PiterPy 2023

Python — побег из террариума: проблемы переносимости и их решения

Арсений Сапелкин

KasperskyOS

О нас и наших проектах

- Делаем инструменты разработчика в KasperskyOS SDK
- Почти всё пишем на python
- Есть cli с широким функционалом (много зависимостей)
- Есть переиспользуемые в других проектах Python пакеты
- Поставляемся в составе deb пакета (пока что)

Дисклеймеры

- 18+ Уберите детей и впечатлительных от экранов
- Говорим только про *nix
- Буду рад услышать альтернативные точки зрения*

*любые, за исключением "переписать всё на go".

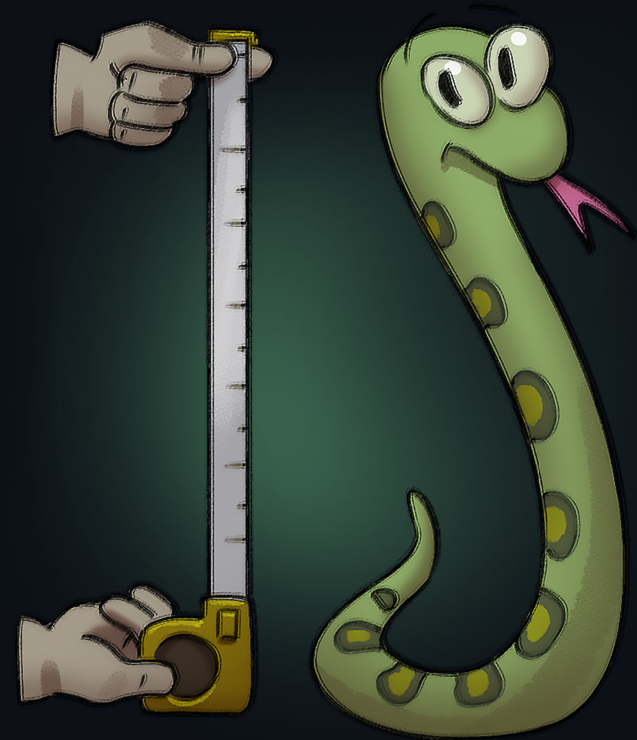
Наши требования

Обязательно:

- переносимость
- разворачивание без pip
- совместимость с докером

Было бы неплохо:

- поставка в виде одного файла
- поставка без установщика
- обработка зависимостей
  (возможно даже включая сам python)



Зачем это всё? Почему python?

- Часть нашего тулинга всегда была на python
- Python очень распространён среди разработчиков и тестировщиков KasperskyOS
- Проще искать C++ разработчиков, знающих python вторым языком

(вместо картинки с троллейбусом)

Структура доклада

- Пройдемся по разным тулам и подходам
- Почему всё это может катастрофически не работать?
- Обсудим, как чинить проблемы

Обозначения

Sys deps	Возможность поставки системных зависимостей
Py deps	Автоматическая рекурсивная py обработка зависимостей
Docker	Нормальная работоспособность в докере без доп настроек
python	Возможность поставки самого python
W/o installer	Поставка без необходимости запуска установщика
Performance	Запуск без потерь в скорости

✅ - полностью удовлетворяет, ❌ - не удовлетворяет,

❌✅ - частично, 🖐🖐 - возможно, но надо делать руками

Пациент

- зависит от pycurl, psycopg2, click
- есть своё C расширение
- не делает ничего осмысленного

```
-> % python3 -m myapp.myapp run
psycopg2 works!
pycurl works!
myextension works!
```

Ссылка на github

(внимание, спойлеры!)

