Урок 8

Подготовка дистрибутива

Подготовка дистрибутива, setuptools, cx_freeze.

Введение

Это завершающее занятие в курсе «Python. Уровень 2.2». Рассмотрим вопросы, связанные с распространением программного обеспечения — в частности, создание инсталяционных пакетов. Итогом курса должно стать готовое для передачи конечному пользователю приложение, снабженное документацией.

Подготовка дистрибутива

Работа с пакетами

Руководство по созданию пакетов Python (Python Packaging User Guide) делит инструменты на 2 группы:

- инструменты установки пакетов (**pip**, **virtualenv**)
- инструменты для создания и распространения пакетов (**setuptools**, **twine**).

В рамках данного занятия рассмотрим **setuptools**.

Настройка проекта

Самый простой способ организации программного кода большого проекта — это разделение его на несколько пакетов. Это сделает код проще, его будет легче понимать, поддерживать и изменять кода. Также это даст возможность переиспользовать пакеты, которые станут удобными компонентами.

setup.py

Входной точкой для установщика пакета (например, для **pip**) будет скрипт **setup.py**.

Структуру директории пакета лучше сформировать таким образом: создать отдельную директорию **package_name**, в нее поместить директорию **src** с исходными файлами пакета; в директории **package_name** также расположить файл **setup.py**:

```
package_name
│   setup.py
└───src
    │   add.py
    │   divide.py
    │   multiply.py
    │   subtract.py
    │   __init__.py
    └───adv
          │ fib.py
          │ sqrt.py
          │ __init__.py
```

Файл **setup.py** содержит метаданные, описанные в модуле distutils, которые передаются функции **setup**. Вместо стандартного модуля **distutils** лучше использовать модуль setuptools, в котором есть полезные расширения.

Минимальное содержимое **setup.py**:

```
from setuptools import setup

setup(
  name='mypackage',
)
```

- **name** задает полное имя пакета.

Теперь скрипт предоставляет полезные команды, узнать которые можно, вызвав скрипт с параметром **--help-commands** (список команд приводится в сокращении):

```
python setup.py --help-commands

Standard commands:
  build             build everything needed to install
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  sdist             create a source distribution (tarball, zip file, etc.)
  register          register the distribution with the Python package index
  bdist             create a built (binary) distribution
  check             perform some checks on the package
  upload            upload binary package to PyPI
Extra commands:
  bdist_wheel       create a wheel distribution
  build_sphinx      Build Sphinx documentation
  alias             define a shortcut to invoke one or more commands
  test              run unit tests after in-place build

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

Раздел **Standard commands** содержит команды, реализуемые модулем **distutils**. Раздел **Extra commands** — команды, предоставляемые дополнительными модулями, такими как **setuptools**. Одна из них — **bdist_wheel**, предоставляемая пакетом **wheel**.

setup.cfg

Файл **setup.cfg** содержит параметры по умолчанию для скрипта **setup.py**. Это может быть полезно, когда процесс сборки и распространения пакета достаточно сложен и требует большого количества аргументов для команд скрипта **setup.py**. Это позволяет хранить параметры по умолчанию в коде или в для каждого проекта отдельно. Распространение становится независимым от проекта, сохраняется прозрачность создания и распространения пакета.

Структура файла **setup.cfg** аналогична той, которая обрабатывается встроенным модулем configparser — это разновидность INI-файлов Microsoft Windows.

Пример файла **setup.cfg**, который предоставляет параметры по умолчанию для всех команд, а также для **sdist** и **bdist_wheel**:

```
[global]
quiet=1

[sdist]
formats=zip,tar

[bdist_wheel]
universal=1
```

Этот пример подразумевает, что распространение исходных кодов (**sdist**) будет выполняться в двух форматах — **ZIP** и **TAR**, а распространение в формате **wheels** (**bdist_wheel**) — в формате **universal wheels** (независимо от версии Python). Установка глобального параметра **quiet=1** означает, что вывод в консоль будет подавляться для каждой команды (однако такой подход не всегда уместен).

MANIFEST.in

При создании дистрибутива командой **sdist**, утилита **distutils** обходит директорию пакета для поиска файлов, которые будут включены в архив:

- все Python-файлы, подразумеваемые в параметрах **py_modules**, **packages** и **scripts**;
- все C-файлы, перечисленные в параметре **ext_modules**.

Если требуется включить в дистрибутив дополнительные файлы, можно создать специальный файл **MANIFEST.in** в той же директории, что и **setup.py**. В этом файле можно указать, какие файлы должны быть дополнительно включены в архив командой **sdist**. Правила включения/исключения указываются по одному на строке, например:

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.txt *.py
```

Дополнительно про файл **MANIFEST.in** можно почитать в документации на distutils.

