Клонируйте и добавьте директории

Выберите, куда Вы хотите клонировать Ваш репозиторий или выполните следующую функцию:

```
git clone https://github.com/notebooktoall/notebookc.git
```

Подставьте свою организацию и репозиторий.

Перейдите в папку проекта с помощью десктопного графического интерфейса или редактора кода. Или используйте командную строку с `cd my-project` и после просмотрите файлы с `ls –A`.

Ваши исходные папки и файлы должны выглядеть так:

```
.git
.gitignore
LICENSE
README.rst
```

Создайте вложенную папку для основных файлов проекта. Я советую назвать ее так же, как и вашу библиотеку. Убедитесь, что в имени нет пробелов.

Создайте файл с именем *__init__.py* в основной вложенной папке. Этот файл пока останется пустым. Он необходим для импорта файлов.

Создайте еще один файл с таким же именем, как у основной вложенной папки, и добавьте .py. Мой файл называется *notebookc.py*. Вы можете назвать этот Python-файл как захотите. Пользователи библиотеки при импорте модуля будут ссылаться на имя этого файла.

Содержимое моей директории *notebookc* выглядит следующим образом:

```
.git
.gitignore
LICENSE
README.rst
notebookc/__init__.py
notebookc/notebookc.py
```

Шаг 7: Скачайте и установите requirements_dev.txt

На верхнем уровне директории проекта создайте файл *requirements_dev.txt*. Часто этот файл называют *requirements.txt*. Назвав его *requirements_dev.txt*, Вы показываете, что эти библиотеки могут устанавливаться только разработчиками

проекта.

В файле укажите, что должны быть установлены pip и wheel.

```
pip==19.0.3
wheel==0.33.1
```

Обратите внимание, что мы указываем точные версии библиотек с двойными знаками равенства и полными номерами версии.

Закрепите версии вашей библиотеку в *requirements_dev.txt*

Соавтор, который разветвляет репозиторий проекта и устанавливает закрепленные библиотеки require_dev.txt с помощью pip, будет иметь те же версии библиотеки, что и Вы. Вы знаете, что эта версия будет работать у них. Кроме того, Read The Docs будет использовать этот файл для установки библиотек при сборке документации.

В вашей активированной виртуальной среде установите библиотеку в файл needs_dev.txt с помощью следующей команды:

```
pip install -r requirements_dev.txt
```

Настоятельно рекомендую обновлять эти библиотеки по мере выхода новых версий. На данный момент установите любые последние версии, доступные на PyPi.

В следующей статье расскажу, как установить инструмент, облегчающий этот процесс. Подпишитесь, чтобы не пропустить.

Шаг 8: Поработайте с кодом

В целях демонстрации давайте создадим базовую функцию. Свою собственную крутую функцию сможете создать позже.

Вбейте следующее в Ваш основной файл (для меня это *notebookc/notebookc/notebookc.py*):

```
def convert(my_name):
    """
    Print a line about converting a notebook.
    Args:
```

```
        my_name (str): person's name
    Returns:
        None
    """
```

```
    print(f"I'll convert a notebook for you some day, {my_name}.")
```

Вот наша функция во всей красе.

Строки документа начинаются и заканчиваются тремя последовательными двойными кавычками. Они будут использованы в следующей статье для автоматического создания документации.
Сохраните изменения. Если хотите освежить память о работе с Git, то можете заглянуть в эту статью.

Шаг 9: Создайте setup.py

Файл *setup.py* — это скрипт сборки для вашей библиотеки. Функция setup из Setuptools создаст библиотеку для загрузки в PyPI. Setuptools содержит информацию о вашей библиотеке, номере версии и о том, какие другие библиотеки требуются для пользователей.

Вот мой пример файла setup.py:

```
from setuptools import setup, find_packages
```

```
with open("README.md", "r") as readme_file:
    readme = readme_file.read()
```

```
requirements = ["ipython>=6", "nbformat>=4", "nbconvert>=5", "requests>=2"]
```

```
setup(
    name="notebookc",
    version="0.0.1",
```

```
    author="Jeff Hale",
    author_email="jeffmshale@gmail.com",
    description="A package to convert your Jupyter Notebook",
    long_description=readme,
    long_description_content_type="text/markdown",
    url="https://github.com/your_package/homepage/",
    packages=find_packages(),
    install_requires=requirements,
    classifiers=[
        "Programming Language :: Python :: 3.7",
        "License :: OSI Approved :: GNU General Public License v3 (GPLv3)",
    ],
)
```

Обратите внимание, что *long_description* установлен на содержимое файла README.md. Список требований (requirements), указанный в *setuptools.setup.install_requires*, включает в себя все необходимые зависимости для работы вашей библиотеки.

В отличие от списка библиотек, требуемых для разработки в файле require_dev.txt, этот список должен быть максимально разрешающим. Узнайте почему здесь.

Ограничьте список install_requires только тем, что Вам надо — Вам не нужно, чтобы пользователи устанавливали лишние библиотеки. Обратите внимание, что необходимо только перечислить те библиотеки, которые не являются частью стандартной библиотеки Python. У Вашего пользователя и так будет установлен Python, если он будет использовать вашу библиотеку.
Наша библиотека не требует никаких внешних зависимостей, поэтому Вы можете исключить четыре библиотеки, перечисленных в примере выше.