Pex (Python EXecutable)

pex ⭐2.4k

- На выходе формат, описанный в PEP 441
- Автоматически обрабатывает зависимости
- Можно собрать под несколько платформ

```
-> % cat myapp.pex
#!/usr/bin/env python3.11
PK^C^D^@!^K^@^@^@.bootstrap/^C^@^@^@^@
^@^@^@^O^@^@^@.bootstrap/pex/^@<BD>
...
```

Можно добавить всё окружение

```
-> % pex $(pip freeze) -o my_virtualenv.pex
-> % deactivate
-> % ./my_virtualenv.pex
Python 3.11.5
(InteractiveConsole)
>>> import ....
```

Можно добавить наш пакет

```
-> % pex . -c myapp -o myapp.pex
-> % ./myapp.pex run
psycopg2 works!
pycurl works!
myextension works!
```

```
Benchmark 1: python3 -m myapp.myapp --help
  Time (mean ± σ):      63.0 ms ±   3.4 ms    [User: 54.5 ms, System: 8.4 ms]
  Range (min … max):    59.4 ms …  79.3 ms    35 runs

Benchmark 2: myapp.pex --help
  Time (mean ± σ):     720.4 ms ±  17.8 ms    [User: 642.4 ms, System: 75.8 ms]
  Range (min … max):   705.7 ms … 769.4 ms    10 runs

Summary
  python3 -m myapp.myapp --help ran
   11.43 ± 0.69 times faster than myapp.pex --help
```

Пробуем Pex

```
-> % pex . -c myapp -o myapp.pex
-> % ./myapp.pex run
psycopg2 works!
pycurl works!
myextension works!
```

Попробуем запустить на другой машине:

```
-> % docker run ubuntu_with_python:latest ./myapp.pex run
```

```
File "/root/.pex/installed_wheels/53...b9/
psycopg2-2.9.7-cp311-cp311-linux_x86_64.whl/psycopg2/__init__.py", line 51, in <module>
   from psycopg2._psycopg import (                              # noqa
ImportError: libpq.so.5: cannot open shared object file: No such file or directory
```

	Pex
Sys deps	❌
Py deps	✅
Docker	✅
python	❌
W/o installer	✅
Performance	❌

Хорошо подходит если вы контролируете окружение и не требовательны к скорости запуска.

Пакуемся в deb пакет

spotify/dh-virtualenv ⭐1.6k

- Комбинация virtualenv и deb пакета
- Позволяет прописать зависимости от системных пакетов

debian/rules

```
%:
	dh $@ --with python-virtualenv

override_dh_virtualenv:
	dh_virtualenv --setuptools --python /usr/bin/python3
```

debian/control

```
...
Build-Depends: python3-dev, python3-setuptools, python3-pip, dh-virtualenv
...
Package: myapp
Depends: ${shlibs:Depends}, ${misc:Depends}, libcurl4-openssl-dev, libpq-dev, python3 (>= 3.8)
```

```
-> % dpkg-buildpackage -us -uc -b
```

Возможность прописывать системные зависимости

```
-> % dpkg-deb -I myapp_0.1-1_amd64.deb
...
 Depends: libc6 (>= 2.28), libcurl4 (>= 7.56.1),
  libexpat1 (>= 2.1~beta3), libpq5 (>= 10~~),
  libssl1.1 (>= 1.1.0), zlib1g (>= 1:1.2.0),
   libcurl4-openssl-dev, libssl-dev, libpq-dev
...

-> % dpkg -i myapp_0.1-1_amd64.deb
-> % /opt/venvs/myapp/bin/myapp run
psycopg2 works!
pycurl works!
myextension works!
```

	dh-virtualenv
Sys deps	✅ (deb only)
Py deps	✅
Docker	✅ (deb only)
python	🖐🖐
W/o installer	❌
Performance	✅

Хорошо подойдёт если вас интересуют только deb системы.

