autocython

Release 1.1

Mar 24, 2018

Contents

Chris Billington, Mar 24, 2018

View on PyPI | View on BitBucket | Read the docs

CHAPTER 1

Introduction

Like `pyx_import`, but when you want to write your own setup.py still, and you want to keep the compiled extensions in the package directory.

"How is that at all like `pyx_import`?" I hear you ask. "Why would I want that?".

Well, `autocython` records a hash of the `.pyx` files and the resulting `.so` or `.pyd` files whenever it compiles anything, and recompiles automatically (by running your `setup.py`) if it detects that there is a mismatch. So that's how it's like `pyx_import`.

The similarities end there. As mentioned, you have to write your own `setup.py`. I don't see this as much of a drawback, it's rare that I have a Cython extension that doesn't need at least some customisation, and that customisation might as well go in a separate file than in a call to `pyx_import`. See the example below for how to write a setup.py that works with `autocython`.

`autocython` expects you to keep all the compiled extensions in the same directory, even for different versions of Python and platforms. Whilst keeping multiple versions of extensions for different platforms in the same directory is easy in Python 3 (since extensions get a platform-specific suffix), it is less easy in Python 2. So `autocython` provides a platform-specific suffix that you can add to the names of your extensions in your `setup.py` (see example below), and an import function that uses the same suffix to import the right version at run time. This allows distributing fat packages with all the supported compiled versions of the extension in the same folder. Whilst it is bad practice to distribute packages like this, it is often what is most convenient if you're a research group doing numerical simulations or lab control systems and sharing code with each other without wanting to think too hard about packaging or build servers.

But perhaps most importantly, `autocython` serves as a reminder to myself as to the current state of compiling cython extensions on all platform. It calls your `setup.py` as: `python setup.py build_ext --inplace`, or on Windows as `python setup.py build_ext --inplace --compiler=msvc`. So long as your `setup.py` imports `setuptools`, on Windows this means you actually get a meaningful error about where to download the correct compiler from Microsoft. Even if you get these steps wrong, `autocython` prints a big fat error message describing what you need to make sure you've done, which I intend to update whenever the state of Windows compiling changes.

Note: As of March 2018, the compilers needed on Windows are available at: Python 2.7: http://aka.ms/vcpython27.

Python 3.5+:”Microsoft Visual C++ Build Tools”, from http://aka.ms/BuildTools

CHAPTER 2

Installation

to install `autocython`, run:

```
$ pip3 install autocython
```

or to install from source:

```
$ python3 setup.py install
```

Note: Works with Python 2.7 or Python 3.4+

CHAPTER 3

Example usage

Below is an example of how to use autocython in a package. All recompilation is triggered by imports.

Note: You can also check and trigger recompilation by running `python -m autocython` in the directory containing the `.pyx` files and your `setup.py`. This can be a more convenient way to compile for a specific platform than having to actually run the program doing the imports.

This example has a top-level script `example.py` which imports the `hello` function from a package `hello_package`:

```python
# example.py

from hello_package import hello
hello()
```

That package's `__init__.py` imports the `hello` function from a Cython extension `hello_module.pyx` using `autocython`, after ensuring all extensions in the folder are up to date:

```python
# hello_package/__init__.py

import os
from autocython import ensure_extensions_compiled, import_extension
this_folder = os.path.dirname(os.path.abspath(__file__))
ensure_extensions_compiled(this_folder)
hello_module = import_extension('hello_package.hello_module')
hello = hello_module.hello
```

Note that it is important that the import line given to *`import_extension()`* is a fully qualified, absolute import. If you are only using Python 3, *`import_extension()`* is unnecessary and you can just use a normal import line (though you should still use *`import_extension()`* if your code needs to run on both Python 2 and 3)

The Cython extension in the package is:

```
# hello_package/hello_module.pyx

def hello():
    print('hello from cython!')
```

And the package directory also contains the `setup.py` for compiling the extension:

```
# hello_package/setup.py

from setuptools import setup
from setuptools.extension import Extension
from Cython.Distutils import build_ext

from autocython import PLATFORM_SUFFIX

ext_modules = [Extension("hello_module" + PLATFORM_SUFFIX, ["hello_module.pyx"])]
setup(
    name = "hello_package",
    cmdclass = {"build_ext": build_ext},
    ext_modules = ext_modules,
)
```

