The Black Magic of Python Wheels

Elana Hashman
Two Sigma
PyGotham 2018 – New York, NY

@ehashdn

Wheels/Black Magic FAQ

Q: But I’m not a witch?!

A: *Sometimes the greater good requires a little sacrifice.*

Topics

- Python packaging and distribution
- ELF (Executable and Linkable Format) files
- Dynamic vs. static linking

Outline

- A brief history of Python packaging and distribution
- An overview of the wheel
- Why we need native extensions
- How do native extensions even work, really?
  - What are `manylinux` and `auditwheel` for?
- How you can get involved

A Brief History of Python Packaging: Eggs

- Organically adopted (no guiding PEP)
- No standard → many incompatible implementations
- Designed to be directly importable, could include compiled Python (`.pyc` files)

Python Packaging Reinvented: The Wheel

- Adopted via PEP 427
- Follows the PEP 376 standard for distributions and PEP 426 standard for package metadata
- Designed for distrubution, cannot include `.pyc` files (but may include other pre-compiled resources)

Wheels “make it easier to roll out” Python

- Pure wheels
  - Only contain Python code
  - May target a specific version of Python
- Universal wheels
  - Python 2/3 compatible pure wheels
- Extension wheels
  - Allow for

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

Extensions without binary distributions

```
$ pip install --no-binary :all: cryptography
...
    c/_cffi_backend.c:2:10: fatal error: Python.h:
No such file or directory
     #include <Python.h>
              ^~~~~~~~~~
    compilation terminated.
    error: command 'x86_64-linux-gnu-gcc' failed
with exit status 1
$ sudo apt install python-dev  # get Python.h
```

Extensions without binary distributions

```
$ pip install --no-binary :all: cryptography
...
    c/_cffi_backend.c:15:10: fatal error: ffi.h: No
such file or directory
     #include <ffi.h>
              ^~~~~~~
    compilation terminated.
    error: command 'x86_64-linux-gnu-gcc' failed
with exit status 1
$ sudo apt install libffi-dev  # get ffi.h
```

Extensions without binary distributions

```
$ pip install --no-binary :all: cryptography

...
   build/temp.linux-x86_64-2.7/_openssl.c:498:10:
fatal error: openssl/opensslv.h: No such file or
directory
     #include <openssl/opensslv.h>
              ^~~~~~~~~~~~~~~~~~~~
    compilation terminated.
    error: command 'x86_64-linux-gnu-gcc' failed
with exit status 1

$ sudo apt install libssl-dev  # get opensslv.h
```

Extensions without binary distributions

```
$ time pip install --no-binary :all: cryptography

Successfully installed asn1crypto-0.24.0 cffi-
1.11.5 cryptography-2.3.1 enum34-1.1.6 idna-2.7
ipaddress-1.0.22 pycparser-2.19 six-1.11.0


real 0m16.369s
user 0m15.823s
sys  0m0.627s
```

Extensions with binary distributions

```
$ time pip install cryptography  # prebuilt binary

Successfully installed asn1crypto-0.24.0 cffi-
1.11.5 cryptography-2.3.1 enum34-1.1.6 idna-2.7
ipaddress-1.0.22 pycparser-2.19 six-1.11.0


real 0m1.088s
user 0m0.980s
sys  0m0.108s
```

What sort of black magic is this? ✨🔮

`cryptography` is a Python Native Extension

- **Native:** the code was compiled specifically for my operating system
- **Extension:** this library extends Python's functionality with non-Python code
- It uses CFFI: the "C Foreign Function Interface" for Python

Python code is not just Python.

For Python to harness its full potential, it must be able to depend on C libraries.

C is a compiled language

```c
// hello.c

#include<stdio.h>

int main(void) {
   puts("hello
         world");
}
```

```
# a.out (hexadecimal)
0000000 7f45 4c46 0201 0100
0000008 0000 0000 0000 0000
0000010 0300 3e00 0100 0000
0000018 5005 0000 0000 0000
0000020 4000 0000 0000 0000
...
```

ELF File

`gcc hello.c`

`hexdump a.out`

Hexes and ELFs

```
$ readelf -a a.out
ELF Header:
  Magic:    7f 45 4c 46 02 01 01 00
            00 00 00 00 00 00 00 00
  Class:                ELF64
  Data:                 2's complement, little endian
  Version:              1 (current)
  OS/ABI:               UNIX - System V
  ABI Version:          0
  Type:                 DYN (Shared object file)
  Machine:              Advanced Micro Devices X86-64
...
```