AppImage

- Не требует установщика
- Позволяет привести с собой python и любое другое окружение
- Готовый и хорошо работающий тулинг
- Один файл для всего

Как использовать AppImage с python?

python-appimage ⭐141

Сборка одной командой!

```
-> % python-appimage build app  .
...
-> % ./myapp-x86_64.AppImage run
...
```

А что по скорости запуска?

```
Benchmark 1: python3 -m myapp.myapp --help
  Time (mean ± σ):      69.7 ms ±   2.8 ms    [User: 63.1 ms, System: 6.4 ms]
  Range (min … max):    64.9 ms …  76.8 ms    40 runs

Benchmark 2: ./myapp.AppImage --help
  Time (mean ± σ):     259.9 ms ±   3.7 ms    [User: 76.4 ms, System: 13.6 ms]
  Range (min … max):   255.5 ms … 267.2 ms    11 runs

Summary
  python3 -m myapp.myapp --help ran
    3.73 ± 0.16 times faster than ./myapp.AppImage --help
```

Проблема с Docker

- Appimage не запустится в докере без `--cap-add SYS_ADMIN --device /dev/fuse:mrw --cap-add MKNOD`

```
-> % sudo docker run debian-fuse:latest myapp.AppImage --help
fuse: device not found, try 'modprobe fuse' first
open dir error: No such file or directory
```

	AppImage
Sys deps	🖐 🖐
Py deps	✅
Docker	❌
python	✅
W/o installer	✅
Performance	❌✅

Очень хороши для всего, что не предполагает запуск в докере. Например графические приложения.

Пакуемся в самораспаковывающийся бинарник

Исчерпывающее исследование:
Паковка python-утилит в бинарник, Евгений Пистун

Выводы, кратко:

- Много хороших инструментов
- На данный момент наш фаворит - pyoxidizer ⭐4.9k

Pyoxidizer

- Весьма нетривиален в настройках, зато много возможностей
- Есть немало подводных камней, например `__file__` не работает
- Предоставляет свои переносимые версии Python

```
def make_exe():
    dist = default_python_distribution(python_version = '3.8')
    ...
    exe = dist.to_python_executable(
    ...
    for resource in exe.pip_install(["myapp"]):
        exe.add_python_resource(resource)
    ...

register_target("exe", make_exe)
...
```

```
Benchmark 1: ./pyoxidizer/myapp --help
  Time (mean ± σ):     102.9 ms ±   3.1 ms    [User: 91.3 ms, System: 10.9 ms]
  Range (min … max):    97.7 ms … 109.5 ms    26 runs

Benchmark 2: python3 -m myapp.myapp --help
  Time (mean ± σ):     115.9 ms ±   1.4 ms    [User: 102.0 ms, System: 13.2 ms]
  Range (min … max):   113.1 ms … 118.4 ms    24 runs

Summary
  ./pyoxidizer/myapp --help ran
    1.13 ± 0.04 times faster than python3 -m myapp.myapp --help
```

	Pyoxidizer
Sys deps	🖐️🖐️
Py deps	✅
Docker	✅
python	✅
W/o installer	✅
Performance	✅

Хорошо подойдёт для тех, у кого много свободного времени.

Компилируемся в честный исполняемый файл

- **nuitka** (наш фаворит)
- **codon**
- **cython**
- ...

Python C Api

Переведём этот код на C вручную

```python
import json
from urllib import request
response = request.urlopen('https://api64.ipify.org?format=json')
ip_info = json.load(response)
print(f"Public IP: {ip_info['ip']}")
```

Понадобится Python.h

```
#include <Python.h>
int main()
{
  Py_Initialize();
  ...
}
```

Импортируем модули

```python
import json
from urllib import request
```

```c
PyObject *jsonModule = PyImport_ImportModule("json");
PyObject *requestModule = PyImport_ImportModule("urllib.request");
```

Получим и вызовем нужные нам функции

```
response = request.urlopen('https://api64.ipify.org?format=json')
ip_info = json.load(response)
```

```
PyObject *urlopen = PyObject_GetAttrString(requestModule, "urlopen");
PyObject *args = Py_BuildValue("(s)", "https://api64.ipify.org?format=json");
PyObject *response = PyObject_CallObject(urlopen, args);

PyObject *loadFunc = PyObject_GetAttrString(jsonModule, "load");
PyObject *ip_info = PyObject_CallObject(loadFunc, Py_BuildValue("(O)", response));
```

