source_contents: with_source_contents != 0,
        with_names: with_names != 0,
    })?;
    *len_out = memdb.len() as c_uint;
    Ok(Box::into_raw(memdb.into_boxed_slice()) as *mut u8)
});
```