robotpy-build

Release 2020.7.0

RobotPy Development Team

Feb 13, 2021

INTRODUCTION

CHAPTER

ONE

ABOUT THIS PROJECT

The RobotPy project maintains Python wrappers around many different C++ libraries to allow its users to use Python for the FIRST Robotics Competition. It was much too work to create/maintain handwritten wrappers around these libraries, and other tools in the python ecosystem did not meet our needs.

robotpy-build was created to automate large parts of the work required to maintain these wrappers. If you have this problem too, we hope that robotpy-build will be useful for you as well.

1.1 Goals

A primary goal of robotpy-build is to make it really simple to define and build python wrappers for C++ code using pybind11. To make that possible, it also bundles these capabilities:

- Manages native/binary C/C++ dependencies via pypi compatible packages
- Autogenerate python wrappers around the native code by parsing C++ header files, including those that contain modern C++ features
- Support extensive customization of the wrappers when the autogeneration fails
- Builds normal python wheels from the generated and handwritten code that can be installed by pip and imported as a normal python package
- robotpy-build projects can depend on native/binary libraries installed by other robotpy-build projects

robotpy-build is intended to be a generally useful tool for any python project that has C/C++ dependencies. If you find that isn't the case, please report a bug on github.

1.2 Non-goals

- robotpy-build built wheels do not currently conform to the manylinux specification as it prohibits dependencies outside of the wheel
- We don't intend to build wrappers around libraries installed on your system

CHAPTER

TWO

INSTALLATION

Note: `robotpy-build` is a build dependency, not a runtime dependency – in otherwords, you only need it to build binary wheels. Once you have a wheel, the wheel is usable without `robotpy-build`

2.1 Supported platforms

`robotpy-build` requires Python 3.6 to be installed. Our integration tests are ran on Ubuntu 18.04, OSX, and Windows.

- Linux and Windows are expected to work in almost any case
- OSX should work, but when depending on shared libraries there are some cases where relinking doesn't work

To compile the generated code created by `robotpy-build`, you must have a modern C++ compiler that is supported by pybind11.

2.2 Install

`robotpy-build` is distributed on pypi and is installable via pip.

On Linux/OSX, you can install with the following command:

```
$ pip3 install robotpy-build
```

On Windows, you can install via:

```
py -3 -m pip install robotpy-build
```

2.3 pybind11

pybind11 is a header-only C++ library that robotpy-build leverages to easily bind together Python and C++ code. All robotpy-build projects are built using a bundled version of pybind11, which is typically a bleeding edge version of pybind11 with custom patches that have not been accepted upstream yet.

Warning: robotpy-build does not use the pybind11 package distributed on pypi.

CHAPTER

THREE

DEMO PROJECT

This walkthrough will take you through all the steps to create a simple robotpy-build project that autogenerates a working wrapper around a C++ class and it's methods.

This demo should work on Linux, OSX, and Windows. Make sure you have robotpy_build installed first!

Note: This demo shows building a python wrapper around C++ code that is self-contained in this project. However, `robotpy-build` also supports wrapping externally compiled libraries and inter-package shared library dependencies.

3.1 Files + descriptions

Note: If you're lazy, the files for this demo are checked into the robotpy-build repository at `examples/demo`.

All of the content required for this demo is contained inline below. Let's start by creating a new directory for your project, and we're going to create the following files:

3.1.1 setup.py

Traditionally python projects required a setup.py that contained all the information and logic needed to compile/install the project. Every project that uses robotpy-build has an identical `setup.py` that looks like this:

```
#!/usr/bin/env python3
from robotpy_build.setup import setup

setup()
```

3.1.2 pyproject.toml

Projects that use robotpy-build must add a `pyproject.toml` to the root of their project as specified in PEP 518. This file is used to configure your project.

