pycompilation Documentation

Release 0.3.3

Bjoern Dahlgren

September 29, 2016

Contents

Contents:

CHAPTER 1

pycompilation package

1.1 Submodules

1.2 pycompilation.compilation module

1.2.1 Motivation

Distutils does not allow to use object files in compilation (see http://bugs.python.org/issue5372) hence the compilation of source files cannot be cached unless doing something like what compile_sources / src2obj do.

Distutils does not support fortran out of the box (motivation of numpy distutils), furthermore: linking mixed C++/Fortran use either Fortran (Intel) or C++ (GNU) compiler.

pycompilation.compilation.**any_cplus**(*srcs*)

pycompilation.compilation.**any_fort**(*srcs*)

pycompilation.compilation.**compile_link_import_py_ext**(*srcs*, *extname=None*, *build_dir=None*, *compile_kwargs=None*, *link_kwargs=None*, ***kwargs*)

Compiles sources in *srcs* to a shared object (python extension) which is imported. If shared object is newer than the sources, they are not recompiled but instead it is imported.

Parameters **srcs: string**

list of paths to sources

extname: string

name of extension (default: None) (taken from the last file in *srcs* - without extension)

build_dir: string

path to directory in which objects files etc. are generated

compile_kwargs: dict

keyword arguments passed to compile_sources

link_kwargs: dict

keyword arguments passed to link_py_so

****kwargs:**

additional keyword arguments overwrites to both compile_kwargs and link_kwargs useful for convenience e.g. when passing logger

Returns the imported module

Examples

```
>>> mod = compile_link_import_py_ext(['fft.f90', 'convolution.cpp',         'fft_wrapper.pyx'], c
>>> Aprim = mod.fft(A)
```

pycompilation.compilation.**compile_link_import_strings**(*codes*, *build_dir=None*, ***kwargs*)

Creates a temporary directory and dumps, compiles and links provided source code.

Parameters **codes: iterable of name/source pair tuples**

build_dir: string (default: None)

path to cache_dir. None implies use a temporary directory.

****kwargs:**

keyword arguments passed onto *compile_link_import_py_ext*

pycompilation.compilation.**compile_sources**(*files*, *CompilerRunner_=None*, *destdir=None*, *cwd=None*, *keep_dir_struct=False*, *per_file_kwargs=None*, ***kwargs*)

Compile source code files to object files.

Parameters **files: iterable of path strings**

source files, if cwd is given, the paths are taken as relative.

CompilerRunner_: CompilerRunner instance (optional)

could be e.g. pycompilation.FortranCompilerRunner Will be inferred from filename extensions if missing.

destdir: path string

output directory, if cwd is given, the path is taken as relative

cwd: path string

working directory. Specify to have compiler run in other directory. also used as root of relative paths.

keep_dir_struct: bool

Reproduce directory structure in *destdir*. default: False

per_file_kwargs: dict

dict mapping instances in *files* to keyword arguments

****kwargs: dict**

default keyword arguments to pass to **CompilerRunner_**

pycompilation.compilation.**get_mixed_fort_c_linker**(*vendor=None*, *metadir=None*, *cplus=False*, *cwd=None*)

pycompilation.compilation.**link**(*obj_files*, *out_file=None*, *shared=False*, *CompilerRunner_=None*, *cwd=None*, *cplus=False*, *fort=False*, ***kwargs*)

Link object files.

Parameters **obj_files: iterable of path strings**

out_file: path string (optional)

path to executable/shared library, if missing it will be deduced from the last item in obj_files.

shared: bool

Generate a shared library? default: False

CompilerRunner_: pycompilation.CompilerRunner subclass (optional)

If not given the *cplus* and *fort* flags will be inspected (fallback is the C compiler)

cwd: path string

root of relative paths and working directory for compiler

cplus: bool

C++ objects? default: False

fort: bool

Fortran objects? default: False

****kwargs: dict**

keyword arguments passed onto **CompilerRunner_**

Returns The absolute to the generated shared object / executable

`pycompilation.compilation.`**`link_py_so`**(*obj_files*, *so_file=None*, *cwd=None*, *libraries=None*, *cplus=False*, *fort=False*, ***kwargs*)

