cygenja's manual

Release 0.3.0

Nikolaj van Omme
Sylvain Arreckx
Dominique Orban

January 18, 2016

Release 0.3

Date January 18, 2016

CHAPTER

ONE

INTRODUCTION

cygenja is a small Python2 library to generate typed source files from Jinja2 source templates. We use it extensively to generate our Cython projects. See *Limitations* to see if this tool is for you.

1.1 What cygenja can do

From a bunch of templated (source) files, **cygenja** can generate several (source) files. The translation part is given to the powerful Jinja2 template engine. **cygenja** is a layer above this template engine and is in charge of dispatching translation rules to the right subdirectories and apply them to the right bunch of files. A file is **only** generated if it is **older** than the template file used to produce it, i.e. a change in a template file triggers a regeneration of the corresponding files [1].

1.2 How cygenja works

Within a *root* directory, you provide some translation rules: each rule is attached to a subdirectory and a file pattern. You can define several rules for one subdirectory. These rules (called *actions* in **cygenja**) are user defined *callbacks*. Once all rules are registered, the **cygenja** engine is given a directory pattern and a file pattern: only the matching rules are triggered. See *Usage* or look at the *Examples* for more.

1.3 Limitations

Here is a small list of limitations [2]. It is of course not exhaustive but it can already give you a hint if this tool is for you or not.

1.3.1 cygenja only parses subdirectories

cygenja can only parse subdirectories from a root directory. This means that it **cannot** generate files located at the root directory level (or outside the root directory).

1.3.2 cygenja generates files in place

Files can only be generated in the same subdirectories as their corresponding templates.

[1] Of course, you can force a file generation.
[2] Most limitations described here can easily be overcome.

1.3.3 Templates and generated files must have different extensions

Templates are identified by specific extensions. The corresponding generated files will be given specific corresponding extensions too. This extension correspondance is defined by the user but both extensions **must** be different. For instance, **.cpd* templated files are transformed into **.pxd* files. Both extensions, *.cpd* and *.pxd* **are** be different.

1.3.4 File patterns: only `fnmatch` patterns

Translation rules can only be applied to files corresponding to fnmatch patterns. While this covers most cases, it might be a limitation for some.

We also use the same kind of file patterns to trigger the translation.

1.3.5 Directory patterns: only `glob` patterns

To select the subdirectory(ies) within which the rules will be applied by **`cygenja`**'s engine, only glob patterns can be used.

1.3.6 Contradictory *actions* are not filtered nor monitored

Nothing prevents you from registering conflicting actions. In this case, only the **first** registered action is guaranteed to be triggered.

1.4 License

`cygenja` is distributed under the GPLv3.

CHAPTER

TWO

INSTALLING CYGENJA

Installing **cygenja** is really easy.

2.1 Dependencies

The only dependency is the Jinja2 library.

If you want to generate the documentation, you need Sphinx and the sphinx_bootstrap_theme (and possibly LaTeX and several sub modules/packages). To generate the documentation, **cygenja** does not need to be installed.

2.2 Installation

Download source files from Github (or clone the repository) and invoke the traditionnal:

```
python setup.py install
```

in a virtual environment or globally with full admin rights.

CHAPTER

THREE

USAGE

We briefly describe the use of **cygenja**. Basically, you register some filters, file extensions and actions before triggering the translation. In the *Examples* section, you can see **cygenja** in action as we detail its use to generate the CySparse library.

3.1 The *Generator* class

The `Generator` class is the main class of the **cygenja** library. It also is the only class you need to interact with. Its constructor is really simple:

```
from cygenja.generator import Generator
...

logger = ...
env = ...
engine = Generator('root_directory', env, logger, True)
```

You provide a `root_directory` directory, a **Jinja2** environment, a *logger* (from the standard `logging` library) and decide if *warnings* must raise *Exception*s or not (default: `False`). The first two arguments are mandatory while the last two are optional. You don't have to provide a logging engine. We describe the root directory a little further in *The root directory*, refer the reader to jinja2.Environment for more about the `env` argument and discuss the two last arguments in the next corresponding subsections.