Вывод результата

Python:

```
print(f"Public IP: {ip_info['ip']}")
```

C:

```
PyObject *pyStr = PyObject_Str(ipString);
const char *str = PyUnicode_AsUTF8(pyStr);
printf("Public IP: %s\n", str);
```

Всё получилось достаточно просто

```
Py_Initialize();
PyObject *jsonModule = PyImport_ImportModule("json");
PyObject *requestModule = PyImport_ImportModule("urllib.request");

PyObject *urlopen = PyObject_GetAttrString(requestModule, "urlopen");
PyObject *args = Py_BuildValue("(s)", "https://api64.ipify.org?format=json");
PyObject *response = PyObject_CallObject(urlopen, args);

PyObject *loadFunc = PyObject_GetAttrString(jsonModule, "load");
PyObject *ip_info = PyObject_CallObject(loadFunc, Py_BuildValue("(O)", response));

PyObject *ipString = PyObject_GetItem(ip_info, Py_BuildValue("s", "ip"));

PyObject *pyStr = PyObject_Str(ipString);
const char *str = PyUnicode_AsUTF8(pyStr);
printf("Public IP: %s\n", str);
```

Честная компиляция

nuitka ⭐10k

- Поддерживает и зависимости тоже
- Идеология одной кнопки "сделай хорошо"
- Можно статически слинковаться с python и не только
- Используется парсер python кода из cpython

```
-> % python -m nuitka --standalone  myapp.py --onefile -o myapp

-> % ldd myapp
linux-vdso.so.1 (0x00007ffe29b2b000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5b5f9ba000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5b5fc0e000)
```

```
Benchmark 1: python3 -m myapp.myapp --help
  Time (mean ± σ):      67.4 ms ±   3.2 ms    [User: 59.7 ms, System: 7.5 ms]
  Range (min … max):    63.3 ms …  82.3 ms    34 runs

Benchmark 2: myapp.nuitka --help
  Time (mean ± σ):     100.8 ms ±   2.0 ms    [User: 95.0 ms, System: 5.8 ms]
  Range (min … max):    98.5 ms … 108.7 ms    28 runs

Summary
  python3 -m myapp.myapp --help ran
    1.50 ± 0.08 times faster than myapp.nuitka --help
```

	Nuitka
Sys deps	✅
Py deps	✅
Docker	✅
python	✅
W/o installer	✅
Performance	✅

Может подойти везде, но:

- Можно нарваться на изменение поведения (e.g. `sys.path`)
- Не теряйте ресурсы и плагины, для чего понадобится приседать
- Осторожнее с лицензиями

Пробуем Nuitka

Собираем в один бинарник

```
python -m nuitka --standalone  myapp.py --onefile -o myapp
```

Проверим что он действительно самодостаточен:

```
-> % docker run -v$PWD:$PWD -w$PWD ubuntu:latest ./myapp run
psycopg2 works!
pycurl works!
myextension works!
```

Класс, вы восхитительны! **В продакшн!**

Первый же клиент...

```
-> % docker run -v$PWD:$PWD -w$PWD ubuntu:20.04 ./myapp run
./myapp: ...libc.so.6: version `GLIBC_2.33' not found (required by ./myapp)
./myapp: ...libc.so.6: version `GLIBC_2.34' not found (required by ./myapp)
```

myapp ссылается на свежий glibc:

```
-> % objdump -T ./myapp | grep GLIBC
...
000...000      DF *UND*  000...000 (GLIBC_2.34) __libc_start_main
...
000...000      DF *UND*  000...000 (GLIBC_2.33) fstat
...
```

glibc на машине клиента:

```
-> % docker run ubuntu:20.04  ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.9) 2.31
```

Упс...

Проблемные пакеты с C расширениями:

- Собранные неправильным тулчейном
- Имеющие внешние зависимости

Беспроблемные:

- Pure python пакеты
- Правильно собранные

Как быть?