Метаданные

Помимо имени и версии распространяемого пакета могут быть указаны следующие важные аргументы функции **setup**:

- **description** — несколько предложений для описания пакета;
- **long_description** — полное описание, которое может быть в формате **reStructuredText**;
- **keywords** — список ключевых слов, характеризующих пакет;
- **author** — имя автора или организации;
- **author_email** — e-mail автора;

- **url** — URL проекта;
- **license** — лицензия (GPL, LGPL и т.п.);
- **packages** — список всех пакетов внутри дистрибутива;
- **namespace_packages** — список пакетов, оформленных в пространство имен (**namespaced**).

Сборка пакета

Существует два основных типа дистрибутивов для пакетов Python:

- дистрибутив исходных кодов (**Source distributions**);
- бинарный дистрибутив (**Built (binary) distributions**).

Дистрибутив исходного кода — самый простой и платформонезависимый вариант. Для пакетов на чистом Python это достаточно просто — дистрибутив включает только исходные коды Python.

Более сложная ситуация, когда пакет имеет дополнения, написанные на C/C++. Распространение исходных кодов тогда подразумевает, что у конечного пользователя есть полный набор разработки (компилятор, заголовочные файлы), а это далеко не всегда так. Для таких случаев лучше выбрать бинарный дистрибутив, который может включать уже собранные C-расширения для конкретных платформ.

Команда sdist

Команда **sdist** — самая простая из доступных setup-команд. Она создает дерево релиза, куда скопировано все, чтобы запустить пакет. Затем это дерево архивируется в один или несколько архивных форматов. Таким образом, дистрибутив пакета создается независимым от дерева разработки.

При создании архива в имя файла также добавляется номер версии. Он задается параметром **version** функции **setup** (если не указан, устанавливается значение 0.0.0). Это значение используется при обновлении пакета. Каждый раз при релизе нужно менять значение, чтобы конечная система знала об изменении.

Пусть исходный файл **setup.py** содержит следующие сведения (см. **листинг 1**):

```
from setuptools import setup

setup(name = "math_package",
      version = "0.1",
      description = "A Simple Math Package",
      author = "Mike Driscoll",
      author_email = "mike@somedomain.com",
      url = "http://www.blog.pythonlibrary.org",
      packages=["src"]
      )
```

Тогда при запуске команды **python3 setup.py sdist** в консоль будет выведено следующее:

```
running sdist
...
```

```
running check
creating math_package-0.1
creating math_package-0.1\math_package.egg-info
creating math_package-0.1\src
...
copying math_package.egg-info\SOURCES.txt ->
math_package-0.1\math_package.egg-info
copying math_package.egg-info\dependency_links.txt ->
math_package-0.1\math_package.egg-info
copying math_package.egg-info\top_level.txt ->
math_package-0.1\math_package.egg-info
copying src\__init__.py -> math_package-0.1\src
...
Writing math_package-0.1\setup.cfg
creating 'dist\math_package-0.1.zip' and adding 'math_package-0.1' to it
adding 'math_package-0.1\MANIFEST.in'
adding 'math_package-0.1\PKG-INFO'
adding 'math_package-0.1\setup.cfg'
adding 'math_package-0.1\setup.py'
...
removing 'math_package-0.1' (and everything under it)
```

После чего в директории с файлом **setup.py** появится поддиректория **dist**, с архивом **math_package-0.1.zip**.

Команда bdist и формат wheel

Для распространения бинарных дистрибутивов **distutils** предоставляет команду **build**, которая собирает пакет за 4 шага:

- **build_py** — сборка Python-модулей: компилирование их в байт-код, копирование в директорию **build**;
- **build_clib** — сборка C-библиотек (если они есть): компилирование в статическую библиотеку и копирование в директорию **build**;
- **build_ext** — аналогично **build_clib**, но для C-расширений;
- **build_scripts** — сборка модулей, отмеченных как **scripts**.

Каждый из этих шагов — команда, которая может быть вызвана отдельно. Результат сборки — директория **build**, содержащая все для установки пакета. На данный момент **distutils** не поддерживает кросс-компиляцию, т.е. сборка производится для той же платформы, что и исходная.