Link python extension module (shared object) for importing

Parameters **obj_files: iterable of path strings**

object files to be linked

so_file: path string

Name (path) of shared object file to create. If not specified it will have the basname of the last object file in *obj_files* but with the extension '.so' (Unix) or '.dll' (Windows).

cwd: path string

root of relative paths and working directory of linker.

libraries: iterable of strings

libraries to link against, e.g. ['m']

cplus: bool

Any C++ objects? default: False

fort: bool

Any Fortran objects? default: False

kwargs: dict**

keyword arguments passed onto *link(...)*

Returns Absolute path to the generate shared object

pycompilation.compilation.**pyx2obj**(*pyxpath*, *objpath=None*, *interm_c_dir=None*, *cwd=None*, *logger=None*, *full_module_name=None*, *only_update=False*, *metadir=None*, *include_numpy=False*, *include_dirs=None*, *cy_kwargs=None*, *gdb=False*, *cplus=None*, ***kwargs*)

Convenience function

If cwd is specified, pyxpath and dst are taken to be relative If only_update is set to *True* the modification time is checked and compilation is only run if the source is newer than the destination

Parameters **pyxpath: path string**

path to Cython source file

objpath: path string (optional)

path to object file to generate

interm_c_dir: path string (optional)

directory to put generated C file.

cwd: path string (optional)

working directory and root of relative paths

logger: logging.Logger (optional)

passed onto *simple_cythonize* and *src2obj*

full_module_name: string (optional)

passed onto *simple_cythonize*

only_update: bool (optional)

passed onto *simple_cythonize* and *src2obj*

metadir: path string (optional)

passed onto src2obj

include_numpy: bool (optional)

Add numpy include directory to include_dirs. default: False

include_dirs: iterable of path strings (optional)

Passed onto src2obj and via cy_kwargs['include_path'] to simple_cythonize.

cy_kwargs: dict (optional)

keyword arguments passed onto *simple_cythonize*

gdb: bool (optional)

convenience: cy_kwargs['gdb_debug'] is set True if gdb=True, default: False

cplus: bool (optional)

Indicate whether C++ is used. default: auto-detect using *pyx_is_cplus*

****kwargs: dict**

keyword arguments passed onto src2obj

Returns Absolute path of generated object file.

pycompilation.compilation.**simple_cythonize**(*src*, *destdir=None*, *cwd=None*, *logger=None*, *full_module_name=None*, *only_update=False*, ***cy_kwargs*)

Generates a C file from a Cython source file.

Parameters **src: path string**

path to Cython source

destdir: path string (optional)

Path to output directory (default: '.')

cwd: path string (optional)

Root of relative paths (default: '.')

logger: logging.Logger

info level used.

full_module_name: string

passed to cy_compile (default: None)

only_update: bool

Only cythonize if source is newer. default: False

****cy_kwargs:**

second argument passed to cy_compile. Generates a .cpp file is cplus=True in cy_kwargs, else a .c file.

pycompilation.compilation.**src2obj**(*srcpath*, *CompilerRunner_=None*, *objpath=None*, *only_update=False*, *cwd=None*, *out_ext=None*, *inc_py=False*, ***kwargs*)

Compiles a source code file to an object file. Files ending with '.pyx' assumed to be cython files and are dispatched to pyx2obj.

Parameters **srcpath: path string**

path to source file

CompilerRunner_: pycompilation.CompilerRunner subclass (optional)

Default: deduced from extension of srcpath

objpath: path string (optional)

path to generated object. defualt: deduced from srcpath

only_update: bool

only compile if source is newer than objpath. default: False

cwd: path string (optional)

working directory and root of relative paths. default: current dir.

out_ext: string

set when objpath is a dir and you want to override defaults ('.o'/'.obj' for Unix/Windows).

inc_py: bool

add Python include path to include_dirs. default: False

****kwargs: dict**

keyword arguments passed onto **CompilerRunner_** or pyx2obj

1.3 pycompilation.util module

class pycompilation.util.**ArbitraryDepthGlob**
Bases: tuple

Attributes

Methods

filename
Alias for field number 0

exception pycompilation.util.**CompilationError**
Bases: exceptions.Exception

exception pycompilation.util.**FileNotFoundError**
Bases: exceptions.Exception

class pycompilation.util.**Glob**
Bases: tuple

Attributes

Methods

pathname
Alias for field number 0

class pycompilation.util.**HasMetaData**
Bases: object

Provides convenice classmethods for a class to pickle some metadata.