Comments describing the function of each section can be found inline below.

```
# This section tells pip to install robotpy-build before starting a build
[build-system]
requires = ["robotpy-build"]

# Tells robotpy-build where to place autogenerated metadata
[tool.robotpy-build]
base_package = "rpydemo"

# This section configures the 'rpydemo' python package. Multiple
# sections are possible to build multiple packages
[tool.robotpy-build.wrappers."rpydemo"]
name = "rpydemo"

# C++ source files to compile, path is relative to the root of the project
sources = [
    "rpydemo/src/demo.cpp",
    "rpydemo/src/main.cpp"
]

# This tells robotpy-build to parse include/demo.h and autogenerate pybind11
# wrappers for the contents of the header.
# -> autogenerate this via `robotpy-build scan-headers`
generate = [
    { demo = "demo.h" }
]

# This is a directory that can be used to customize the autogenerated
# C++ code
# -> autogenerate those files via `robotpy-build create-gen`
generation_data = "gen"

# Standard python package metadata
[tool.robotpy-build.metadata]
name = "robotpy-build-demo"
description = "robotpy-build demo program"
author = "RobotPy Development Team"
author_email = "robotpy@googlegroups.com"
url = "https://github.com/robotpy/robotpy-build"
license = "BSD-3-Clause"
install_requires = []
```

See also:

For detailed information about the contents of `pyproject.toml` see *setup.py and pyproject.toml*.

3.1.3 rpydemo/__init__.py

```
# file is empty for now
```

3.1.4 rpydemo/src/demo.cpp

This is the (very simple) C++ code that we will wrap so that it can be called from python.

```
#include "demo.h"

int add2(int x) {
    return x + 2;
}

void DemoClass::setX(int x) {
    m_x = x;
}

int DemoClass::getX() const {
    return m_x;
}
```

3.1.5 rpydemo/include/demo.h

This is the C++ header file for the code that we're wrapping. In `pyproject.toml` we told `robotpy-build` to parse this file and autogenerate wrappers for it.

For simple C++ code such as this, autogeneration will 'just work' and no other customization is required. However, certain C++ code (templates and sometimes code that depends on templated types, and other complex circumstances) will require providing customization in a YAML file.

```
#pragma once

/** Adds 2 to the first parameter and returns it */
int add2(int x);

/**
    Doxygen documentation is automatically added to your python objects
    when the bindings are autogenerated.
*/
class DemoClass {
public:

    /** Sets X */
    void setX(int x);

    /** Gets X */
    int getX() const;

private:
    int m_x = 0;
};
```

3.1.6 rpydemo/src/main.cpp

Finally, you need to define your pybind11 python module. Custom `pybind11` projects would use a `PYBIND11_MODULE` macro to define a module, but it's easier to use the `RPYBUILD_PYBIND11_MODULE` macro which automatically sets the module name when robotpy-build compiles the file.

```
#include <rpygen_wrapper.hpp>

RPYBUILD_PYBIND11_MODULE(m) { initWrapper(m); }
```

Note: If you wanted to add your own handwritten pybind11 code here, you can add it in addition to the `initWrapper` call made here. See the pybind11 documentation for more details.

3.2 Install the project

When developing a new project, it's easiest to just install in 'develop' mode which will build/install everything in the currect directory.

```
$ python3 setup.py develop
```

Note: When using develop mode for robotpy-build projects, it is recommended to invoke setup.py directly instead of using `pip3 install -e`

3.3 Adjust the project

As we've currently built the project, the CPython extension will be built as `rpydemo._rpydemo`. For example:

```
>>> from rpydemo._rpydemo import DemoClass
>>> DemoClass
<class 'rpydemo._rpydemo.DemoClass'>
```

While that works, we really would like users to be able to access our module directly by importing them into `__init__.py`. There's a robotpy_build command for generating the contents of `__init__.py`:

```
$ python -m robotpy_build create-imports rpydemo rpydemo._rpydemo
```

The output from this you can put into your `rpydemo/__init__.py`. It'll look like this:

```
# autogenerated by 'robotpy-build create-imports rpydemo rpydemo._rpydemo'
from ._rpydemo import DemoClass, add2

__all__ = ["DemoClass", "add2"]
```

Now when we put this in our `__init__.py`, that allows this to work instead:

```
>>> from rpydemo import DemoClass
>>> DemoClass
<class 'rpydemo._rpydemo.DemoClass'>
```

3.4 Trying out the project

Alright, now that all the pieces are assembled, we can try out our project:

```
>>> import rpydemo
>>> rpydemo.add2(2)
4
>>> d = rpydemo.DemoClass()
>>> d.setX(2)
>>> d.getX()
2
```

3.5 More Examples

The integration tests in `tests/cpp` contains a python project that contains several autogenerated wrappers packages and various customizations.

