FROM SCRIPT TO PACKAGES

good practices for hassle-free code reuse

WHAT'S THIS TUTORIAL IS ABOUT

- How to make your code usable by someone else

WHO AM I ?

- Contributor to numpy/scipy since 2007
  - Windows, Mac OS X packaging
  - Build/Packaging issues
- Occasional contributor to scons (make-like tool in python), debian

WHAT'S THIS TUTORIAL IS ABOUT

- How to organize your project as it grows
- How to share software effectively
- Set of good practices to foster collaboration (within or outside the lab)
- A few tools that can help along the way
- Python-focused, but a lot of this applies to many projects

A SIMPLE EXAMPLE

THE USUAL STORY

- A set of scripts that barely work together for data munging
- Someone else starts using them
  - tarball snapshot of the directory passed between people
  - everyone starts modifying it
  - few months from there, everybody uses different, incompatible code
  - Someone else may well be yourself 6 months from now
- How to go away from there ?

CODE ORGANIZATION (1)

- Don’t be creative, follow conventions:
  - For a single python file, no configuration, no data file->a module (i.e. one .py file)
  - Anything else: use python packages (i.e. directories with ___init___.py in them)
  - Top directory should generally be named as your project name
- There are exceptions, I won’t talk about them

EXAMPLE

```
Foo               # Top directory (source tree)
Foo/setup.py
Foo/README
# Top python package (“import foo”)
Foo/foo
Foo/foo/__init__.py
Foo/foo/bar.py
```

KEY POINTS

- Use a sensible name for the source tree (Foo-0.1)
- Add a README (free form, text-file) with at least a few word about the purpose of the software
- setup.py will be used for packaging:
  - Use **distutils** to provide basic packaging features
  - Write one for anything that will ever leave your computer
- Don’t use a Lib or a src directory to put your top package into

THE SETUP.PY

```
from distutils.core import setup

setup(name="foo", version="0.1",
      summary="a few words about the package",
      author="John Doe", author_email="john@doe.com",
      maintainer="John Doe",
      maintainer_email="john@doe.com",
      license="BSD", url="http://www.example.com",
      packages=["foo"])
```

WHY A SETUP.PY ?

- Every python developer knows what to do with it
- Document a few metadata which are useful
- Simple setup.py are easy to write, and get you a lot of features:
  - One command install: python setup.py install
  - You can generate tarball, Mac OS X, windows installers

A FEW WORDS ABOUT VCS

- Always use a Version Source Control (VCS) system for your code
  - example: svn, bzr, git, hg (mercurial)
  - don’t use anything else if you don’t have a good reason
  - Python in general seems to go toward hg, the scipy community is generally going to git
- We won’t talk about VCS (another tutorial ?) - just use one
- There is no justification for not using - no exception ever

FROM A PACKAGE TO A PROJECT

RATIONALE

- Package vs project: from one to N persons

- A project usually involved documentation, tests, scheduled releases, etc...

- In many ways, this is about bootstrapping to make other people do the work for you !

DOCUMENTATION

- Two kinds of documentation:
  - API documentation: what a given function/class does
  - “Proper” documentation: usage-oriented, should be the main documentation as the project matures
- To deal with documentation, use sphinx (developed later)

VERSIONING

- Proper versioning is important
  - Everybody can refer to the same code
  - Carry compatibility information
- Again, don’t be creative:
  - Major.Minor.Micro where each field is an int
  - If possible, add a development version (e.g. svn1234, gitaef2345, etc…)

VERSIONING (2)

- Follow convention for version:
  - Backward incompatible change: N.M.L -> N+1.0.0
  - New features: N.M.L -> N.M+1.0
  - Bug fixes: N.M.L -> N.M.L+1

VERSIONING (3)

- Don’t Repeat Yourself (DRY): define your version once and for all
  - Nice in theory, a bit difficult in practice
  - For simple (most ?) cases, define the version in setup.py, generate a simple module from it

VERSIONING (4)

setup.py

```
version_info = [0, 1, 0]
version = ".".join([str(i) for i in version_info])
def generate_version_py():
    fid = open("foo/__dev_version.py", "w")
    try:
        fid.write("version_info = %s\n" % version_info)
        fid.write("version = '%s'\n" % version)
    finally:
        fid.close()
# run everytime setup.py is imported or run
generate_version_py()
if __name__ == "__main__":
    # Run when doing 'python setup.py build|install|
    # etc...'
    setup(...)
```