Methods

classmethod get_from_metadata_file(*dirpath*, *key*)
Get value of key in metadata file dict.

metadata_filename = u'.metadata'

classmethod save_to_metadata_file(*dirpath*, *key*, *value*)
Store *key: value* in metadata file dict.

pycompilation.util.**MetaReaderWriter**(*filename*)

pycompilation.util.**copy**(*src*, *dst*, *only_update=False*, *copystat=True*, *cwd=None*, *dest_is_dir=False*, *create_dest_dirs=False*, *logger=None*)
Augmented shutil.copy with extra options and slightly modified behaviour

Parameters src: string

path to source file

dst: string

path to destingation

only_update: bool

only copy if source is newer than destination (returns None if it was newer), default: False

copystat: bool

See shutil.copystat. default: True

cwd: string

Path to working directory (root of relative paths)

dest_is_dir: bool

ensures that dst is treated as a directory. default: False

create_dest_dirs: bool

creates directories if needed.

logger: logging.Looger

debug level info emitted. Passed onto make_dirs.

Returns Path to the copied file.

pycompilation.util.**expand_collection_in_dict**(*d*, *key*, *new_items*, *no_duplicates=True*)

Parameters d: dict

dict in which a key will be inserted/expanded

key: hashable key in d

new_items: iterable d[key] will be extended with items in new_items

no_duplicates: bool avoid inserting duplicates in d[key] (default: True)

pycompilation.util.**find_binary_of_command**(*candidates*)

Calls *find_executable* from distuils for provided candidates and returns first hit. If no candidate mathces, a RuntimeError is raised

pycompilation.util.**get_abspath**(*path*, *cwd=None*)

pycompilation.util.**glob_at_depth**(*filename_glob*, *cwd=None*)

pycompilation.util.**import_module_from_file**(*filename*, *only_if_newer_than=None*)

Imports (cython generated) shared object file (.so)

Provide a list of paths in *only_if_newer_than* to check timestamps of dependencies. **import_** raises an ImportError if any is newer.

Word of warning: Python's caching or the OS caching (unclear to author) is horrible for reimporting same path of an .so file. It will not detect the new time stamp nor new checksum but will use old module.

Use unique names for this reason.

Parameters **filename: string**

path to shared object

only_if_newer_than: iterable of strings

paths to dependencies of the shared object

Raises **ImportError if any of the files specified in only_if_newer_than are newer than the file given by filename.**

pycompilation.util.**make_dirs**(*path*, *logger=None*)

pycompilation.util.**md5_of_file**(*path*, *nblocks=128*)

Computes the md5 hash of a file.

Parameters **path: string**

path to file to compute hash of

Returns hashlib md5 hash object. Use .digest() or .hexdigest()

on returned object to get binary or hex encoded string.

pycompilation.util.**md5_of_string**(*string*)

pycompilation.util.**missing_or_other_newer**(*path*, *other_path*, *cwd=None*)

Investigate if path is non-existant or older than provided reference path.

Parameters **path: string**

path to path which might be missing or too old

other_path: string

reference path

cwd: string

working directory (root of relative paths)

Returns True if path is older or missing.

pycompilation.util.**pyx_is_cplus**(*path*)

Inspect a Cython source file (.pyx) and look for comment line like:

distutils: language = c++

Returns True if such a file is present in the file, else False.

pycompilation.util.**uniquify**(*l*)

Uniquify a list (skip duplicate items).

1.4 Module contents

pycompilation is a package for meta programming. It aims to support multiple compilers: GNU, Intel, PGI.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

p