- Собираемся со старым glibc
- Внешние зависимости берём с собой

manylinux

- серия тэгов, стандартизирующих совместимость бинарного пакета с версиями linux

Эволюция manylinux

- PEP 425 (2012) - стандартизированы теги платформы для пакетов `{distribution}-{version}(-{build tag})?-{python tag}-{abi tag}-{platform tag}.whl`
- PEP 513 (2016) - тэг manylinux1
- PEP 571 (2018) - тэг manylinux2010
- PEP 599 (2019) - тэг manylinux2014
- PEP 600 (2019) - последняя, более общая система тэгов manylinux_x_y `manylinux_{GLIBCMAJOR}_${GLIBCMINOR}_${ARCH}`

Как читать тэг `manylinux_2_5_x86_64`:

- GLIBC_2.5 - максимально допустимая версия базовых символов glibc
- Жестко фиксированный список допустимых зависимостей, например libgcc_s.so.1, libstdc++.so.6, libpthread.so.0 и еще с десяток других, имеющихся в *практически всех* дистрибутивах с glibc 2.5
- архетиктура x86_64

Читается как "работает на *практически всех* мейнстримовых linux x86_64 дистрибутивов с версией glibc 2.5 и выше"

Минусы manylinux

- Усложнённая сборка
- Большие бинарники (статическая линковка)
- Использование устаревших библиотек (помним security риски)

Есть даже целый проект no-manylinux

Примеры пакетов без тэга manylinux

```
- pycurl

- libvirt

- PyGObject

- pycairo

- netifaces
```

```
- pycrypto

- gssapi

- pycups

- pykerberos

- наш собственный myapp
```

Отлавливаем проблемные пакеты

Формат имени пакета:

{python tag}-{abitag}-{platform tag}.whl

В `pycurl-7.45.2-cp38-cp38-linux_x86_64.whl`,

linux_x86_64 - тэг платформы

Получается простейшая проверка

(ссылка на наш полный скрипт):

```
allowed_tags = ["any", "manylinux1"...
platform_tags = parse_wheel_filename(filename).platform_tags
assert any(tag in allowed_tags for tag in platform_tags)
```

Проверим наше приложение `myapp`:

```
-> % pip3 wheel -r requirements.txt --wheel-dir wheelhouse
-> % whl-tags-checker.py wheelhouse any manylinux2014_x86_64 manylinux_2_5_x86_64
Error: wheel package without supported platform tags was found:
pycurl-7.45.2-cp311-cp311-linux_x86_64.whl.
Error: wheel package without supported platform tags was found:
psycopg2-2.9.7-cp311-cp311-linux_x86_64.whl.
```

Как приготовить свой пакет или починить 3rd-party

Используем готовый docker образ

```
-> % docker run quay.io/pypa/manylinux2010_x86_64
-> % /opt/python/cp38-cp38/bin/pip wheel pycurl -w weelhouse
...
  Created wheel for pycurl:
  filename=pycurl-7.45.2-cp38-cp38-linux_x86_64.whl
...
```

Если у пакета внешние зависимости - добавляем их в колесо

```
-> % auditwheel repair pycurl-7.45.2-cp38-cp38-linux_x86_64.whl
Repairing pycurl-7.45.2-cp38-cp38-linux_x86_64.whl
Previous filename tags: linux_x86_64
New filename tags: manylinux_2_17_x86_64, manylinux....
Previous WHEEL info tags: cp38-cp38-linux_x86_64
New WHEEL info tags: cp38-cp38-manylinux_2_17_x86_64, ...
Fixed-up wheel written to
...pycurl-7.45.2-cp38-cp38-manylinux_2_17_x86_64.ma....
```

Теперь такие пакеты как-то так

```
pip3 install --no-index --find-links ./wheelhouse
```

Важные подитоги:

- правильный платформенный тэг важен независимо от выбранного способа поставки
- любая компиляций должна производится с правильным тулчейном
- переносимость не бесплатна

```
Benchmark 1: myapp.p38.gcc6 --help
  Time (mean ± σ):     316.0 ms ±  21.6 ms    [User: 303.2 ms, System: 11.9 ms]
  Range (min … max):   302.7 ms … 375.3 ms    10 runs

Benchmark 2: myapp.p311.gcc13 --help
  Time (mean ± σ):     172.2 ms ±   2.9 ms    [User: 163.9 ms, System: 8.3 ms]
  Range (min … max):   168.7 ms … 180.2 ms    16 runs

Summary
  myapp.p311.gcc13 ran
    1.83 ± 0.13 times faster than myapp.p38.gcc6 --help
```

Делаем все сами

- Понадобится правильный python
- Берём с собой только правильные пакеты

Берём питона с собой

Откуда взять правильный python?

- python-appimage ★141
- manylinux контейнеры ★1.3k
- python-build-standalone ★1.2k
- Собрать самому

	Всё руками
Sys deps	🖐🖐
Py deps	✅
Docker	✅
Python	🖐🖐
W/o installer	🖐🖐
Performance	✅

Хорошо подходит в случае наличия уже готовой схемы распространиния SDK

Выводы

- Существует много готового тулинга
- Есть набор правил, которые необходимо соблюдать
- За последние 10 лет Python сделал огромный шаг вперёд в этом направлении

Небольшое сравнение

	Pex	dh-venv	AppImage	Pyoxidizer	Nuitka	Руками
Sys deps	❌	✅ (deb only)	🖐🖐	🖐🖐	✅	🖐🖐
Py deps	✅	✅	✅	✅	✅	✅
Docker	✅	✅ (deb only)	❌	✅	✅	✅
python	❌	🖐🖐	✅	✅	✅	🖐🖐
W/o installer	✅	❌	✅	✅	✅	🖐🖐
Performance	❌	✅	❌✅	✅	✅	✅

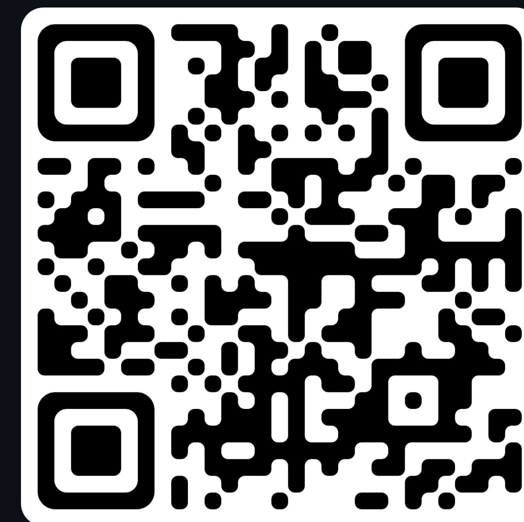
Проект-песочница - ссылка

```
-> % make build
```

```
-> % make portability-test
Success: myapp_nuitka_onefile passed on ubuntu:17.04
Success: myapp_nuitka_onefile passed on ubuntu:20.04
Success: myapp_nuitka_onefile passed on debian:9
Success: myapp_nuitka_onefile passed on opensuse/leap:15.0
... etc ...
```

```
-> % make benchmark
Benchmarking startup time...

Summary
  pyoxidizer/myapp --help ran
    3.21 ± 0.26 times faster than myapp_nuitka_as_folder/myapp --help
    5.72 ± 0.71 times faster than myapp_nuitka_onefile --help
    7.04 ± 1.53 times faster than myapp.AppImage --help
```



Источники

- An Overview of Packaging for Python
- Python Packaging
- How We Deploy Python Code
- DevOps Tool Bazaar
- GLibc backward compatibility
- PEP 425, 513, 571, 599, 600
- Platform compatibility tags
- no-manylinux github
- Nuitka Manual
- Как работают snap, flatpak, appimage
- WTF is PEX?