Hexes and ELFs

```
$ readelf -a a.out
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

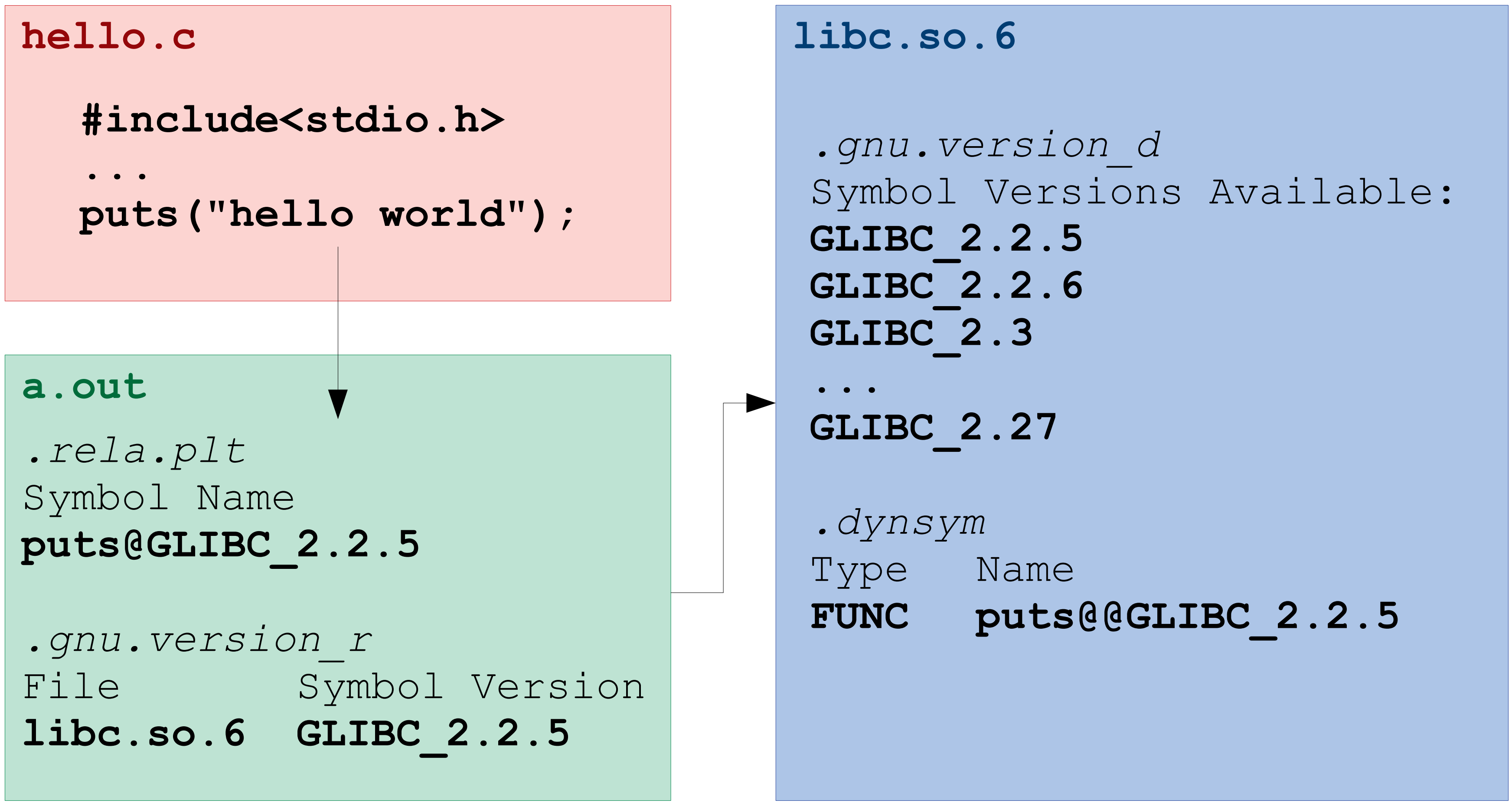
Hexes and ELFs

```
$ readelf -a a.out
...
Relocation section '.rela.plt' at offset
0x4d0 contains 1 entry:
  Offset          Info           Type
000000200fd0  000200000007 R_X86_64_JUMP_SLO
  Sym. Value     Sym. Name + Addend
0000000000000000 puts@GLIBC_2.2.5 + 0
```

Hexes and ELFs