Соавтор, который разветвляет репозиторий проекта и устанавливает закрепленные библиотеки с помощью pip, будет иметь те же версии, что и Вы. Это значит, что они должны работать.

Измените информацию setuptools так, чтобы она соответствовала информации вашей библиотеки. Существует множество других необязательных аргументов и классификаторов ключевых слов — см. перечень здесь. Более подробные руководства по setup.py можно найти здесь и здесь.

Сохраните свой код в локальном репозитории Git. Пора переходить к созданию библиотеки!

Шаг 10: Соберите первую версию

Twine — это набор утилит для безопасной публикации библиотек Python на PyPI. Добавьте библиотеку Twine в следующую пустую строку файла *require_dev.txt* таким образом:

```
twine==1.13.0
```

Затем закрепите Twine в Вашей виртуальной среде, переустановив библиотеки needs_dev.txt.

```
pip install -r requirements_dev.txt
```

Затем выполните следующую команду, чтобы создать файлы библиотеки:

```
python setup.py sdist bdist_wheel
```

Необходимо создать несколько скрытых папок: dist, build и — в моем случае — notebookc.egg-info. Давайте посмотрим на файлы в папке dist. Файл .whl — это файл Wheel — встроенный дистрибутив. Файл .tar.gz является исходным архивом.

На компьютере пользователя pip будет по мере возможности устанавливать библиотеки как wheels/колеса. Они устанавливаются быстрее. Когда pip не может этого сделать, он возвращается к исходному архиву.

Давайте подготовимся к загрузке нашего колеса и исходного архива.

Шаг 11: Создайте учётную запись TestPyPI

PyPI — каталог библиотек Python (Python Package Index). Это официальный менеджер библиотек Python. Если файлы не установлены локально, pip получает их оттуда.

TestPyPI — это работающая тестовая версия PyPI. Создайте здесь учетную запись TestPyPI и подтвердите адрес электронной почты. Обратите внимание, что у Вас должны быть отдельные пароли для загрузки на тестовый сайт и официальный сайт.

Шаг 12: Опубликуйте библиотеку в PyPI

Используйте Twine для безопасной публикации вашей библиотеки в TestPyPI. Введите следующую команду — никаких изменений не требуется.

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Вам будет предложено ввести имя пользователя и пароль. Не забывайте, что TestPyPI и PyPI имеют разные пароли!

При необходимости исправьте все ошибки, создайте новый номер версии в файле setup.py и удалите старые артефакты сборки: папки build, dist и egg. Перестройте задачу с помощью `python setup.py sdist bdist_wheel` и повторно загрузите с помощью Twine. Наличие номеров версий в TestPyPI, которые ничего не значат, особой роли не играют — Вы единственный, кто будет использовать эти версии библиотек.

После того, как Вы успешно загрузили свою библиотеку, давайте удостоверимся, что Вы можете установить его и использовать.

Шаг 13: Проверьте и используйте установленную библиотеку

Создайте еще одну вкладку в командном интерпретаторе и запустите другую виртуальную среду.

```
python3.7 -m venv my_env
```

Активируйте ее.

```
source my_env/bin/activate
```

Если Вы уже загрузили свою библиотеку на официальный сайт PyPI, то сможете выполнить команду `pip install your-package`. Мы можем извлечь библиотеку из TestPyPI и установить его с помощью измененной команды.

Вот официальные инструкции по установке вашей библиотеки из TestPyPI:

Вы можете заставить pip загружать библиотеки из TestPyPI вместо PyPI, указав это в index-url.

```
pip install --index-url https://test.pypi.org/simple/ my_package
```

Если хотите, чтобы pip также извлекал и другие библиотеки из PyPI, Вы можете добавить — extra-index-url для указания на PyPI. Это полезно, когда тестируемая библиотека имеет зависимости:

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url http
```

Если у вашей библиотеки есть зависимости, используйте вторую команду и подставьте имя вашей библиотеки.

Вы должны увидеть последнюю версию библиотеки, установленного в Вашей виртуальной среде.

Чтобы убедиться, что Вы можете использовать свою библиотеку, запустите сеанс IPython в терминале следующим образом:

```
python
```

Импортируйте свою функцию и вызовите ее со строковым аргументом. Вот как выглядит мой код:

```
from notebookc.notebookc import convert
```

```
convert("Jeff")
```

После я получаю следующий вывод:

```
I'll convert a notebook for you some day, Jeff.
```

(Когда-нибудь я конвертирую для тебя блокнот, Джефф)

Я в Вас верю.

Шаг 14: Залейте код на PyPI

Залейте Ваш код на настоящий сайт PyPI, чтобы люди могли скачать его с помощью `pip install my_package`.

Загрузить код можно так:

```
twine upload dist/*
```

Обратите внимание, что Вам нужно обновить номер версии в setup.py, если Вы хотите залить новую версию в PyPI.

Отлично, теперь давайте загрузим нашу работу на GitHub.

Шаг 15: Залейте библиотеку на GitHub

Убедитесь, что Ваш код сохранен.

Моя папка проекта notebookc выглядит так:

`.git`

`.gitignore`

`LICENSE`

`README.md`

`requirements_dev.txt`

`setup.py`

`notebookc/__init__.py`

`notebookc/notebookc.py`

Исключите любые виртуальные среды, которые Вы не хотите загружать. Файл Python .gitignore, который мы выбрали при создании репозитория, не должен допускать индексации артефактов сборки. Возможно, Вам придется удалить папки виртуальной среды.

Переместите вашу локальную ветку на GitHub с помощью `git push origin my_branch`.

Шаг 16: Создайте и объедините PR

В браузере перейдите к GitHub. У Вас должна появиться опция сделать pull-запрос. Нажимайте на зеленые кнопки, чтобы создать, объединить PR и чтобы убрать удаленную ветку.
Вернувшись в терминал, удалите локальную ветку с `git branch -d my_feature_branch`.

Шаг 17: Обновите рабочую версию на GitHub

Создайте новую версию библиотеки на GitHub, кликнув на релизы на главной странице репозитория. Введите необходимую информацию о релизе и сохраните.

На сегодня достаточно!

Мы научимся добавлять другие файлы и папки в будущих статьях.
А пока давайте повторим шаги, которые мы разобрали.

Итог: 17 шагов к рабочей библиотеке

1. Составьте план.
2. Дайте имя библиотеке.
3. Настройте среду.
4. Создайте организацию в GitHub.
5. Настройте GitHub Repo.
6. Клонируйте и добавьте директории.
7. Скачайте и установите requirements_dev.txt.
8. Поработайте с кодом.
9. Создайте setup.py.
10. Соберите первую версию.
11. Создайте учётную запись TestPyPI.
12. Опубликуйте библиотеку в PyPI.
13. Проверьте и используйте установленную библиотеку.
14. Залейте код на PyPI.
15. Залейте библиотеку на GitHub.
16. Создайте и объедините PR.
17. Обновите рабочую версию на GitHub.

pyqtdeploy, или упаковываем Python-программу в exe'шник… the hard way

- Python,
- Qt
- Из песочницы

Наверняка, каждый, кто хоть раз писал что-то на Python, задумывался о том, как распространять свою программу (или, пусть даже, простой скрипт) без лишней головной боли: без необходимости устанавливать сам интерпретатор, различные зависимости, кроссплатформенно, чтобы одним файлом-exe'шником (на крайний случай, архивом) и минимально возможного размера.

Для этой цели существует немало инструментов: *PyInstaller*, *cx_Freeze*, *py2exe*, *py2app*, *Nuitka* и многие другие… Но что, если вы используете в своей программе *PyQt*? Несмотря на то, что многие (если не все) из выше перечисленных инструментов умеют упаковывать программы, использующие PyQt, существует другой инструмент от разработчиков самого PyQt под названием pyqtdeploy. К моему несчастью, я не смог найти ни одного вменяемого гайда по симу чуду, ни на русском, ни на английском. На хабре и вовсе, если верить поиску, есть всего одно упоминание, и то — в комментариях (из него я и узнал про эту утилиту). К сожалению, официальная документация написана довольно поверхностно: не указан ряд опций, которые можно использовать во время сборки, для выяснения которых мне пришлось лезть в исходники, не описан ряд тонкостей, с которыми мне пришлось столкнуться.

Данная статья не претендует на всеобъемлющее описание *pyqtdeploy* и работы с ним, но, в конце концов, всегда приятно иметь все в одном месте, не так ли?

Замечание. В статье исполняемый файл собирается под linux. Несмотря на это, в качестве синонима используется слово "exe'шник" для экономии букв и уменьшения числа повторений.

Все началось с того, что мне захотелось один мой проект запихнуть в исполняемый файл со всеми зависимостями (вы и сами уже догадались). Сначала я решил попробовать провернуть эту операцию с помощью PyInstaller — шикарный инструмент, простой, хорошо документированный. Но на выходе я получил папку размером 170 МБ (для сравнения, весь PyQt5 весил около 180 МБ). Поковырявшись в собранных либах, я понял, что используемые мной модули — QtCore, QtGui, QtWidgets — тащат с собой почти весь пакет. Попытки поиграться с опцией `--exclude-module` не увенчались успехом. Справедливости ради, если использовать опцию `--onefile` и включить сжатие, то получится файл размером 60 МБ, что все равно много. К тому же, во время запуска происходит

разархивирование программы во временную папку, что увеличивает время старта и все равно (пусть и где-то там) отжирает все те же 170 МБ.

Тут мне подвернулся pyqtdeploy. "Утилита от самих разработчиков PyQt... Ну уж они-то должны знать, как по-максимуму отвязаться от лишних зависимостей внутри PyQt и Qt?" — подумал я и взялся плотненько за сей агрегат.

Так что же такое pyqtdeploy? В первом приближении, то же самое, что и выше перечисленные программы. Все ваши модули (стандартная библиотека, PyQt, все прочие модули) упаковываются средствами Qt (используется утилита rcc) в так называемый файл ресурсов, генерируется обертка вокруг питоновского интерпретатора на C++, позволяющая получать доступ ко все вашим модулям, и потом все это пакуется/компилируется/... в исполняемый файл. Для работы самого pyqtdeploy нужны Python 3.5+ и PyQt5. Перечислим несколько особенностей (за подробностями сюда и сюда):

- может собирать exe'шники на основе PyQt4 и PyQt5, Python 2.7 и Python 3.3+ (максимальная поддерживаемая версия на данный момент Python 3.7.2);
- позволяет статически (все пихаем в exe'шник) и динамически привязывать зависимости (использовать уже установленные в системе библиотеки, пакеты — с рядом ограничений);
- поддерживаемые платформы:
  - android-32;
  - android-64;
  - ios-64;
  - linux-64;
  - macos-64;
  - win-32;
  - win-64;
- также позволяет собирать несвязанные с PyQt и Qt программы, но из-за тесной интеграции с QtCore, будет тянуть оттуда кое-что в качестве зависимостей.

Установка pyqtdeploy

Как уже было сказано выше, у нас должен быть установлен Python 3.5+ и PyQt5:

```
pip install PyQt5 pyqtdeploy
```

Сборка нашего exe'шника состоит из нескольких этапов:

- Разработка нашей Python-программы, как обычно (сюрприз!);
- Сборка так называемого **sysroot** для нашей платформы, где будут лежать собранные из исходников нужные зависимости;
- Создание "проектного" файла с расширением **.pdy**, где будет вся необходимая информация для сборки нашего exe'шника (пути к собранным Qt, PyQt, Python, прочим библиотекам и модулям и другие опции);
- Собственно сборка exe'шника с помощью qmake.

Структура программы

Возьмем в качестве примера проект со следующей структурой: *main.py* — "точка входа" для нашей программы, она вызывает *mainwindow.py* — допустим, отрисовывает окошечко с виджетами и берет
из *resources* иконку *icon.png* и *mainwindow.ui*, сгенерированный нами с помощью Qt Designer. Имеющиеся зависимости, версии библиотек и прочие необходимые вещи будут всплывать по ходу повествования:

```
main.py
src/
    |---__init__.py
    |---gui/
        |---mainwindow.py
        |---__init__.py
    |---resources/
        |---__init__.py
        |---images/
```

```
            |---icon.png
            |---__init__.py
        |---ui/
            |---mainwindow.ui
            |---__init__.py
```

Обзор плагинов sysroot (документация)

Как уже было сказано ранее, на этом этапе мы собираем все необходимые части, которые затем будут использоваться при генерации исполняемого файла. Данный процесс осуществляется с использованием конфигурационного файла *sysroot.json* (в принципе, вы можете назвать его как хотите и указать затем путь к нему). Он состоит из блоков, каждый из которых описывает сборку отдельного компонента (Python, Qt и т.д.). В pyqtdeploy реализован API, позволяющий вам написать свой плагин, управляющий сборкой необходимой вам библиотеки/модуля/whatever, если он еще не реализован разработчиками pyqtdeploy. Давайте пробежимся по стандартным плагинам и их параметрам (примеры из документации):

openssl (не обязательный) — позволяет собирать из исходников или использовать установленную в системе библиотеку (подробности). Компонент, описывающий данный плагин в *sysroot.json*, выглядит следующим образом:

```
"android|macos|win#openssl": {
    "android#source": "openssl-1.0.2r.tar.gz",
    "macos|win#source": "openssl-1.1.0j.tar.gz",
    "win#no_asm": true
```

```
}
```

Первое, на что следует обратить внимание, это синтаксис: `arch1|arch2|...#plugin-name`. То есть мы можем выбрать, на какой платформе использовать этот плагин (ios, android, macos, win, linux), а на какой — нет. Более того, этот синтаксис применим и к параметрам внутри блока.

Параметры:

- source (обязательный) — имя архива с исходниками;
- no_asm (не обязательный) — выключаем ассемблерные оптимизации. Если включен, в PATH должен быть установлен nasm;
- python_source (не обязательный) — имя архива, содержащего патчи, необходимые для сборки OpenSSL под macOS для Python v3.6.4 и более ранних версий;

zlib (не обязательный) — используется при сборке других компонентов (если не указан, по идее, будет использоваться тот, что установлен в системе) (подробности):

```
"ios|linux|macos|win#zlib": {
    "source": "zlib-1.2.11.tar.gz",
    "static_msvc_runtime": true
}
```

Параметры:

- source (обязательный) — очевидно, имя архива с исходниками;
- static_msvc_runtime (не обязательный) — статически привязать MSVC библиотеки (Windows);

qt5 (обязательный) — тут понятно (подробности):

```
"qt5": {
    "android-32#qt_dir": "android_armv7",
    "android-64#qt_dir": "android_arm64_v8a",
    "ios#qt_dir": "ios",
    "linux|macos|win#source": "qt-everywhere-src-5.12.2.tar.xz",
    "edition": "opensource",
    "android|linux#ssl": "openssl-runtime",
    "ios#ssl": "securetransport",
    "macos|win#ssl": "openssl-linked",
    "configure_options": [
        "-opengl", "desktop", "-no-dbus", "-qt-pcre"
    ],
    "skip": [
        "qtactiveqt", "qtconnectivity", "qtdoc", "qtgamepad",
        ...
    ],
    "static_msvc_runtime": true
}
```

Параметры:

- qt_dir (не обязательный, если указан source) — путь к папке с установленным Qt;
- source (не обязательный, если указан qt_dir) — имя архива с исходниками Qt;
- edition (обязательный, если указан source) — один из 2 вариантов:
  - commercial;
  - opensource;
- ssl — 3 возможных варианта:
  - openssl-linked — будет собран из исходников (подробности должны быть указаны в описании компонента openssl);
  - securetransport — используется SSL, реализованный в Qt (который, в свою очередь, будет использовать Apple's Secure Transport);
  - openssl-runtime — используется версия OpenSSL, установленная в системе;
- configure_options — дополнительные опции, используемые при сборке Qt. Существует их целая прорва, смотрим тут;
- skip — позволяет исключить из сборки ненужные модули (точнее говоря, top-level директории, содержащие модули). Открываем архив с исходниками Qt и видим папки, начинающиеся с qt — это и есть top-level директории. Имейте в виду, что эти папки могут содержать и те модули, что вам нужны. К сожалению, можно скипнуть только top-level директорию целиком (подробности);
- disabled_features — позволяет исключить выбранный функционал. Для просмотра всех возможных фич можно воспользоваться командой `configure -list-features` (подробности)
- static_msvc_runtime (не обязательный) — статически привязать MSVC библиотеки (Windows);

python (обязательный) — тут тоже понятно (подробности):