VERSIONING (5)

foo/__init__.py

```
try:
    from foo.__dev_version import version as __version__
    from foo.__dev_version import version_info as __version_info__
except ImportError:
    __version__ = "nobuilt"
    __version_info__ = ("nobuilt",)
```

VERSIONING (6)

- Key points:
  - version **string** generated from the version **number(s)**
  - write a file inside your package everytime setup.py is called
  - if possible, use a default in your package so that it can still run without being built
  - Generated from setup.py -> can update build revision (git hash, svn revision number, etc...)
  - make your setup.py “importable” (inside __main__)

NUMPY.DISTUTILS

- Extensions to distutils for numpy needs (fortran, custom compiler support, etc...)
- Enable “recursive” setup.py for complex, deeply nested packages

SIMPLE EXAMPLE

setup.py

```
from numpy.distutils.core import setup
from numpy.distutils.misc_util import Configuration

def configuration(parent_package='', top_path=None):
    config = Configuration(None, parent_package, top_path)
    config.add_subpackage("foo")
    return config

if __name__ == "__main__":
    setup(name=NAME, version=VERSION,
          configuration=configuration)
```

Top setup.py: use None

SIMPLE EXAMPLE (2)

foo/setup.py

```
from numpy.distutils.misc_util import Configuration

def configuration(parent_package='', top_path=None):
    config = Configuration("foo", parent_package, top_path)
    config.add_data_files("yo.dat")
    return config
```

Same value as add_subpackage

Relative to this setup.py

FILES CATEGORIES

- Python packages: directory with __init__
  - Every .py files in that directory included (not recursively)
- Installable data files: add_data_files argument
  - Data files installed relatively to package
  - Other options (4 !), none for flexible installation (install path hardcoded)
- Extra distribution files (not installed): MANIFEST.in

FILES CATEGORIES (2)

- MANIFEST.in:
  - glob-like syntax
  - You can put things which are built there (example: doc)

```
include site.cfg.win32
include site.cfg.bdist_wininst
include Changelog
include common.py
recursive-include docs/pdf *
recursive-include docs/html *
```

- Those files are not installed, but included in sdist tarball

DOCUMENTATION

WRITING DOCUMENTATION

- Use sphinx: http://sphinx.pocoo.org/
  - Use reST for writing documentation
  - Generate html, pdf (latex)
  - Can include docstrings
- I won't talk about how to use sphinx - just how to integrate it

SIMPLE EXAMPLE

```
Foo
Foo/setup.py
Foo/doc/
Foo/doc/Makefile                  # Created by sphinx
Foo/doc/src/conf.py                  ...
Foo/doc/src/...
```

- Two issues w.r.t. packaging:
  - How to build the documentation
  - How to integrate the output to your package