CHAPTER

FOUR

AUTOMATED C++ HEADER WRAPPING

robotpy-build can be told to parse C/C++ headers and automatically generate pybind11 wrappers around the functions and objects found in that header.

Note: We use a fork of CppHeaderParser to parse headers. We've improved it to handle many complicated modern C++ features, but if you run into problems please file a bug on github.

4.1 C++ Features

robotpy-build uses a pure python C++ parser and macro processor to attempt to parse header files. As a result, a full AST of the header files is not created. This means particularly opaque code might confuse the parser, as robotpy-build only receives the names, not the actual type information.

However, most basic features typically work without needing to coerce the generator into working correctly, including:

- functions/methods (overloads, static, etc)
- public class variables
- protected class variables are (when possible) exported with a _ prefix
- inheritance - detects and sets up Python object hierarchy automatically
- abstract classes - autogenerated code ensures they cannot be created directly
- virtual functions - automatically generates trampoline classes as described in the pybind11 documentation so that python classes can override them
- final classes/methods - cannot be overridden from Python code
- Enumerations
- Global variables

Additionally, the following features are supported, but require some manual intervention:

- C++ *class* and *function* templates

To tell the autogenerator to parse headers, you need to add a `generate` to your package in `pyproject.toml`:

```
generate = [
  { demo = "demo.h" }
]
```

That causes `demo.h` to be parsed and wrapped.

Note: If you're importing a large number of headers, the `robotpy-build scan-headers` tool can generate this for you automatically.

4.2 Documentation

robotpy-build will find doxygen documentation comments on many types of elements and use sphinxify to translate them into python docstrings. All elements that support documentation strings can have their docstrings set explicitly using a `doc` value in the YAML file.

```
classes:
  X:
    doc: Docstring for this class
```

4.3 Parameters

TODO

4.3.1 Out parameters

TODO

4.4 Conditional compilation

4.5 Class templates

The code generator needs to be told which instantiations of the class template to create. For a given class:

```
template <typename T>
struct TBasic
{
    virtual ~TBasic() {}

    T getT() { return t; }
    virtual void setT(const T &t) { this->t = t; }

    T t;
};
```

You need to tell the code generator two things about your class:

- Identify the template parameters in the class
- Declare explicit instantiations that you wish to expose, and their name

To cause a python class to be created called `TBasicString` which wraps `TBasic<std::string>`:

```yaml
classes:
  TBasic:
    template_params:
    - T

templates:
  TBasicString:
    qualname: TBasic
    params:
    - std::string
```

4.6 Function templates

The code generator needs to be told which instantiations of the function template to create. For a given function:

```
struct TClassWithFn
{
    template <typename T>
    static T getT(T t)
    {
        return t;
    }
};
```

The following would go in your YAML to create overloads callable from python that call `bool getT(bool)` and `int getT(int)`.

```yaml
classes:
  TClassWithFn:
    methods:
      getT:
        template_impls:
        - ["bool"]
        - ["int"]
```

4.7 Differing python and C++ function signatures

Custom configuration of your functions allows you to define a more pythonic API for your C++ classes.

4.7.1 Python only

This often comes up when the python type and a C++ type of a function parameter or return value is different, or you want to omit a parameter. Just define a lambda via `cpp_code`:

```
// original code
int foo(int param1);
```

```yaml
functions:
  foo:
    cpp_code:
```

(continues on next page)

(continued from previous page)

```
      [](int param1) -> std::string {
        return std::to_string(param1);
      }
```

If you change the parameters, then you need to use `param_override` to adjust the parameters. Let's say you wanted to remove 'param2':

```
functions:
  foo:
    param_override:
      param2:
        ignore: true
```

Note: When you change things like this, these inline definitions are *not* callable from C++, you need virtual functions for that.