3.1.1 Logging

A logging engine *can* be used but is not mandatory. If you don't want to log **cygenja**'s behavior, simply pass *None* as the value of the logger argument in the constructor (this is the default). The logging engine is an object from Python's logging library.

```
from cygenja.generator import Generator
import logging

logger = logging.getLogger('Logger name')
engine = Generator('root_directory', logger, True)
```

Three message logging helpers are provided:

```
def log_info(self, msg)
def log_warning(self, msg)
def log_error(self, msg)
```

Their names and signatures are quite self-explanatory.

3.1.2 Raising exceptions on *Warnings*

Errors **always** trigger *RuntimeError*s while warnings may or may not trigger *RuntimeError*s. To raise exceptions on warnings, set the `raise_exception_on_warning` to `True` in the constructor:

```
engine = Generator('root_directory', logger=logger, raise_exception_on_warning=True)
```

By default, `raise_exception_on_warning` is set to `False`.

3.2 Patterns

There are only **two** types of patterns:

- fnmatch patterns for file names and
- glob patterns for directory names.

This is a general rule for the whole library. When you register an action though, you must provide a directory name, **not** a directory name pattern.

We encourage the reader to (re)read the specifications of these two libraries.

3.3 The *root* directory

The root directory is really the main working directory: all file generations can **only** be done inside **subdirectories** of this directory.

This is so important that it is worth a warning:

> **Warning:** File generations can **only** be done inside **subdirectories** of the *root* directory.

This directory is given as the first parameter of `Generator`'s constructor and can be absolute or relative. At any moment, you can retrieve this directory as an absolute path:

```
engine = Generator('root_directory', ...)

absolute_root_directory = engine.root_directory()
```

3.4 Filters

Filters are simply **`Jinja2`** filters. These filters are *registered*:

```
def my_jinja2_filter(filter_argument):
    ...
    return filter_result

engine = Generator(...)
engine.register_filter('my_filter_name', my_jinja2_filter)
```

where `'my_filter_name'` is the name of the filter used inside your **`Jinja2`** template files and `my_jinja2_filter` is a reference to the actual filter.

The signature of `register_filter` is:

```
register_filter(self, filter_name, filter_ref, force=False)
```

allowing you to register a new filter under an already existing filter name. If you keep `force` set to `False`, a warning is triggered each time you try to register a new filter under an already existing filter name and this **new** filter is disregarded.

You also can register several filters at once with a dictionary of filters:

```
engine = Generator(...)
filters = { 'f1' : filter1,
            'f2' : filter2}

engine.register_filters(filters, force=False)
```

At any time, you can list the registered filters:

```
engine = Generator(...)
print engine.filters_list()
```

This list also includes predefined **Jinja2** filters (see builtin filter). If you only want the filters you registered, invoke:

```
engine.registered_filters_list()
```

3.5 File extensions

cygenja uses a correspondance table between template files and generated files. This table defines a correspondance between file *extensions*. For instance, to have **.cpd* templates generate **.pxd* files:

```
engine = Generator(...)
engine.register_extension(`.cpd`, `.pxd`)
```

Again, we use a `force` switch to force the redefinition of such a correspondance. By default, this switch is set to `False` and if you try to redefine an association with a given template extension, you will trigger a warning and this new correspondance will be disregarded.

You can use a `dict` to register several extensions at once:

```
engine = Generator(...)
ext_correspondance = { '.cpd' : '.pxd',
                       '.cpx' : 'pyx'}
engine.register_extensions(ext_correspondance, force=False):
```

As with filters, you can retrieve the registered extensions:

```
engine.registered_extensions_list()
```

Files with extensions registered as template file extensions are systematically parsed, i.e. you cannot use these extensions for files that are not templates because **cygenja** will try to parse them. What about generated file extensions? Files with these extensions can peacefully coexist with generated files, i.e. existing files, regardless of their extensions, can coexist with generated files and will not be plagued by **cyjenja**. This means that you can safely delete files: only generated files will be deleted [1].

Note: Only generated files are deleted. You can thus safely delete files with **cygenja**.

---

[1] The user is responsible to not to define a translation rule that overwrites any existing files.