Другой тип бинарного дистрибутива — **wheels**, предоставляемый пакетом **wheel**. Создание такого формата выполняется командой **bdist_wheel**. Она является альтернативой для **bdist** и позволяет создать платформозависимые дистрибутивы (Windows, Mas OS). Этот формат был введен на замену устаревшему **setuptools-eggs** (egg-формат более не поддерживается).

Некоторые преимущества формата **wheel** (полное описание можно найти на портале Python Wheels):

- более быстрая установка Python-модулей и C-расширений;
- не использует дополнительного запуска кода при установке (не запускает файл **setup.py**);

- установка C-расширений не требует наличия компилятора на Windows и Mac;
- улучшает процесс кэширования для тестирования и непрерывной интеграции;
- создает .pyc-файлы как часть установки для проверки соответствия версии интерпретатора Python;
- более последовательная установка на разных платформах и машинах.

Ассоциация Python Packaging Authority рекомендует использовать **wheel** как формат по умолчанию для дистрибутива.

Результат сборки командой **bdist_wheel** будет размещен в директории **dist**: файл **math_package-0.1-py2.py3-none-any.whl**.

Исполняемые файлы

Тема создания отдельных исполняемых файлов часто не включается в материалы по сборке пакетов. Python в стандартной библиотеке не содержит инструментов создания исполняемых файлов — тех, которые могут запускаться без установки интерпретатора Python.

Компилируемые языки программирования имеют большое преимущество перед Python в том, что позволяют создавать отдельные исполняемые файлы для конкретной платформы. Пакеты Python подразумевают обязательное наличие интерпретатора Python. Это не всегда удобно для конечного пользователя, который привык запускать программы двойным кликом мыши. Поэтому отличным навыком в разработке будет умение создать бинарную сборку, которая работает как обычное скомпилированное приложение. Можно создать исполняемый файл, который внутри себя содержит интерпретатор Python и скрипты приложения. Это избавит конечного пользователя от «головной боли» об интерпретаторе и зависимостях.

Распространение в виде отдельных исполняемых файлов будет **лучшим решением** для нетехнического конечного пользователя и распространения приложений под Windows, а также для:

- приложений, использующих определенную версию Python, которую сложно установить на конечную систему;
- приложений, использующих модифицированную версию CPython;
- приложений с графическим интерфейсом пользователя;
- приложений, имеющих многочисленные бинарные дополнения, написанные на разных языках;
- игр.

Популярные инструменты для сборки

Существует несколько проектов, решающих проблему отсутствия в стандартной библиотеке Python инструментов для создания исполняемых файлов:

- **PyInstaller** (http://www.pyinstaller.org/) — наиболее продвинутый инструмент, поддерживает Python версий 2.7 и 3.3+, а также платформы:
    - Windows (32-bit and 64-bit);
    - Linux (32-bit and 64-bit);
    - Mac OS X (32-bit and 64-bit);
    - FreeBSD, Solaris, and AIX;

- **cx_Freeze** (https://anthony-tuininga.github.io/cx_Freeze/) — проще, чем **PyInstaller**; тем не менее поддерживает три основные платформы:
    - Windows;
    - Linux;
    - Mac OS X;
- **py2exe** (http://www.py2exe.org/) — работает через **distutils/setuptools**, создает исполняемые файлы под Windows;
- **py2app** (https://py2app.readthedocs.io/en/latest/) — работает через **distutils/setuptools**, создает приложения под Mac OS X;
- **pyqtdeploy** (http://pyqt.sourceforge.net/Docs/pyqtdeploy/tutorial.html) — GUI-утилита для сборки PyQt-приложений (потребуется утилита **nmake** из комплекта Microsoft Visual Studio Build Tools).

Все эти проекты немного отличаются в использовании и имеют разные ограничения. Перед выбором конкретного инструмента необходимо определить конечную платформу, потому что каждый из них поддерживает определенный набор операционных систем.

Оптимальный подход — выбрать один из этих инструментов в самом начале проекта. Ни одна из утилит не выполняет глубокого взаимодействия с исходным кодом. Но если отложить процесс сборки на финальную стадию проекта, можно оказаться в ситуации, когда ни один из инструментов не сможет работать. Тогда сборка исполняемого файла может отнять много сил и времени.