4.7.2 Python and C++

Let's say that you have a C++ virtual function `void MyClass::foo(std::iostream &s)`. Semantically, it's just returning a string. Because you really don't want to wrap `std::iostream`, you decide that the function should just return a string in python.

Because this is a virtual function, you need to define a `virtual_xform` lambda that will take the original arguments, call the python API, then return the original return type. Then when C++ code calls that virtual function, it will call the xform function which will call your python API.

```
classes:
  MyClass:
    methods:
      foo:
        param_override:
          s:
            ignore: true
        cpp_code: |
          // python API
          [](MyClass * self) -> std::string {
            std::stringstream ss;
            self->foo(ss);
            return ss.str();
          }
        virtual_xform: |
          // C++ virtual function transformer
          [&](py::function &overload) {
            auto s = py::cast<std::string>(overload());
            ss << s;
          }
```

CHAPTER

FIVE

EXTERNAL DEPENDENCIES

5.1 Download files

Used to download files from an arbitrary URL. This can download headers, shared/static libraries, and sources.

```
[[tool.robotpy-build.wrappers."PACKAGENAME".download]]
url = "https://my/url/something.zip"
incdir = "include"
libs = ["mylib"]
libdir = "libpath"
```

That tells robotpy-build:

- To download the zipfile at that URL
- Anything in the "include" directory will be extracted to the include path for the wrapper
- Libraries will be searched in the "libpath" directory (default is "")
- The shared library "mylib" will be linked to the python module being built

5.2 Maven artifacts

Used to download artifacts from a maven repository. This can download headers, shared libraries, and sources.

```
[tool.robotpy-build.wrappers."PACKAGENAME".maven_lib_download]
artifact_id = "mything"
group_id = "com.example.thing"
repo_url = "http://example.com/maven"
version = "1.2.3"
```

When robotpy-build downloads an artifact from a maven repo, it will unpack it into a temporary directory and add the relevant directories to your package's include/library paths. Additionally, any headers will automatically be copied into your wheel so that other packages can utilize them when linking to your wheel.

For development purposes, you can download/extract the files locally by running `python3 setup.py build_dl`.

See also:

Reference for all *MavenLibDownload* options

Note: It's possible that the archive format is specific to FIRST Robotics. If you're trying to use this and it isn't working, file a bug on github!

For FIRST Robotics related projects, the metadata for `maven_lib_download` can be found in a vendor JSON file.

CHAPTER

SIX

TYPE CASTERS

Most of the time, pybind11 is able to figure out how to convert C++ to Python types and back, especially classes that are explicitly wrapped. However, some types (such as STL containers) are converted indirectly and require special "type casters" to be included in your wrapper code.

6.1 Using type casters

When parsing a header, robotpy-build will match the types found against its list of type casters. If found, it will automatically add an include directive for the header that contains the type caster. robotpy-build contains support for the STL type casters that come with pybind11 (vector, function, map, etc).

Sometimes the autodetection will fail because a type is hidden in some way. You can specify the missing types on a per-class basis. For example, if `std::vector` is aliased and your class uses it:

```
classes:
  MyClass:
    force_type_casters:
    - std::vector
```

6.2 Custom type casters

In some rare cases, you may need to create your own custom type casters. We won't cover that here, refer to the pybind11 documentation for custom type casters instead.

However, once you have a custom type caster, here's what you need to do to automatically include it in a robotpy-build project.

1. Create a header file and put the type caster in that header file. It should be in one of your project's python packages.
2. Add a section to `pyproject.toml`

   ```
   [tool.robotpy-build.wrappers."package_name".type_casters]
   "header_name.h" =     ["somens::SomeType"]
   ```

   This directive will cause robotpy-build's autogenerator to automatically include `header_name.h` whenever it notices `SomeType` in a file. This allows pybind11 to use it. The type specification is a list, so if there are multiple types covered in the same header, just specify all the types. However, most of the time it is preferred to put type casters for separate types in separate files.

Warning: Because type casters are included at compile time, if you change a custom type caster in a project you should recompile all dependent projects as well, otherwise undefined behavior may occur.