3.6 Actions

Actions (defined in the `GeneratorAction` class) are really the core concept of **cygenja**: an action correspond to a *translation rule*. This translation rule makes a correspondance between a subdirectory and a file pattern and a user callback. Here is the signature of the `register_action` method:

```
def register_action(self, relative_directory, file_pattern, action_function)
```

The `relative_directory` argument holds the name of a relative directory from the *root* directory. The separator is OS dependent. For instance, under linux, you can register the following:

```
engine = Generator(...)

def action_function(...):
    ...
    return ...

engine.register_action('cysparse/sparse/utils', 'find*.cpy', action_function)
```

This means that all files corresponding to the `'find*.cpy'` fnmatch pattern inside the `cysparse/sparse/utils` directory can be dealt with the `action_function`.

Contrary to filters and file extensions, you **cannot** ask for a list of registered actions. But you can ask **cygenja** to perform a *dry* session: **cygenja** outputs what it would normaly do but without taking any action [3].

3.6.1 User callback

The `action_function()` is a user-defined callback without argument. It returns a file suffix with a corresponding **Jinja2** variables dict (this is a simple **Python** `dict`). Let's illustrate this by an example:

```
GENERAL_CONTEXT = {...}
INDEX_TYPES = ['INT32', 'INT64']
ELEMENT_TYPES = ['FLOAT32', 'FLOAT64']

def generate_following_index_and_type():
    """

    """
    for index in INDEX_TYPES:
        GENERAL_CONTEXT['index'] = index
        for type in ELEMENT_TYPES:
            GENERAL_CONTEXT['type'] = type
            yield '_%s_%s' % (index, type), GENERAL_CONTEXT
```

The user-defined callback `generate_following_index_and_type()` doesn't take any input argument and returns the `'_%s_%s'` suffix string together with the variables `dict GENERAL_CONTEXT`. This function allows **cygenja** to create files with this suffix from any matching template file. The `GENERAL_CONTEXT` is given to **Jinja2** for the appropriate translation.

For instance, let's use the `ext_correspondance` extensions `dict` discussed earlier (see *File extensions*):

```
ext_correspondance = { '.cpd' : '.pxd',
                       '.cpx' : 'pyx'}
```

[3] You also have access to the internal `TreeMap` object with the `registered_actions_treemap()` method and thus you have access to all its methods. One interesting method is `to_string()`. It gives you a representation of all involved subdirectories.

Any template file with a `.cpd` or `.cpx` extension will be translated into a `_index_type.pxd` or `_index_type.pyx` file respectively. For instance, the template file `my_template_code_file.cpd` will be translated to:

- `my_template_code_file_INT32_FLOAT32.cpd`
- `my_template_code_file_INT32_FLOAT64.cpd`
- `my_template_code_file_INT64_FLOAT32.cpd`
- `my_template_code_file_INT64_FLOAT64.cpd`

As this function is defined by the user, you have total control on what you want to generate or not. In our example, we redefine `GENERAL_CONTEXT['index']` and `GENERAL_CONTEXT['type']` for each index and element types.

We use generators (`yield`) but you could return a `list` if you prefer.

3.6.2 Incompatible actions

You could register incompatible actions, i.e. register competing actions that would translate a file in different ways. Our approach is to **only** use the first compatible action and to disregard all the other actions, regardless if they could be applied or not. So the order in which you register your actions is important. A file will be dealt with the **first** compatible action found. This is worth a warning:

> **Warning:** A template is translated with the **first** compatible action found and only that action.

3.6.3 Default action

`cygenja` allows to define **one** default action that will be triggered when no other compatible action is found for a given template file that corresponds to a fnmatch pattern:

```
engine = Generator(...)

def default_action():
    return ...

engine.register_default_action('*.*',  default_action)
```

Be careful when defining a default action. This action is applied to **all** template files (corresponding to the **`fnmatch`** pattern) for which no compatible action is found. You might want to prefer declaring explicit actions than relying on this implicit default action. That said, if you have lots of default cases, this default action can be very convenient and avoid lots of unnecessary action declarations.