Рассмотрим подробнее **cx_Freeze**.

cx_Freeze

cx_Freeze не позволяет выполнять кросс-компиляцию — исполняемый файл можно собирать только на той платформе, для которой он создается.

cx_Freeze создает дистрибутив, который содержит не просто исполняемый файл, но и необходимые библиотеки (например, в виде DLL-файлов).

Установка стандартная:

```
pip install cx_freeze
```

Для Windows лучше скачать последнюю сборку https://pypi.python.org/pypi/cx_Freeze, после чего выполнить команду установки:

```
pip install <имя_файла.whl>
```

После установки под Windows необходимо проверить наличие команды **cxfreeze** в консоли:

```
> cxfreeze
```

Если утилита доступна, будет выведено следующее:

```
Usage: cxfreeze [options] [SCRIPT]
Freeze a Python script and all of its referenced modules to a base
executable which can then be distributed without requiring a Python installation.
cxfreeze: error: script or a list of modules must be specified
```

При отсутствии команды **cxfreeze** необходимо выполнить постустановочный скрипт **cxfreeze-postinstall**, обычно располагающийся в поддиректории **Scripts** директории установки Python:

```
python3 C:\Python\Scripts\cxfreeze-postinstall
```

Скрипт создаст файл **Scripts\cxfreeze.bat** — команда **cxfreeze** станет доступна в консоли.

Для примера создадим простое приложение Hello World — **simple_hello_world.py** (листинг 2):

```
print('Simple Hello World!')
input()
```

Дальнейшее использование **cx_Freeze** достаточно просто:

```
cxfreeze simple_hello_world.py
```

В результате будет создано следующее дерево файлов (приводится в сокращенном варианте):

```
dist
│   python36.dll
│   simple_hello_world.exe  │
└───lib
    │   library.zip
    │   pyexpat.pyd
    │   select.pyd
    │   unicodedata.pyd
    │   _bz2.pyd
    │   _hashlib.pyd
    │   _lzma.pyd
    │   _socket.pyd
    │   _ssl.pyd
    ...
```

cx_Freeze расширяет пакет **distutils**. Поэтому можно управлять созданием сборки через файл **setup.py**, что делает **cx_Freeze** очень удобным, если процесс распространения уже построен с использованием **setuptools/distutils**.

Пример файла **setup.py** для использования с **cx_Freeze** под Windows (см. **листинг 3**):

```
import sys
from cx_Freeze import setup, Executable

# Зависимости определяются автоматически, но может потребоваться настройка.
build_exe_options = {"packages": ["os"], "excludes": ["tkinter"]}
setup(
```

```
  name="myscript",
  version="0.0.1",
  description="My Hello World application!",
  options={
    "build_exe": build_exe_options
  },
  executables=[Executable("myscript.py")]
)
```

С таким файлом исполняемый файл можно получить новой командой **build_exe**, которая добавлена в скрипт **setup.py**:

```
python3 setup.py build_exe
```

Исполняемые файлы и безопасность

Важно понимать, что исполняемый файл не делает код приложения безопасным (во всех аспектах). Есть возможность декомпилировать встроенный код из исполняемого файла. Результаты подобной декомпиляции позволяют получить код, похожий на изначальный.

Поэтому исполняемые файлы не могут быть полноценным решением для проектов с закрытым исходным кодом (особенно когда важно сохранить закрытость).

Усложнение декомпиляции

Усложнение **не подразумевает** полное предотвращение декомпиляции. Часто чем сложнее задача, тем интереснее ее решать.

Идеи по усложнению декомпиляции:

- удаление всех метаданных, доступных во время выполнения кода (строки документации);
- модификация байт-кода;
- использование модифицированной версии CPython (это потребует декомпилировать еще и CPython);
- использование обфускации исходных кодов перед упаковкой их в исполняемый файл (код останется плохо читаемым и после декомпиляции).

Эти подходы усложнят процесс разработки. Некоторые из них подразумевают глубокое понимание «внутренностей» Python. Если какой-то подход по усложнению декомпиляции будет однажды вскрыт, надо будет тратить время и ресурсы на его обновление.

Поэтому более действенный способ — закрыть конечному пользователю доступ к исходным кодам (предоставлять сервис, а не приложение).

Публикация дистрибутива

После завершения разработки Python-проекта можно сделать его доступным для других разработчиков и пользователей — разместить на **PyPi**. Это сервис-репозиторий для сторонних Python-проектов, которые устанавливаются с помощью команды **pip**.