CHAPTER

SEVEN

PLATFORM-SPECIFIC CONSIDERATIONS

7.1 cross-compilation

We have successfully used crossenv to cross-compile python modules. For an example set of dockerfiles used to compile for the RoboRIO platform, see the robotpy-cross-docker project.

7.2 macOS

macOS handles library files differently than Windows or Linux. In particular, macOS will not automatically find and link libraries at runtime. Rather, libraries need to be explicitly linked beforehand. On Linux/Windows, if your dependency is not specified explcitly, but you import the dependency before importing your package, it often works without complaining. **This is not the case on OSX**.

This presents a special challenge when repackaging libraries to be shipped in wheels, particularly when linking to libraries installed by other wheels. robotpy-build provides automatic relinking support for libraries that are installed via other robotpy-build created wheels, provided that you specify your dependencies explcitly.

External libraries required must always be defined by `depends`, even if they are indirect dependencies.

```
[tool.robotpy-build.wrappers."PACKAGENAME"]
depends = ["ntcore"]
```

If you forget an external dependency, the library won't load and a linking error will occur at runtime when your package is imported.

CHAPTER

EIGHT

DEVELOPER TIPS

Over time, we've collected a number of tips and best practices for working with robotpy-build projects which will save you lots of time – particularly when dealing with very large projects.

8.1 Build using develop mode

For rapid iteration development, we highly recommend using `python3 setup.py develop` to build your project. This allows in-tree editing of python files and leads to faster results if you're disciplined.

8.2 Use parallel builds

While setuptools does not support parallel builds, robotpy-build will compile things in parallel if you define the environment variable `RPYBUILD_PARALLEL=1`.

```
$ RPYBUILD_PARALLEL=1 python3 setup.py develop
```

8.3 Partial code generation

When developing wrappers of very large projects, the wrapper regeneration step can take a very long time. Often you find that you only want to modify a single file. You can define a YAML file and tell robotpy-build to only regenerate the output for that single file, instead of scanning all files and regenerating everything.

Create a filter.yml:

```
# the names in this list correspond to the keys in your 'generate' section
only_generate:
- SendableChooser
```

Then run your build like so:

```
$ RPYBUILD_GEN_FILTER=filter.yml python setup.py develop
```

8.4 Use ccache

Note: ccache does not support Visual Studio

If you create a large robotpy-build based project, you'll notice that your build times can be *really* long. This occurs because robotpy-build uses pybind11 for connecting C++ and Python code, and pybind11 relies heaviliy on C++ templates and other modern C++ techniques which cause build times to take a really long time.

From ccache's website:

> Ccache (or "ccache") is a compiler cache. It speeds up recompilation by caching previous compilations and detecting when the same compilation is being done again.

This can dramatically improve your compile times if you're just changing a small portions of your project.

Once you install ccache, define these environment variables:

```
export RPYBUILD_PARALLEL=1
export CC="ccache gcc"
export CXX="ccache g++"
export GCC_COLORS=1
```

The first one will make robotpy-build compile in parallel. The second tells setuptools to use ccache, and the third makes error output nice when using ccache.

CHAPTER

NINE

TOOLS

robotpy-build has some useful command line tools that can be executed via `python3 -m robotpy_build TOOLNAME`. On OSX/Linux, it is also installed as the `robotpy-build` script which can be executed directly.

9.1 scan-headers

This tool is designed to make it easy to populate the `generate` key for your python package.

This will scan all of your defined includes directories (including those of downloaded artifacts) and output something you can paste into the `generate` key of `pyproject.toml`.

9.2 create-gen

This will parse your defined generate items (`generate` key of `pyproject.toml`) and output yaml for them in the directory defined by `generation_data`. By default it will just print the file contents to stdout.

Use the `--write` argument to write the files, but it won't overwrite existing files.