3.7 File generation

To generate the files from template files, there is only **one** method to invoke: *generate()*. Its signature is:

```
def generate(self, dir_pattern, file_pattern, action_ch='g',
             recursively=False, force=False)
```

`dir_pattern` is a `glob` pattern taken from the root directory and it is **only** used for directories while `file_pattern` is a `fnmatch` pattern taken from all matching directories and is **only** used for files. The `action_ch` is a character that triggers different behaviours:

- `g`: Generate all files that match both directory and file patterns. This is the default behavior.

- `d`: Same as *g* but with doing anything, i.e. dry run.
- `c`: Same as *g* but erasing the generated files instead, i.e. clean.

These actions can be done in a given directory or in all its corresponding subdirectories. To choose between these two options, use the `recursively` switch. Finally, by default, files are only generated if they are outdated, i.e. if they are older than the template they were originated from. You can force the generation with the `force` switch.

CHAPTER

FOUR

EXAMPLES

In this section, we demonstrate the use of **cygenja** to generate the CySparse library.

4.1 Init

We start by creating a **cygenja** engine:

```
from cygenja.generator import Generator
...

# read cysparse.cfg
cysparse_config = ConfigParser.SafeConfigParser()
cysparse_config.read('cysparse.cfg')

# create logger
logger = make_logger(cysparse_config=cysparse_config)

# cygenja engine
current_directory = os.path.dirname(os.path.abspath(__file__))
cygenja_engine = Generator(current_directory, logger=logger)
```

`make_logger` is just a wrapper around a logging logger. This logger is not mandatory but can be quite handy to debug sessions. The `current_directory` can be absolute or relative. In this example, let's say its value is `'cysparse'` [1], the main project directory.

We now define some variables:

```
ELEMENT_TYPES = ['INT32_t', 'INT64_t',
                 'FLOAT32_t', 'FLOAT64_t', 'FLOAT128_t',
                 'COMPLEX64_t', 'COMPLEX128_t', 'COMPLEX256_t']
INDEX_TYPES = ['INT32_t', 'INT64_t']
...

GENERAL_CONTEXT = {
                'type_list': ELEMENT_TYPES,
                'index_list' : INDEX_TYPES,
                'default_index_type' : DEFAULT_INDEX_TYPE,
                'integer_list' : INTEGER_ELEMENT_TYPES,
                'real_list' : REAL_ELEMENT_TYPES,
                'complex_list' : COMPLEX_ELEMENT_TYPES,
                ...
              }
```

[1] Yes, we are well aware that this not what is expected from the code. `os.path.abspath(__file__)` will never only return `cysparse`.

4.2 File extensions

CySparse is written essentially in Cython. We can thus generate four types of files: `.pyx`, `.pxd`, `.pxi` and of course `.py` files. For each type of file, we have defined a corresponding extension for a template file: `.cpx`, `.cpd`, `.cpi` and `cpy`. We register this correspondance like so:

```
# register extensions
cygenja_engine.register_extension('.cpy', '.py')
cygenja_engine.register_extension('.cpx', '.pyx')
cygenja_engine.register_extension('.cpd', '.pxd')
cygenja_engine.register_extension('.cpi', '.pxi')
```

Now, each time **cygenja** will encounter a template `.cpx` file, it will generate one or several corresponding `.pyx` files.

4.3 Filters

Jinja2 filters are essentially functions that take a string as input and return a modified version of this string. Here is an example:

```
def cysparse_type_to_numpy_c_type(cysparse_type):
    """
    Transform a :program:`CySparse` enum type into the corresponding
    :program:`NumPy` C-type.

    For instance:

        INT32_T -> npy_int32

    Args:
        cysparse_type:

    """
    return 'npy_' + str(cysparse_type.lower()[:-2])
```

We keep the same name for the function as the function name itself to register it (this is not mandatory):

```
engine.register_filter('cysparse_type_to_numpy_c_type', cysparse_type_to_numpy_c_type)
```

Now you can use `cysparse_type_to_numpy_c_type()` in your **Jinja2** template [2]:

```
cnp.ndarray[cnp.@index|cysparse_type_to_numpy_c_type@, ndim=1] a_row =
    cnp.PyArray_SimpleNew( 1, dmat, cnp.@index|cysparse_type_to_numpy_enum_type@)
```

