LECTURE 21

Distributing Python Applications

DISTRIBUTING PYTHON APPLICATIONS

So, now that you know how to build some Python applications, the first thing you'll want to do is build some fantastic Python application or library and distribute it for general use. Depending on your application or audience, this may or may not be a little painful. Generally, you'd want to distribute your Python code under one of two circumstances:

- Distributing a useful open-source library.
- Distributing a useful Python application for a general user (sometimes, preferably, without the source being available to the user). In other words, "commercial" distribution.

PYTHON DISTRIBUTIONS

You should already be familiar with at least one of the various Python package managers that connect to the Python Package Index – the official third-party repository of open-source Python packages.

- pip
- easy_install
- PyPM
- conda

Package managers, operating locally, connect to PyPI to install, update, uninstall (etc) Python distributions.

PYTHON DISTRIBUTION = PYTHON PACKAGE

Python distributions, also called Python packages, are basically directories with a top-level __init__.py file and whatever other source files are needed. A typical package structure might look like following:

```
ticket
|-- __init__.py
|-- exception.py
|-- model.py
|-- ticket.py
|-- test
|   |-- models.py
|   |-- test_ticket.py
```

Our old ticket scraper __init__.py:

```
from flask import Flask
my_app=Flask(__name__)
my_app.config.from_object('config')
from app import views
```

PYTHON PACKAGE STRUCTURE

More generally, your project should include a top-level directory which houses your actual application as well as some top-level files like requirements.txt and the documentation subfolder.

- License and README files.
- A docs subfolder for generated documentation (i.e. Sphinx).
- A requirements.txt for virtualenv setup.
- And *setup.py* for distribution.

```
|- LICENSE
|- README.md
|- docs
|    |-- conf.py
|    |-- generated
|    |-- index.rst
|    |-- installation.rst
|    |-- modules.rst
|    |-- quickstart.rst
|    |-- ticket.rst
|- requirements.txt
|- ticket
|    |-- __init__.py
|    |-- exception.py
|    |-- model.py
|    |-- ticket.py
|    |-- test
|         |-- models.py
|         |-- test_ticket.py
|- setup.py
```

SETUP.PY

As of right now, distutils is the standard packaging tool included in the Python standard library. The setuptools library, however, is the preferred tool over distutils and should be imported within setup.py to specify metadata about the project. The simplest setuptools-driven setup.py looks like this:

```
from setuptools import setup

setup(
    name = "ticket",
    version = "0.1",
    packages = ['ticket']
)
```

SETUP.PY

Of course, we can provide more metadata about the project (and we definitely should).

Here's a more comprehensive example.

```
from setuptools import setup

setup(
    name='ticket',
    version='0.1',
    description='ticket scraper library',
    classifiers=[
        'Development Status :: 3 - Alpha',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 2.7',
        'Topic :: Text Processing :: Linguistic',
    ],
    keywords='ticket scraping taylor swift',
    url='http://github.com/blah/blah',
    author='CIS4930',
    author_email='cis4930@example.com',
    license='MIT',
    packages=['ticket'],
    install_requires=[ 'lxml', 'requests', ]
)
```

SETUP.PY

- Distributing Python Modules
- Setuptools documentation

There are a ton of setup.py options supported by setuptools, so what you need will depend on the application. Check out the above resources for more specific information. Additionally, when including static contents (as in the the /static folder of a Flask application), include a MANIFEST.in file at the top level. The contents might simply be:

```
recursive-include app/static *
```

DISTRIBUTE!

1. Create the distribution file. The sdist option creates source distribution file (a .tar.gz will magically appear inside of a dist directory).

```
$ python setup.py sdist
```

2. Installing the application. The install option installs everything from build directory.

```
$ python setup.py install
```

3. Distributing the application. The register option registers the distribution with the Python package index.

```
$ python setup.py register
```

4. Upload registered package with login.

```
$ python setup.py sdist upload
```

UPDATING DISTRIBUTIONS

Just update the version in setup.py, create a CHANGES.txt at the top level, create new source distribution and re-register with login.

Couldn’t be simpler.

COMMERCIAL DISTRIBUTION

So distributing Python in the jolly world of open-source third-party libraries is actually quite easy (if not laden with great responsibility).

Distributing a Python application commercially (i.e. where an .exe is typically used) is not as painless.

First, we'll review some fundamentals.

COMPILATION VS INTERPRETATION

Programs written in a "compiled" language, like C, are compiled into an *executable* written in machine language for a target machine. This separates the processes of building and running the program.

```
$ gcc hello.c
$ ./a.out
```

→ compiling the program into machine code.

→ running the executable machine code.

As long as the executable is in a compatible format, I can distribute my executable file a.out to any other machine and it can be run separate from the building process.

For real commercial applications, the process is a bit more complicated but the general idea is the same – create the executable as a one-time deal and distribute with minimal worry about exposing source code.