9.3 create-imports

Given a base package and a compiled module within that package, this will generate contents that contain all of the things exported by the compiled module. Often it is useful to put this in the `__init__.py` of your python package.

```
$ python -m robotpy_build create-imports rpydemo rpydemo._rpydemo
```

CHAPTER

TEN

SETUP.PY AND PYPROJECT.TOML

10.1 setup.py

Projects that use robotpy-build must use the setup function provided by robotpy-build. Your project's setup.py should look like this:

```
#!/usr/bin/env python3
from robotpy_build.setup import setup
setup()
```

10.2 pyproject.toml

Projects that use robotpy-build must add a `pyproject.toml` to the root of their project as specified in PEP 518.

It is recommended that projects include the standard `build-system` section to tell pip to install robotpy-build (and any other dependencies) before starting a build.

```
[build-system]
requires = ["robotpy-build>=2020.1.0,<2021.0.0"]
```

Projects must include robotpy-build specific sections in their pyproject.toml. robotpy-build takes `pyproject.toml` and converts it to a python dictionary using `toml.load`. The resulting dictionary is given to pydantic, which validates the structure of the dictionary against the objects described below.

Required sections:

- *RobotpyBuildConfig* - base project configuration
- *DistutilsMetadata* - standard python distutils metadata

Optional sections:

- *WrapperConfig* - per-package configuration
- *StaticLibConfig*
- *MavenLibDownload* - download per-package maven artifacts
- *PatchInfo* - patch downloaded sources

Note: For a complete example pyproject.toml file, see `tests/cpp/pyproject.toml.tmpl`

10.3 Overrides

You can define 'override' sections that will be grafted onto the configuration if they match a particular platform. For example, to change the dependencies for a wrapper section on Windows:

```
[tool.robotpy-build.wrappers."PACKAGENAME".override.os_windows]
depends = ["windows-thing"]
```

Any element in the robotpy-build section of pyproject.toml can be overridden by specifying the identical section as '.override.KEYNAME'. If the key matches the current configuration, the override will be written to the original section. The matched keys are generated at runtime. Current supported platform override keys are:

- `arch_{platform_arch}`
- `os_{platform_os}`
- `platform_{platform_os}_{platform_arch}`

To get information about the current platform, you can run:

```
robotpy-build platform-info
```

To show the available platforms:

```
robotpy-build platform-info --list
```

To process a pyproject.toml and see the result of applying various overrides, you can use this tool to process for the current platform:

```
robotpy-build show-override
```

To show what would be processed for a different platform:

```
robotpy-build show-override -p linux-athena
```

Current supported platform/os/arch combinations are:

- OS: windows/osx/linux
- Arch: x86/x86-64/armv7l/aarch64

For ARM linux distributions we support:

- armv7l + nilrt (RoboRIO)
- armv7l + raspbian (Raspbian 10)
- aarch64 + bionic (Ubuntu 18.04)

10.4 Reference

class robotpy_build.pyproject_configs.**DistutilsMetadata**
Configures the metadata that robotpy-build passes to setuptools when the project is installed. The keys in this section match the standard arguments passed to the `setuptools.setup` function.

```
[tool.robotpy-build.metadata]
name = "my-awesome-dist"
description = "Cool thing"
license = "MIT"
```

robotpy-build will automatically detect/set the following keys:

- cmdclass
- ext_modules
- include_package_data - `True`
- long_description - Contents of README.md/README.rst
- long_description_content_type - If required
- packages
- python_requires - `>=3.6`
- version - via setuptools_scm
- zip_safe - `False`

Note: This section is required

---

author: str
The name of the package author

Type str

author_email: str
The email address of the package author

Type str

description: Optional[str] = None
A single line describing the package

Type Optional[str]

install_requires: List[str]
A string or list of strings specifying what other distributions need to be installed when this one is.

Type List[str]

license: str
The license for the package

Type str

name: str
The name of the package

Type str

url: str
A URL for the package (homepage)

Type str

class robotpy_build.pyproject_configs.**Download**
Download sources/libs/includes from a single file

```
[[tool.robotpy-build.wrappers."PACKAGENAME".download]]
url = "https://my/url/something.zip"
incdir = "include"
libs = ["mylib"]
```

dlopenlibs: Optional[List[str]] = None
If specified, names of contained shared link only libraries (in loading order). If None, set to name. If empty list, link only libs will not be downloaded.

