cpp_cookiecutter

Thorsten Beier

Aug 16, 2019

USAGE

cpp_cookiecutter is cookiecutter template for a modern C++ project with python bindings

CHAPTER

ONE

DEMO PROJECT

Have a look at github.com/DerThorsten/cpptools, an unmodified example project created with this cpp_cookiecutter.

CHAPTER

TWO

LICENSING

This software is licensed under the BSD 3-Clause License. See the LICENSE.txt file for details.

2.1 Basic usage

2.1.1 Install cookiecutter

Install _cookiecutter via conda (recommended)

```
$ conda install cookiecutter -c conda-forge
```

or

```
$ pip install cookiecutter
```

2.1.2 Create Project With Cookiecutter

After installing cookiecutter, use the cpp-cookiecutter:

```
$ cookiecutter https://github.com/DerThorsten/cpp_cookiecutter
```

There you need to specify the variables of the cpp_cookiecutter which are explained in the table below:

Table 1: Title

Cookiecutter variable	Description	Default
full_name	Author Name	John Doe
email	email of author	john@doe.de
project_name	name of the project	cpptools
project_slug	url friendly version of package name	cpptools
package_name	package name	cpptools
conda_package_name	conda package name	cpptools
readthedocs_package_name	readthedocs package name	cpptools
cpp_namespace	name of C++ namespace	cpptools
cpp_root_folder_name	name of the root C++ folder	cpptools
cpp_macro_prefix	prefix for all macros in C++	CPPTOOLS
cpp_standart	which C++ standard should be used	14
cmake_project_name	name of the project within cmake	cpptools
cmake_interface_library_name	name of the cmake interface library	cpptools
github_project_name	name of the project on github	cpptools
python_bindings	should python bindings be included	Yes
python_package_name	name of the python package	cpptools
github_user_name	authors github user name	JohnDoe
dockerhub_user_name	authors dockerhub user name	johndoe
dockerhub_project_name	name for this project on dockerhub	johndoe
azure_user_name	authors user name on microsoft azure	JohnDoe
open_source_license	which license shall be used	MIT LICENCE
summary	a short summary of the project	cpptools is a modern C++ Library
description	a short description of the project	cpptools is a modern C++ Library

2.1.3 Build Generated Project

This cookiecutter is best used in conjunction with conda: Assuming your package is named cpptools the following script shows the usage of the generated project cookiecutter on Linux/MacOS

```
cd cpptools
conda env create -f cpptools-dev-requirements.yml
source activate cpptools-dev-requirements
mkdir build
```

(continues on next page)

(continued from previous page)

```
cd build
cmake ..
make -j2
make cpp-test
make python-test
cd examples
./hello_world
cd ..
cd benchmark
./benchmark_cpptools
```

On a windows machine this looks like:

```
cd cpptools
conda env create -f cpptools-dev-requirements.yml
call activate cpptools-dev-requirements
mkdir build
cd build
cmake .. -G"Visual Studio 15 2017 Win64" -DCMAKE_BUILD_TYPE=Release  ^
      -DDEPENDENCY_SEARCH_PREFIX="%CONDA_PREFIX%\Library" -DCMAKE_PREFIX_PATH="%CONDA_
↪PREFIX%\Library"
call activate cpptools-dev-requirements
cmake --build . --target ALL_BUILD
cmake --build . --target python-test
cmake --build . --target cpp-test
```

2.2 Install dev requirements

To install all dev requirements install the dependencies via the requirements yaml file

```
$ conda env create -f cpp_cookiecutter-dev-requiremnts.yml
```

This will create a fresh conda environments with all dependencies to use the cookiecutter and to build the documentation of this project.

2.3 Folder Structure

The generated project has the following folder structure

```
{{cookiecutter.cmake_project_name}}
   ├──azure-pipelines.yml                                  # Ci script
   │
   ├──benchmark                                            # C++ benchmark code
   │    └── ...
   │
   ├──binder                                               # dockerfile for mybinder.org
   │    └── Dockerfile
   │
   ├──cmake                                                # Cmake script/modules
   │    └── ...
   │
   ├──CMakeLists.txt                                       # Main cmake list
```

(continues on next page)

(continued from previous page)

```
  ├──CONTRIBUTING.rst                                    # Introduction how to␣
↪constribute
  │
  ├──{{cookiecutter.cmake_project_name}}Config.cmake.in # Script to make find_
↪package(...)
  │                                                      # work for this package
  │
  ├──{{cookiecutter.cmake_project_name}}.pc.in           # Packaging info
  │
  ├──{{cookiecutter.package_name}}-dev-requirements.yml # List of development conda␣
↪dependencies
  │
  ├──docker                                               # dockerfile for dockerhub
  │   └── Dockerfile
  │
  ├──docs                                                # Sources for sphinx␣
↪documentation
  │   └── ...
  │
  ├──examples                                            # C++ examples
  │   └── ...
  │
  ├──include                                             # C++ include directory for␣
↪this folder
  │   └── ...
  │
  ├──LICENCE.txt                                         # License file
  │
  ├──python                                              # Python binding source code
  │   └── ...
  │
  ├──README.rst                                          # Readme shown on github
  │
  ├──readthedocs.yml                                     # Entry point for automated
  │                                                      # documentation build on␣
↪readthedocs.org
  │
  ├──recipe                                              # Folder for conda recipes
  │   └── ...
  │
  └──test                                                # Folder containing C++ unit␣
↪tests
      └── ...
```

2.4 Unit Tests

We use doctest to create a benchmark for the C++ code.