Use of `setuptools` is crucial on Windows, otherwise compilation will not be able to find the Microsoft compilers. Importing *`PLATFORM_SUFFIX`* and appending it to the extension name allows each version of the extension to have a platform- specific unique name on Python 2 (*`import_extension()`* makes sure it gets the right one at import time). If you are only using Python 3, you don't need to add this suffix, but you still should if your code needs to run on both Pytohn 2 and 3 (in Python 3 *`PLATFORM_SUFFIX`* is just an empty string)

The result of all this is:

```
$ python example.py
Extension(s) out of date, recompiling...
<compilation output>
hello from cython!

$ python example.py # again, no compilation output this time:
hello from cython!

$ python3 example.py # different Python version:
Extension(s) out of date, recompiling...
<compilation output>
hello from cython!

$ python example.py # original Python again, still no recompilation neccesary:
hello from cython!

$ ls hello_package/ # See what files have been generated:
autocython_compile_state.json              hello_module.pyx  __pycache__
hello_module.cpython-36m-x86_64-linux-gnu.so  __init__.py       setup.py
hello_module_py27_linux2_64bit.so          __init__.pyc
```

CHAPTER 4

Limitations

Note: The following limitation only applies to Python 2.

When importing an extension from a package, provided that the package's `__init__.py` uses *`import_extension()`*, then the extension will be available for ordinary import. That is, in the above example, if `example.py` had instead imported the hello function with the line:

```
from hello_package.hello_module import hello
hello()
```

everything would have still been fine. However, importing extensions in the following way does not in general work with `autocython`:

```
import hello_package.hello_module
hello_package.hello_module.hello()
```

This may fail with `AttributeError: 'module' object has no attribute 'hello_module'`, since even though the import succeeded, Python thinks that `hello_package.hello_module` is getting an attribute from the `hello_package`, as opposed to being the name of a submodule. This is a side effect of *`import_extension()`* renaming the extension module after import to remove *`PLATFORM_SUFFIX`*.

The workaround, as done in the above example, is to ensure that `hello_module` *is* an attribute of `hello_package`, by making the import line in your package's `__init__.py` look like:

```
hello_module = import_extension('hello_package.hello_module')
```

with the extension module assigned to a variable with the same name as the extension module itself. If you do this then importers will be able to import the extension module or its members using any of the different forms of the `import` statement.

CHAPTER 5

Module reference

The public API comprises two functions and a constant:

`autocython.`**`ensure_extensions_compiled`**(*folder*, *names=None*)

Ensure the Cython extensions in the given folder with the given list of names are compiled, and if not (or if they are in need of recompilation), compile them by running `setup.py` (assumed to be in the same folder). If no names are given, they will be inferred from any `.pyx` files in the folder. It is assumed that each cython file is called `<name>.pyx`, and that each extension (as specified in `setup.py`) is called `<name><PLATFORM_SUFFIX>`, where *`PLATFORM_SUFFIX`* is a constant defined in this module that specifies the platform details for Python 2, allowing *`import_extension()`* to import the correct version of the extension if multiple versions exist for different platforms. In Python 3 *`PLATFORM_SUFFIX`* is the empty string since Python 3 does a similar thing automatically.

`autocython.`**`import_extension`**(*fullname*)

Import the extension, after appending *`PLATFORM_SUFFIX`* in order to ensure we get the right version for our platform. This is not neccesary on Python 3, which does a similar thing automatically if you use an ordinary import (On Python 3 *`PLATFORM_SUFFIX`* is an empty string). `fullname` must be a fully qualified, absolute import. This function also inserts the module into sys.modules under the name fullname, and hence it will be available for ordinary import without this function, so long as this function is called once first (say in the `__init__.py` of the package)

`autocython.`**`PLATFORM_SUFFIX`**

A platform-specific string that should be appended to extension names in `setup.py` (see above example) in order to make them uniquely named on a per- platform basis in Python 2. On Python 3 this is the empty string. On Python 2 it is `b'_py27_{}_{}'.format(sys.platform, platform.architecture()[0])`, leading to extensions with names like `hello_module_py27_linux2_64bit.so`

Index