```
"python": {
    "build_host_from_source": false,
    "build_target_from_source": true,
    "source": "Python-3.7.2.tar.xz"
```

```
}
```

Параметры:

- build_host_from_source (обязательный) — true — собираем Python для хоста из исходников, false — используем установленный Python (не поддерживается для win32);
- build_target_from_source (обязательный) — true — собираем Python для целевой платформы из исходников, false — используем установленный Python (использование установленного Python поддерживается только на win32);
- source (обязательный, если Python собирается из исходников) — имя архива с исходниками Python;
- version (обязательный, если используется установленный Python) — версия установленного Python;
- dynamic_loading (не обязательный) — true — включить поддержку динамической загрузки модулей расширения (тех, что на C);
- host_installation_bin_dir (не обязательный) — путь к установленному Python, если не собирается из исходников (если не указан, на win ищется в реестре автоматически, на других платформах — в PATH);

sip (обязательный) — компонент, отвечающий за автоматическое генерирование Python-bindings для C/C++ библиотек (подробности тут и тут):

```
"sip": {
    "module_name": "PyQt5.sip",
    "source": "sip-4.19.15.tar.gz"
}
```

Параметры:

- module_name (обязательный) — имя sip-модуля;
- source (обязательный) — имя архива с исходниками sip;

pyqt5 (обязательный) — тут тоже понятно (подробности):

```
"pyqt5": {
    "android#disabled_features": [
        "PyQt_Desktop_OpenGL", "PyQt_Printer", "PyQt_PrintDialog",
        "PyQt_PrintPreviewDialog", "PyQt_PrintPreviewWidget"
    ],
    "android#modules": [
        "QtCore", "QtGui", "QtNetwork", "QtPrintSupport", "QtWidgets",
        "QtAndroidExtras"
    ],
    "ios#disabled_features": [
        "PyQt_Desktop_OpenGL", "PyQt_MacOSXOnly",
        ...
    ],
    "ios|macos#modules": [
        "QtCore", "QtGui", "QtNetwork", "QtPrintSupport", "QtWidgets",
        "QtMacExtras"
    ],
    "linux#modules": [
        "QtCore", "QtGui", "QtNetwork", "QtPrintSupport", "QtWidgets",
```

```
        "QtX11Extras"
    ],
    "win#disabled_features": ["PyQt_Desktop_OpenGL"],
    "win#modules": [
        "QtCore", "QtGui", "QtNetwork", "QtPrintSupport", "QtWidgets",
        "QtWinExtras"
    ],
    "source": "PyQt5_*-5.12.1.tar.gz"
}
```

Параметры:

- disabled_features (не обязательный) — позволяет выключить конкретный функционал. Если не указан, выключаемые фичи определяются автоматически на основе фич, выключенных в собранном нами Qt (подробности);
- modules (обязательный) — перечисляем модули, которые мы хотим собрать (подробности);
- source (обязательный) — имя архива с исходниками PyQt;

pyqt3D, **pyqtchart**, **pyqtdatavisualization**, **pyqtpurchasing**, **qscintilla** (не обязательные) — дополнительные модули, не входящие в состав PyQt. Имеют единственный параметр `source` — имя архива с исходниками.

Стоит заметить, что некоторые значения параметров могут не работать друг с другом. В таких случаях вы получите ошибку при сборке **sysroot** с информацией, что не так. Я постарался здесь описать такие случаи, по крайней мере, для обязательных компонентов.

Собираем sysroot

Давайте взглянем на итоговый *sysroot.json* для нашей программы:

```
{
    "linux#zlib": {
        "source": "zlib-1.2.11.tar.gz"
    },
    "linux#qt5": {
        "source": "qt-everywhere-src-5.12.2.tar",
        "edition": "opensource",
        "configure_options": [
            "-no-dbus", "-no-system-proxies", "-no-cups", "-no-sql-db2",
            "-no-sql-ibase", "-no-sql-mysql", "-no-sql-sqlite",
            "-no-sql-sqlite2", "-no-sql-oci", "-no-sql-odbc",
            "-no-sql-psql", "-no-sql-tds", "-no-sqlite", "-ccache",
            "-optimize-size"
        ],
        "skip": [
            "qt3d", "qtactiveqt", "qtandroidextras", "qtcanvas3d",
            "qtcharts", "qtconnectivity", "qtdatavis3d", "qtdeclarative",
            "qtdoc", "qtgamepad", "qtgraphicaleffects", "qtlocation",
```