COMPILATION VS INTERPRETATION

Python, however, is an interpreted language.

```
$ python hello.py
```

- The program just runs, no compilation phase.
- The program *python* is the software environment that understands python language.
- The program hello.py is executed (interpreted) within the environment.

So, there's no executable to distribute – a Python program requires an environment to interpret the program.

PYTHON INTERPRETATION

Just because Python is interpreted doesn't mean there isn't some compiling involved. You may have noticed some `.pyc` files in your directory. These are bytecode files – the bytecode compiled version of hello.py will be hello.pyc.

This bytecode is the "machine language" of the interpreter, which understands it natively.

The CPython implementation of the Python interpreter runs C code that matches the current operation in the bytecode.

PYTHON INTERPRETATION

```
>>> from dis import dis
>>> co = compile("spam = eggs - 1",
                 "<string>", "exec")
>>> dis(co)
 1           0 LOAD_NAME                0 (eggs)
             3 LOAD_CONST               0 (1)
             6 BINARY_SUBTRACT
             7 STORE_NAME               1 (spam)
            10 LOAD_CONST               1 (None)
            13 RETURN_VALUE
>>>
```

In ceval.c:

```
TARGET(BINARY_SUBTRACT)
    w = POP();
    v = TOP();
    x = PyNumber_Subtract(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
```

This is not a one-way street. You need to move back and forth from bytecode to c code to figure out what's going on.

Example source: http://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/

COMMERCIAL PYTHON

So you might think: I'll just take my .pyc file and distribute that. Most platforms have a decent Python interpreter installed, right? Ha. No.

Besides, they're not exactly impenetrable fortresses of code secrecy. In fact, there are Python libraries for the sole purpose of reverse-engineering Python bytecode.

NAÏVE SOLUTIONS

- Bundle together python.exe with myproj.py and myothermodule.py (and whatever else you need).
  - Some start-up script can run $ python myproj.py.
- Also, you could bundle myothermodule.py and myproj.py (renamed to __main__.py) in myproj.zip.
  - Now, your start-up script can run $ python myproj.zip.
- On Unix, use the #! directive to condense this to $ ./start-myproj.

But these are a lot of “tricks”.

FREEZING

Freezing involves bundling your application with a Python interpreter and shipping the whole thing as an executable. The Python interpreter is custom in that it's only job is to run the frozen bytecode.

- Freeze (ships with Python – only for Unix systems)
- py2exe (for Windows)
- py2app (for Mac)
- cx_Freeze
- pyinstaller

All solutions need MS Visual C++ dll to be installed on target machine, except py2app.

FREEZING

Let's say you create a single python module that simply holds the contents

```
print "Hello, World!"
```

```
$ python freeze.py hello.py
```

This freezes ~75 modules and creates a C file for each one, which can be compiled and linked with 'make' to create ~5MB executable invoked as ./hello.

You can reduce the modules included but that would necessitate knowledge about how Python uses its own standard library. As the complexity of application grows, the more dependencies you're going to have.

NUITKA

Nuitka is a Python compiler that is compatible with Python 2.6-3.4.

It translates the Python into a (very C-ish) C++ program and bundles it all into as much of a single executable as possible.

Accepts ALL of Python (not just a subset of Python) and works with most packages.

In general: `$ nuitka --standalone program.py` → **program.exe**

Kay Hayen's Nuitka Presentation at EuroPython.

CYTHON

As stated before, Cython is basically another language. A weird kind of hybrid C/Python language that is almost – but not quite – a superset of pure Python.

Cython compiles to C code.

Typically Cython is used to create extension modules for use from pure Python programs. It is, however, possible to write a standalone programs in Cython. This is done via embedding the Python interpreter with the --embed option which will cause a .c file to be generated that you can freely compile with whichever compiler you like.

It's not super easy as you need to find the appropriate Python header files on the system, but it can be done.

Example embedding: https://github.com/cython/cython/tree/master/Demos/embed

WHICH OPTION SHOULD YOU CHOOSE?

To make a choice among the available options, you should consider a few things:

- Size – does it matter how big the executable is?
- Complexity of project – you may be limited if your project requires imports that do not work with or are not supported by your chosen compiler/freezer.
- Target platform – the naïve python bundle with a __main__ inside of a zip is pretty much good enough for linux. You'll be more limited with Mac and especially Windows options.

Truthfully, there is no "standard". Pyinstaller and py2exe are pretty popular. Nuitka is up-and-coming and Cython has a PR issue. Of course, the landscape of options is always growing (with bad options getting pruned, of course).

THE DAY OF THE EXE IS UPON US

For more detailed information, watch the talk "The Day of the Exe is Upon Us" by Brandon Rhodes at PyCon '14.

Experiments performed for the talk can be accessed at https://github.com/brandon-rhodes/exe-from-python.