The test subfolder contains all the code related to the C++ unit tests. In `main.cpp` implements the benchmarks runner, The unit tets are implemented in `test_*.cpp`. Assuming your project is names `cpptools`, the test folder looks like.

```
project
├── ...
├── test
│   ├── CMakeLists.txt
│   ├── main.cpp
│   ├── test_cpptools_config.cpp
└── ...
```

2.4.1 Build System

There is a meta target called `test_cpptools` (assuming your cpptools is the package name) which bundles the build process of unit tests. Assuming you cmake-build directory is called `bld` the following will build all examples.

```
$ cd bld
$ make test_cpptools
```

To run the actual test you can use the target `cpp_tests`

```
$ cd bld
$ make cpp_tests
```

2.4.2 Adding New Tests

To add new tests just add a new cpp file to the test folder and update the `CMakeLists.txt`. Assuming we named the new cpp file `test_my_new_feture.cpp`, the relevant part in the `CMakeLists.txt` shall look like this:

```
# all tests
set(${PROJECT_NAME}_TESTS
    test_cpptools_config.cpp
    test_my_new_feture.cpp
)
```

After changing the `CMakeLists.txt` cmake needs to be rerun to configure the build again. After that `make examples` will build all examples including the freshly added examples.

```
$ cd bld
$ cmake .
$ make examples
```

2.5 Benchmark

We use gbench to create a benchmark for the C++ code.

The benchmark subfolder contains all the code related to the benchmarks. In main.cpp the actual benchmarks are implemented.

```
project
├── ...
├── benchmark
│   ├── main.cpp
├── ...
```

2.6 Python Module

Here we document how to use and extend the python bindings for a project created with this cookiecutter. See **'cpptools.readthedocs.io/en/latest/python.html https://cpptools.readthedocs.io/en/latest/python.html>'_** for the python documentation of a sample project created with this cookiecutter.

2.6.1 Folder Structure

We use pybind11 to create the python bindings. The python subfolder contains all the code related to the python bindings. The `module/{{cookiecutter.python_package_name}}` subfolder contains all the `*.py` files of the module. The src folder contains the `*.cpp` files used to export the C++ functionality to python via pybind11. The `test` folder contains all python tests.

```
{{cookiecutter.github_project_name}}
├── ...
├── python
│   ├── module
│   │   └── {{cookiecutter.python_package_name}}
│   │       ├── __init__.py
│   │       └── ...
│   ├── src
│   │   ├── CMakeLists.txt
│   │   ├── main.cpp
│   │   ├── def_build_config.cpp
│   │   ├── ...
│   └── test
│       ├── test_build_configuration.py
│       └── ...
└── ...
```

2.6.2 Build System

To build the python package use the `python-module` target.

```
make python-module
```

This will build the `*.cpp` files in the `src` folder and copy the folder `module/{{cookiecutter.python_package_name}}` folder to build location of the python module, namely `${CMAKE_BINARY_DIR}/python/module/` where `${CMAKE_BINARY_DIR}` is the build directory.

2.6.3 Usage

After a successfully building and installing the python module can be imported like the following:

```
import {{cookiecutter.python_package_name}}

config = {{cookiecutter.python_package_name}}.BuildConfiguration
print(config.VERSION_MAJOR)
```

2.6.4 Run Python Tests

To run the python test suite use the *python-test* target:

```
make python-test
```

2.6.5 Adding New Python Functionality

We use pybind11 to export functionality from C++ to Python. pybind11 can create modules from C++ without the use of any `*.py` files. Nevertheless we prefer to have a regular Python package with a proper `__init__.py`. From the `__init__.py` we import all the C++ / pybind11 exported functionality from the build submodule named `_{{cookiecutter.python_package_name}}`. This allows us to add new functionality in different ways:

- new functionality from c++ via pybind11
- new puren python functionality

Add New Python Functionality from C++

To export functionality from C++ to python via pybind11 it is good practice to split functionality in multiple `def_*.cpp` files. This allow for readable code, and parallel builds. To add news functionality we create a new file, for example `def_new_stuff.cpp`.

```
#include "pybind11/pybind11.h"
#include "pybind11/numpy.h"

#include <iostream>
#include <numeric>

#define FORCE_IMPORT_ARRAY
#include "xtensor-python/pyarray.hpp"
#include "xtensor-python/pytensor.hpp"

// our headers
#include "{{cookiecutter.cpp_root_folder_name}}/{{ cookiecutter.package_name}}.hpp"

namespace py = pybind11;


namespace {{cookiecutter.cpp_namespace}} {

    void def_new_stuff(py::module & m)
    {
        py::def('new_stuff',[](xt::pytensor<1,double> values){
            return values * 42.0;
        });
    }

}
```