```
            "qtmacextras", "qtmultimedia", "qtnetworkauth", "qtpurchasing",
            "qtquickcontrols", "qtquickcontrols2", "qtremoteobjects",
            "qtscript", "qtscxml", "qtsensors", "qtserialbus",
            "qtserialport", "qtspeech", "qtsvg", "qttools",
            "qttranslations", "qtvirtualkeyboard", "qtwayland",
            "qtwebchannel", "qtwebengine", "qtwebglplugin",
            "qtwebsockets", "qtwebview", "qtwinextras", "qtx11extras",
            "qtxmlpatterns"
        ],
        "disabled_features": [
            "network", "bearermanagement", "dnslookup", "dtls", "ftp",
            "http", "localserver", "networkdiskcache", "networkinterface",
            "networkproxy", "socks5", "udpsocket", "concurrent", "future",
            "cups", "printer", "printdialog", "printpreviewdialog",
            "printpreviewwidget", "sql", "sqlmodel", "testlib", "xml"
        ]
    },

    "linux#python": {
        "build_host_from_source": false,
        "build_target_from_source": true,
```

```
        "source": "Python-3.7.2.tgz",
        "dynamic_loading": true
    },
    "linux#sip": {
        "module_name": "PyQt5.sip",
        "source": "sip-4.19.15.tar.gz"
    },
    "linux#pyqt5": {
        "modules": ["QtCore", "QtGui", "QtWidgets"],
        "source": "PyQt5_*-5.12.2.tar.gz"
    }
}
```

Что интересного мы тут видим? Во-первых, не используется ряд компонентов(например, ssl, pyqt3D и прочие). Во-вторых, собирать наш exe'шник мы будет под linux (а точнее, linux-64; в нашем случае, можно не указывать перед каждым компонентом платформу).

Далее, в qt5 по-максимуму выключены модули и функции, которые не будут использоваться (те, о назначении которых у меня было хотя бы минимальное представление). Среди top-level директорий собирается только QtBase. Особо упомяну опции `-optimize-size` и `-ccache`. Первая позволяет уменьшить размер собранного Qt и, соответственно, итогового файла (у меня получилось минус 5 МБ), но увеличится время компиляции, вторая — использовать ccache (по крайней мере, на linux), что при повторных компиляциях СУЩЕСТВЕННО уменьшает время (у меня уменьшилось раз в 5). Никакой настройки не требует, просто ставим командой `apt install ccache`.

В pyqt5 собираем только модули QtCore, QtGui, QtWidgets.

В python включен `dynamic_loading`, так как мы хотим позднее динамически прилинковать C-extension.

Прежде чем приступить к сборке **sysroot**, не забываем скачать все необходимые исходники: zlib, Qt5, Python, sip, PyQt5 и кладем их в папочку с *sysroot.json* (можно и любую другую, указав потом путь к ней). Запускаем сборку:

```
pyqtdeploy-sysroot sysroot.json
```

Данная команда имеет еще несколько опций, которые можно посмотреть здесь.

Крайне рекомендую также использовать опцию `--verbose`. Будьте готовы к тому, что вы получите целую кучу ошибок, прежде чем все удачно соберется. Многие из них будут связаны с тем, что у вас не установлены dev-пакеты. Я их здесь не перечисляю, ибо они зависят от вашей конфигурации и платформы. Наверняка, вам нужен будет *python3-dev*, также смотрим тут (особенно, разделы *Requirements*). Правда, вам никто не запрещает использовать для тех же Qt и Python уже установленные версии (я не пробовал, возможны свои подводные камни).

Ну и запаситесь попкорном, ибо, в зависимости от мощности вашего ~~калькулятора~~ компьютера, это может занять немалое время.

Создаем "проектный" файл (документация)

Как только у нас все удачно собралось, приступаем к выбору модулей, которые мы хотим запаковать в exe'шник. Для этого в pyqtdeploy есть удобная утилита с GUI. Запускаем (имя **.pdy** файла может быть любым):