4.4 Actions

Before we can register any **cygenja** actions, we need to define some callbacks. Here are a few examples:

```
def single_generation():
    yield '', GENERAL_CONTEXT
```

[2] **CySparse**'s **Jinja2** environment allows us to use variables names like so: `@my_variable@`.

```
def generate_following_only_index():
    GENERAL_CONTEXT['type'] = None
    for index in INDEX_TYPES:
        GENERAL_CONTEXT['index'] = index

        yield '_%s' % index, GENERAL_CONTEXT
```

The first function, `single_generation`, only generates one file without changing its name (the extension will be changed though). The second function, `generate_following_only_index`, is more interesting. It generates one file for each index type. These files all have a suffix `_index` attached to their names (i.e. `_INT32_t`, `_INT64_t`) and the `GENERAL_CONTEXT dict` is changed every time with the corresponding entry `index` updated. Here is a more complex version where we generate files with respect to an index type but also an element type:

```
def generate_following_index_and_element():
    for index in INDEX_TYPES:
        GENERAL_CONTEXT['index'] = index
        for type in ELEMENT_TYPES:
            GENERAL_CONTEXT['type'] = type
            yield '_%s_%s' % (index, type), GENERAL_CONTEXT
```

Because these functions are user-defined, you have total control and can generate any complicated combinations that you like.

Now we can use these callbacks and register them. For instance:

```
engine.register_action('config', '*.*', single_generation)
```

This registers any template file (`'*.*'`) located in `cysparse/config` (linux version) with the user callback `single_generation`.

```
engine.register_action('cysparse/sparse/sparse_utils/generic',
                       'generate_indices.*',
                       generate_following_only_index)
```

This time, we associate template files with the name `generate_indices` inside the subdirectory `cysparse/sparse/sparse_utils/generic` with the `generate_following_only_index` callback [3].

Here, we only associate template files with extension `.cpi` to the `generate_following_index_and_element` callback inside subdirectory `cysparse/sparse/csc_mat_matrices/csc_mat_kernel`:

```
cygenja_engine.register_action('cysparse/sparse/csc_mat_matrices/csc_mat_kernel',
                               '*.cpi',
                               generate_following_index_and_element)
```

You are allowed to define multiple actions for one subdirectory:

```
cygenja_engine.register_action('cysparse/sparse/sparse_utils/generic',
                               'find.*',
                               generate_following_index_and_element)

cygenja_engine.register_action('cysparse/sparse/sparse_utils/generic',
                               'generate_indices.*',
                               generate_following_only_index)
```

Remember that if a template file can be associated with several actions, only the **first** action will be triggered.

[3] Thus the real directory is `cysparse/cysparse/sparse/sparse_utils/generic`.

4.5 File generation

We are now ready to generate some files from some templates. There is only one method to call: `generate`. Its signature is:

```
engine.generate(dir_pattern, file_pattern, action_ch='g', recursively=True, force=False)
```

where `dir_pattern` is a **`glob`** pattern used to match directories and `file_pattern` a **`fnmatch`** pattern taken from all matching directories. This combination allows you to refine your operations with a great flexibility. The `action_ch` argument can be `g` (generate files), `c` (clean or erase files) or `d` (dry run).

This is the beginning of the output **`cygenja`** generates when asked a dry run for **all** file generation:

```
Process file 'config/setup.cpy' with function 'single_generation':
   -> config/setup.py
Process file 'cysparse/sparse/ll_mat.cpx' with function 'single_generation':
   -> cysparse/sparse/ll_mat.pyx
Process file 'cysparse/sparse/csc_mat_matrices/csc_mat.cpx' with
                           function 'generate_following_index_and_element':
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_INT32_t.pyx
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_INT64_t.pyx
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_FLOAT32_t.pyx
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_FLOAT64_t.pyx
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_FLOAT128_t.pyx
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_COMPLEX64_t.pyx
   -> cysparse/sparse/csc_mat_matrices/csc_mat_INT32_t_COMPLEX128_t.pyx
...
```

At the moment of writing, we have 23 registered actions that trigger 492 file generations.

A

D

F

P

R