```
$ readelf -a a.out
...
Version needs section '.gnu.version_r' contains
1 entry:
 Addr: 0x00000000000003f0  Offset: 0x0003f0  Link: 6
(.dynstr)
  000000: Version: 1  File: libc.so.6  Cnt: 1
  0x0010:   Name: GLIBC_2.2.5  Flags: none  Version: 2
```

Symbol Versions in Action



What happens when we run this?

- OS parses "magic ELF" text
- OS invokes the ELF interpreter specified by the binary
- ELF interpreter loads any required files with valid versions
- ELF interpreter relocates the program code and dependencies in memory so that it can run
- This is called *dynamic linking*

Q: That all sounds really complex. Couldn’t I just include all the code I need in my output binary?

A: Sure! That’s called *static linking*.

Q: ...then why doesn’t everyone just do that??

Dynamic vs. Static Linking

- **Pros: Dynamic**
  - Less storage space used
  - One copy of a library = one upgrade
- **Cons: Dynamic**
  - Complex
  - Needs some kind of dependency management
- **Cons: Static**
  - More storage space used
  - May store many copies of one library
- **Pros: Static**
  - Simple
  - Dependencies are bundled with your programs

Conclusion: Static linking is great, but should be used sparingly.

…so…

What if we "used it sparingly" to build Python extensions for easier distribution?

But that might not work! What if my C standard library is too old to run your binary?

...so...

What if we made sure to statically link against symbol versions that are maximally compatible?

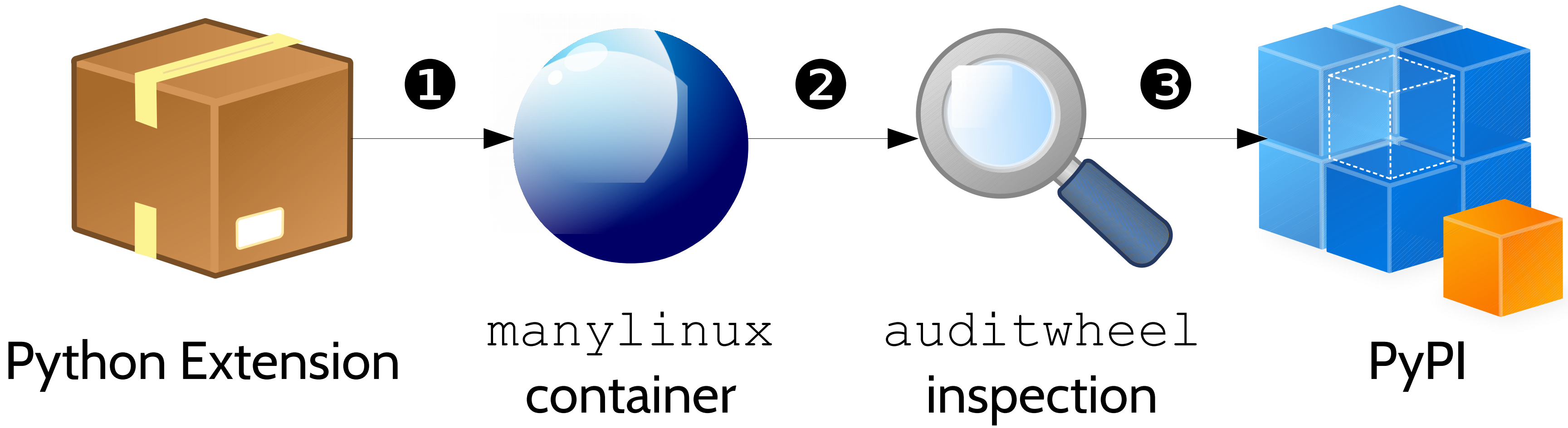
Q: How can we ship compiled Python extensions compatible with as many systems as possible?

A: Static linking (manylinux) and symbol versioning (auditwheel).

What are `manylinux` and `auditwheel`?

- PEPs 513 and 571 define a set of permitted libraries and their symbol versions
  - "Many" Linux systems are compatible with this standard
- `manylinux` is an ancient CentOS Docker image that implements the policy
- `auditwheel` is a tool to enforce the symbol policies

Wheel Builder’s Pipeline



❶ Add your code, dependencies to the `manylinux` Docker image

❷ Inspect the built wheel with `auditwheel` for compliance

❸ Upload to PyPI!

Want in on the magic?

- Help us build wheels!
  - Feedback enthusiastically welcomed ✨
- pythonwheels.com
  - See what packages already build wheels
  - Find examples for how to build yours
- github.com/pypa/python-manylinux-demo
  - Simple demo to learn wheelbuilding


Questions?

Thanks to:

Two Sigma
Nelson Elhage, Paul Kehrer

Talk resources: **https://hashman.ca/pygotham-2018**

Image License Information

- **Tree Cat Silhouette Sunset:** Public Domain (CC0) @besshamiti **https://plixs.com/photo/3297/tree-cat-silhouette-sunset**
- **Happy Halloween!** (Costume Dog): Public Domain (CC0) @milkyfactory **https://flic.kr/p/ArW1N9**