Next we need to declare and call the `def_new_stuff` from `main.cpp`. To declare the function modify the following block in `main.cpp`

```
namespace {{cookiecutter.cpp_namespace}} {

    // ....
    // ....
    // ....
```

(continues on next page)

(continued from previous page)

```
    // implementation in def_myclass.cpp
    void def_class(py::module & m);

    // implementation in def_myclass.cpp
    void def_build_config(py::module & m);

    // implementation in def.cpp
    void def_build_config(py::module & m);

    // implementation in def.cpp
    void def_build_config(py::module & m);

    // implementation in def_new_stuff.cpp
    void def_new_stuff(py::module & m);             // <- our new functionality

}
```

After declaring the function `def_new_stuff`, we can call `def_new_stuff`. We modify the `PYBIND11_MODULE` in code:*main.cpp*:

```
// Python Module and Docstrings
PYBIND11_MODULE(_{{cookiecutter.python_package_name}} , module)
{
    xt::import_numpy();

    module.doc() = R"pbdoc(
        _{{cookiecutter.python_package_name}}  python bindings

        .. currentmodule:: _{{cookiecutter.python_package_name}}

        .. autosummary::
           :toctree: _generate

           BuildConfiguration
           MyClass
           new_stuff
    )pbdoc";

    {{cookiecutter.cpp_namespace}}::def_build_config(module);
    {{cookiecutter.cpp_namespace}}::def_class(module);
    {{cookiecutter.cpp_namespace}}::def_new_stuff(module);  // <- our new␣
↪functionality

    // make version string
    std::stringstream ss;
    ss<<{{cookiecutter.cpp_macro_prefix}}_VERSION_MAJOR<<"."
      <<{{cookiecutter.cpp_macro_prefix}}_VERSION_MINOR<<"."
      <<{{cookiecutter.cpp_macro_prefix}}_VERSION_PATCH;
    module.attr("__version__") = ss.str();
}
```

We need to add this file to the `CMakeLists.txt` file at `{cookiecutter.github_project_name}}/python/src/CMakeLists.txt` The file needs to be passed as an argument to the `pybind11_add_module` function.

```
# add the python library
```

(continues on next page)

(continued from previous page)

```
pybind11_add_module(${PY_MOD_LIB_NAME}
    main.cpp
    def_build_config.cpp
    def_myclass.cpp
    def_new_stuff.cpp  # <- our new functionality
)
```

Now we are ready to build the freshly added functionality.

```
make python-test
```

After a successful build we can use the new functionality from python.

```
import numpy as np
import {{cookiecutter.python_package_name}}

{{cookiecutter.python_package_name}}.new_stuff(numpy.arange(5), dtype='float64')
```

Add New Pure Python Functionality

To add new pure Python functionality, just add the desired function / classes to a new `*.py` file and put this file to the `module/{{cookiecutter.python_package_name}}` subfolder. After adding the new file, cmake needs to be rerun since we copy the content `module/{{cookiecutter.python_package_name}}` during the build process.

2.6.6 Adding New Python Tests

We use pytest as python test framework. To add new tests, just add new `test_*.py` files to the test subfolder. To run the actual test use the `python-test` target

```
make python-test
```

2.7 Examples

The examples subfolder contains C++ examples which shall show the usage of the C++ library.

```
project
├── ...
├── examples
│   ├── CMakeLists.txt
│   ├── hello_world.cpp
├── ...
```

2.7.1 Build System

There is a meta target called `examples` which bundles the build process of all cpp files in the folder examples in one target. Assuming your cmake-build directory is called `bld` the following will build all examples.

```
$ cd bld
$ make examples
```

2.7.2 Adding New Examples

To add new examples just add a new cpp file to the example folder and update the `CMakeLists.txt`. Assuming we named the new cpp file `my_new_example.cpp`, the relevant part in the `CMakeLists.txt` shall look like this:

```
# all examples
set(CPP_EXAMPLE_FILES
   hello_world.cpp
   my_new_example.cpp
)
```

After changing the `CMakeLists.txt` cmake needs to be rerun to configure the build again. After that `make examples` will build all examples including the freshly added examples.

```
$ cd bld
$ cmake .
$ make examples
```

2.8 Conda Recipe

The recipe subfolder contains all the code related to the conda recipe

```
project
├── ...
├── recipe
│   ├── bld.bat
│   ├── build.sh
│   ├── meta.tml
├── ...
```

2.9 Changelog

2.9.1 0.4.0

- added ci pipeline scripts in rendered projects

2.9.2 0.5.0

- added documentation on readthedocs.org