```
pyqtdeploy main.pdy
```

Application Source. В первой вкладке мы видим следующие настройки:

- Name — имя вашего будущего exe'шника;

- Main script file (не указывается, если используется Entry Point) — скрипт, используемый для запуска программы (в нашем случае, *main.py*);
- Entry Point (не указывается, если используется Main script file) — точка входа для программы, основанной на setuptools;
- sys.path — используется для указания дополнительных директорий, zip-файлов и яиц (тех, что Python egg), которые будут добавлены в sys.path (я не использовал, смотрим доки, там подробно описана эта опция);
- Target Python version — версия Python;
- Target PyQt version — PyQt4 или PyQt5 (игнорируется, если вы мазохист и решили собрать программу, не использующую PyQt, этим монстром);
- Use console — выбрать, если приложение должно использовать консоль (только Windows). Может быть полезно для дебага;
- Application bundle — выбрать, если приложение должно быть собрано как bundle (только MacOS);
- Application Package Directory — содержит все файлы, составляющие вашу программу. Для добавления жмем кнопку Scan… У нас папка со всеми "кишками" (*src*) отделена от *main.py*, так что выбираем эту папку и галочками выделяем все файлы, которые мы хотим включить в итоговый файл. Если же у вас нет такого разделения (т.е. *main.py* находится внутри *src*), то напротив *main.py* галочку нужно снять (или напротив вашего аналога, указанного в Main script file).

Еще один момент: любой файл с расширением **.py** будет "заморожен" (будет сгенерирован байт-код) — в ряде случаев это может быть нежелательным.

Кнопки справа:

- Scan… — добавляем файлы в Application Package Directory;
- Remove all — очищаем Application Package Directory;
- Include all — выделяем все файлы в Application Package Directory;
- Exclude all — снимаем выделение со всех файлов в Application Package Directory;
- Exclusions — паттерны, позволяющие исключить файлы из Application Package Directory. Дважды кликаем на пустой строке для добавления;

qmake. Так как в сборке участвует qmake, здесь можно добавить дополнительные параметры для него (я не использовал);

PyQt Modules. На этой вкладке выделяем все PyQt-модули, которые мы явно импортируем в нашей программе. Если они зависят от других модулей, те

выделятся автоматически. В нашем случае использовались QtCore, QtGui, QtWidgets, uic; sip подхватился автоматом.
Если планируется использовать уже установленный PyQt, а не привязывать статически его к нашему исполняемому файлу, ничего не выделяем (такой сценарий не тестировался).

Standard Library. Здесь тот же подход, что и в предыдущем пункте, только для стандартной библиотеки. Если у вас в программе явно импортируется какой-то модуль, ставим галку. Если выделенным нами модулям (или самому интерпретатору) нужны другие модули, они выделятся автоматом (квадратики).

Правда это не всегда работает. Если поставили какой-то пакет со стороны (через тот же pip), и он импортирует что-то из стандартной библиотеки (еще не выделенное), вы получите при запуске `ImportError`. Так что вам придется вернуться сюда и поставить галочку. Например, я использую библиотеку PIL, и одному из модулей нужны была библиотека fractions.

Python использует ряд модулей/пакетов (например, ssl), которым для работы нужны внешние библиотеки. Если мы хотим их статически привязать, то мы настраиваем это дело справа. В INCLUDEPATH указываем путь к заголовочным файлам (headers), в LIBS — путь к этой либе (мной не использовались, так что подробности смотрим в доках).

Other Packages. На этой вкладке выбираем необходимые нам сторонние пакеты (например, установленные из pypi). Подход тот же, что и в Application source: кликаем дважды на пустой строке, выбираем папку (в нашем случае, site-packages используемого при разработке virtual environment), жмем Scan и выбираем нужные пакеты/модули (у нас это PIL).

Other Extension Modules. Тут мы настраиваем модули расширения на C, которые хотим СТАТИЧЕСКИ привязать к exe'шнику (сторонние; те, что в стандартной библиотеке, привязываются сами).

Мы может настроить как компиляцию с нуля этих самых расширений, так и привязку уже скомпилированных. Второе делается довольно просто. Допустим у нас есть пакет **Package** со статической либой **Lib.a**, то в поле Name указываем полное имя расширения, используемое во время импорта — **Package.Lib** (без расширения **.a**); затем в поле LIBS указываем путь к этому расширению, например, *-L/home/user1/venv/programme1/lib/python3.7/site-packages/Package -*

lLib (это специальный формат, также можно указать путь "по старинке", */home/user1/venv/programme1/lib/python3.7/site-packages/Package/Lib.a*).

С компиляцией я не разбирался, но советую почитать, во-первых, про эту вкладку в доках, во-вторых, про qmake (там гораздо подробнее описаны опции, чем в pyqt'шных доках).

А что, если у нас динамическая либа, например, **Lib.so**? Еще проще — переименовываем ее в **Package.Lib.so** (т.е. все то же полное имя расширения, используемое во время импорта + расширение) и кладем его рядом с нашим exe'шником. Все должно подхватится, если это простое расширение без всяких зависимостей. В противном случае, ждите опять кучу `ImportError`. Мне, например, так и не удалось прикрутить **_imaging.so**, используемый PIL'ом.

Locations. Тут тоже подробно не останавливаемся, за описанием отдельных путей сюда. Если вы действовали в соответствии с этой статьей (собранный **sysroot** лежит тут же, рядом с *main.pdy*), тут менять ничего не надо.

Собираем exe'шник (документация)

Наконец-таки собираем наш исполняемый файл:

```
pyqtdeploy-build main.pdy
```

```
cd build-linux-64
```

```
../sysroot-linux-64/host/bin/qmake
```

```
make #nmake для win
```

Гипотетически, все должно собраться, на деле — доки и гугл вам в помощь.

Лирическое отступление #1 — меняем поведение программы в зависимости от того, "заморожено" оно или нет

Если вам нужно определить, запущена ваша программа как есть или из собранного exe'шника, используется тот же подход, что и в PyInstaller:

```
if getattr(sys, 'frozen', False):
    # запустили из exe'шника
else:
    # запустили не из exe'шника
```

Лирическое отступление #2 — использование ресурсов (изображения, иконки и пр.)

У Qt имеется специальная "система ресурсов", которая позволяет с помощью утилиты rcc упаковать любые бинарные файлы в exe'шник. Далее с помощью пути специального формата вы можете получить доступ к необходимому ресурсу. В нашем проекте файл с иконкой *icon.png* расположен в *src/resources/images*, тогда путь в "системе ресурсов" будет выглядеть так — *:/src/resources/images/icon.png*. Как видите, ничего хитрого. Однако с таким путем есть одна засада — его понимают только Qt'шные функции. Т.е. если вы напишите у себя в программе что-нибудь в духе:

```
icon = QIcon(':/src/resources/images/icon.png')
```

Все будет в порядке. Но если, например, так:

```
icon_file = open(':/src/resources/images/icon.png', 'rb')
icon = icon_file.read()
```

Ничего не выйдет, ибо open будет пытаться найти такой путь в вашей файловой системе и, естественно, ничего не найдет.

Если вам нужно читать запакованные ресурсы не только средствами Qt (например, вы, как и я, создавали GUI с помощью Qt Designer и получили файл **.ui**, который потом надо прочитать с помощью `loadUi`), нужно будет сделать как-то так:

```
ui_file = QtCore.QFile(':/src/resources/images/icon.png')
```

```
ui_file.open(QtCore.QIODevice.ReadOnly)
```

```
data = ui_file.readAll()
```

```
ui_file.close()
```

```
ui_file = BytesIO(bytes(data))
```

Итоги

Стоит ли так сильно заморачиваться, если вам нужен exe'шник, и старые добрые дедовские способы распространения программы вам по каким-то причинам не подходят? Если вы не используете PyQt, то, на мой взгляд, точно не стоит. Используйте что-нибудь более дружелюбное (тот же PyInstaller). Если хотите выжать максимум соков из вашего файла — дерзайте. В конечном счете мне таки удалось уменьшить размер файла до ~40 МБ (с `-optimize-size` ~35 МБ), что все-равно больше, чем хотелось бы.

Когда у нас собрана минимально необходимая Qt и PyQt, было бы неплохо попробовать сделать на их основе exe'шник с помощью PyInstaller или cx_Freeze и посмотреть на размер, но это, как говорится, уже другая история...

Теги:

- python
- pyqtdeploy
- pyqt
- qt
- исполняемый файл
- freeze
- pyinstaller
- cx_freeze
- py2exe
- py2app

Хабы:

- Python
- Qt

A `setup.cfg` template for my Python projects

Raw

setup.cfg

```
# https://gist.github.com/althonos/6914b896789d3f2078d1e6237642c35c


[metadata]
name = {name}
version = file: {name}/_version.txt
author = Martin Larralde
author-email = martin.larralde@embl.de
home-page = https://github.com/althonos/{name}
description = {description}
long-description = file: README.md
long_description_content_type = text/markdown
license = MIT
license-file = COPYING
platform = any
keywords = {keywords}
classifiers =
    Development Status :: 3 - Alpha
    Intended Audience :: Developers
    License :: OSI Approved :: MIT License
    Operating System :: OS Independent
    Programming Language :: Python
    Programming Language :: Python :: 3.4
    Programming Language :: Python :: 3.5
    Programming Language :: Python :: 3.6
    Programming Language :: Python :: 3.7
    Programming Language :: Python :: 3.8
    Topic :: Software Development :: Libraries :: Python Modules
project_urls =
    Bug Tracker = https://github.com/althonos/{name}/issues
    Changelog = https://github.com/althonos/{name}/blob/master/CHANGELOG.md


[options]
zip_safe = false
include_package_data = true
python_requires = >= 2.7, != 3.0.*, != 3.1.*, != 3.2.*
packages = {name}
test_suite = tests
setup_requires =
    setuptools
    # setuptools >=30.3.0     # minimal version for `setup.cfg`
    # setuptools >=38.3.0     # version with most `setup.cfg` bugfixes
```

```
install_requires =
    {install_requires}
tests_require =
    {tests_require}


[options.extras_require]
dev =
    docutils
    Pygments
test =
    green
    coverage
ci =
    # codacy-coverage
    # codecov


[options.package_data]
{name} = py.typed, _version.txt, *.pyi


[bdist_wheel]
universal = true


[check]
metadata = true
restructuredtext = true
strict = true


[sdist]
formats = zip, gztar


[coverage:report]
show_missing = true
exclude_lines =
    pragma: no cover
    if False
    # @abc.abstractmethod
    # @abc.abstractproperty
    # raise NotImplementedError
    # return NotImplemented
    # except ImportError
    # raise MemoryError
    # if __name__ == .__main__.:
```

```
    # if typing.TYPE_CHECKING:


[green]
file-pattern = test_*.py
verbose = 2
no-skip-report = true
quiet-stdout = true
run-coverage = true


[pydocstyle]
match-dir = (?!tests)(?!resources)(?!docs)[^\.].*
match = (?!test)(?!setup)[^\._].*\.py
inherit = false
ignore = D200, D203, D213, D406, D407   # Google conventions

[flake8]
max-line-length = 99
doctests = True
exclude =  .git, .eggs, __pycache__, tests/, docs/, build/, dist/


[mypy]
disallow_any_decorated = true
disallow_any_generics = true
disallow_any_unimported = false
disallow_subclassing_any = false
disallow_untyped_calls = true
disallow_untyped_defs = true
ignore_missing_imports = true
warn_unused_ignores = true
warn_return_any = true
```