Публикация нового Python-пакета на **PyPi** осуществляется в несколько шагов:

1. Подготовка файла с описанием пакета. Это файл **setup.py**, содержащий описание проекта: название, версию, зависимости, различные технические параметры. Например, такой файл может выглядеть следующим образом:

```
from setuptools import setup, find_packages

setup(
    name="new-package",
    version="1.0.0",
    packages=find_packages(),
    author="Ivanov I.I.",
    author_email="iii@google.com",
    url="https://github.com/iii/new-package",
)
```

2. Подготовка пакета к публикации. Чтобы пакет был доступен для импорта из других модулей, необходимо запустить команду:

```
pip install --editable
```

3. Отправка пакета в **PyPi**:

   a. Создание аккаунта на сайте сервиса **PyPi** и указание его реквизитов в файле **~/.pypirc**:

```
[distutils]
index-servers = pypi

[pypi]
repository = https://pypi.python.org/pypi
username = <имя пользователя>
password = <пароль>
```

   b. Непосредственная загрузка пакета:

```
python setup.py register
python setup.py sdist upload
```

Первая команда отвечает за регистрацию пакета в сервисе **PyPi**, объединяет файлы с исходным кодом в архив. Вторая — отправляет созданный архив в репозиторий.

Итоги

За рамками данного курса остались некоторые темы углубленного изучения Python — например, создание расширений на языке C, работа с байт-кодом и другие. Надеемся, курс был полезен для слушателей, и он поможет дальнейшему росту и развитию Python-программистов.

Практическое задание

1. Для разработанного проекта «Мессенджер» сформировать whl-пакеты с дистрибутивами сервера и клиента.

2. * Выполнить процедуру сборки созданного проекта «Мессенджер» с помощью утилиты **cx_Freeze**.

3. Выполнить загрузку сформированных whl-пакетов с дистрибутивами сервера и клиента в репозиторий сервиса PyPi.

4. В качестве защиты курсового проекта необходимо записать в любой удобной для вас программе видеоролик (скринкаст) продолжительностью 1-5 минут. Представьте, что вам необходимо презентовать вашу работу заказчику или аудитории. В скринкасте расскажите о вашем проекте, продемонстрируйте его возможности и функционал. Ссылку на видео приложите к практическому заданию, например, в комментарии к уроку. И не забудьте открыть доступ на просмотр! :) Видеопрезентация продукта развивает у вас дополнительные мягкие навыки и является обязательной для засчитывания курсового проекта.

Дополнительные материалы

1. Welcome to Setuptools' documentation.
2. Getting started with setuptools and setup.py.
3. Создание запускаемых файлов из скриптов на языке Python с помощью cx_Freeze.
4. Python. Компиляция exe-программ в cx_freeze.
5. cx_Freeze + virtualenv = баги и зигзаги.
6. Компиляция программы на Python 3 в exe с помощью программы cx_Freeze.
7. Компиляция в .exe Python программы.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. Michael Driscoll. Python 101 (каталог «Дополнительные материалы»).
3. Michał Jaworski, Tarek Ziadé. Expert Python Programming. Second Edition (каталог «Дополнительные материалы»).
4. Бизли Дэвид. Python. Подробный справочник. 4-е издание (каталог «Дополнительные материалы»).

Приложение

Листинг 1

```
# ====================== Распространение приложений
============================
# ---------------------------- Создание пакета
--------------------------------

#   --------------------- Использование setuptools -------------------------

from setuptools import setup

setup(name = "math_package",
      version = "0.1",
      description = "A Simple Math Package",
      author = "Mike Driscoll",
      author_email = "mike@somedomain.com",
      url =
"http://www.blog.pythonlibrary.org/2012/07/08/python-201-creating-modules-and-pa
ckages/",
      packages=["src"]
      )
```

Листинг 2

```
# ====================== Распространение приложений
============================
# --------------------- Создание исполняемого файла
---------------------------

#   ---------------------- Использование cx_Freeze -------------------------

print('Simple Hello World!')
input()
```

Листинг 3

```
import sys
from cx_Freeze import setup, Executable

# Зависимости определяются автоматически, но может потребоваться некоторая
настройка.
build_exe_options = {"packages": ["os"], "excludes": ["tkinter"]}
setup(
  name="simple_hello_world",
  version="0.0.1",
  description="Simple Hello World application!",
  options={
    "build_exe": build_exe_options
  },
```

```
  executables=[Executable("simple_hello_world.py")]
)
```