BUILDING THE DOC

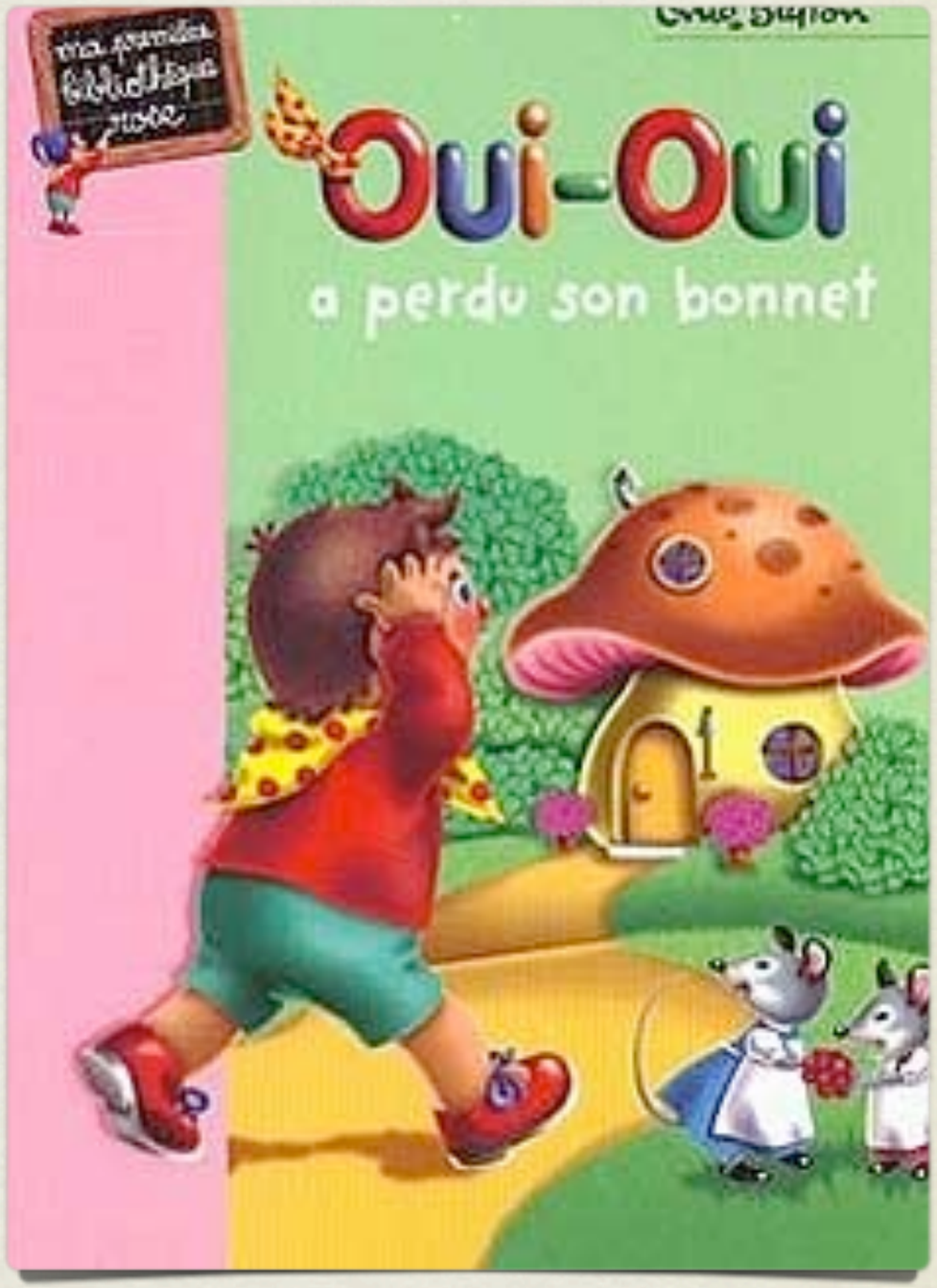
- Sphinx automatically create a makefile for you:
  - make html, make latex + make all-pdf
- But if you include API:
  - docstrings are extracted from whatever is importable by python as executed from sphinx
  - Use virtualenv here

BUILDING DOC

```
#! /bin/bash
virtualenv bootstrap
. bootstrap/bin/activate
# Installing sphinx into the virtualenv is necessary so that
# sphinx-build use the virtualenved python
easy_install sphinx
python setup.py install
(cd doc && make html)
```

DISTUTILS WART

NOT EVERYTHING IS ROSY IN DISTUTILS



DISTUTILS WARTS (1)

- 5 (5 !) ways of installing data files, none of them work well:
  - no flexible installation path (install relatively to package ?)
- Installation command is stupid: copy your build directory
  - You change your modules location, old and new will be installed together
  - Workaround: scratch your build directory

DISTUTILS WARTS (2)

- No automatic dependency handling:
  - half-baked system in numpy.distutils, not reliable
  - workaround: scrap build directory
- Changing compilation options:
  - add_extension can add some flags
  - Want to remove some ? don't even think about it
    - You could write your own compiler class, though

DISTUTILS WARTS (3)

- Extending distutils

WHY EXTENDING ?

- Examples of useful extensions:
  - A flexible data files installation (with overridable path)
  - Build ctypes extensions
  - Add cython support
  - Build your extension with scons/waf/make

THE ISSUE(S)

- Adding new distutils command:
  - You need to call it from another command (say install)
  - So you need to override install command
    - What happens if another package overrides install (setuptools, distribute, something else...)
      - You have to dynamically special case your classes !
- Basically, avoid it as much as you can
  - Writing commands with no interaction with others is fine

NEW DISTUTILS COMMAND

- Use paver for simple commands:
  - http://www.blueskyonmars.com/projects/paver/
  - test, build doc, etc...

```
@task
@needs(['html', "distutils.command.sdist"])
def sdist():
    """Generate docs and source distribution."""
    sh("cd doc && make html")
```

THAT'S ALL FOLKS