Type Optional[List[str]]

extra_includes: List[str] = []
Extra include paths, relative to the include directory

{{ARCH}} and {{OS}} are replaced with the architecture/os name

Type List[str]

incdir: Optional[str] = None
Directory that contains include files.

{{ARCH}} and {{OS}} are replaced with the architecture/os name

Type Optional[str]

libdir: str = ''
Directory that contains library files

{{ARCH}} and {{OS}} are replaced with the architecture/os name

Type str

libexts: Dict[str, str] = {}
Library extensions map

Type Dict[str, str]

libs: Optional[List[str]] = None
If specified, names of contained shared libraries (in loading order)

Type Optional[List[str]]

linkexts: Dict[str, str] = {}
Compile time extensions map

Type Dict[str, str]

patches: Optional[List[*robotpy_build.pyproject_configs.PatchInfo*]] = None
If *sources* is set, apply the following patches to the sources. Patches must be in unified diff format.

Type Optional[List[*PatchInfo*]]

sources: Optional[List[str]] = None
List of sources to compile

Type Optional[List[str]]

url: str
URL of zipfile to download

{{ARCH}} and {{OS}} are replaced with the architecture/os name

Type str

class robotpy_build.pyproject_configs.**MavenLibDownload**
Used to download artifacts from a maven repository. This can download headers, shared libraries, and sources.

```
[tool.robotpy-build.wrappers."PACKAGENAME".maven_lib_download]
artifact_id = "mything"
group_id = "com.example.thing"
repo_url = "http://example.com/maven"
version = "1.2.3"
```

Note: For FIRST Robotics libraries, the information required can be found in the vendor JSON file

artifact_id: str
Maven artifact ID

Type str

dlopenlibs: Optional[List[str]] = None
Names of contained shared link only libraries (in loading order). If None, set to name. If empty list, link only libs will not be downloaded.

Type Optional[List[str]]

group_id: str
Maven group ID

Type str

libexts: Dict[str, str] = {}
Library extensions map

Type Dict[str, str]

libs: Optional[List[str]] = None
Names of contained shared libraries (in loading order). If None, set to artifact_id.

Type Optional[List[str]]

linkexts: Dict[str, str] = {}
Compile time extensions map

Type Dict[str, str]

patches: Optional[List[*robotpy_build.pyproject_configs.PatchInfo*]] = None
If *use_sources* is set, apply the following patches to the sources. Patches must be in unified diff format.

Type Optional[List[*PatchInfo*]]

repo_url: str
Maven repository URL

Type str

sources: Optional[List[str]] = None
If *use_sources* is set, this is the list of sources to compile

Type Optional[List[str]]

sources_classifier: str = 'sources'
Configure the sources classifier

Type str

use_sources: bool = False
When set, download sources instead of downloading libraries. When using this, you need to manually add the sources to the configuration to be compiled via *sources*.

Type bool

version: str
Version of artifact to download

Type str

class robotpy_build.pyproject_configs.**PatchInfo**
A unified diff to apply to downloaded source code before building a a wrapper.

```
[[tool.robotpy-build.wrappers."MY.PACKAGE.NAME".maven_lib_download.patches]]
patch = "path/to/my.patch"
strip = 0
```

patch: str
Name of patch file to apply

Type str

strip: int = 0
Number of directories to strip

Type int

class robotpy_build.pyproject_configs.**RobotpyBuildConfig**
Contains information for configuring the project

```
[tool.robotpy-build]
base_package = "my.package"
```

Note: This section is required

base_package: str
Python package to store version information and robotpy-build metadata in

Type str

supported_platforms: List[*robotpy_build.pyproject_configs.SupportedPlatform*] = []

See also:

SupportedPlatform

Type List[*SupportedPlatform*]

class robotpy_build.pyproject_configs.**StaticLibConfig**
Static libraries that can be consumed as a dependency by other wrappers in the same project. Static libraries are not directly installed, and as a result cannot be consumed by other projects.

```
[tool.robotpy-build.static_libs."MY.PACKAGE.NAME"]
```

download: Optional[List[*robotpy_build.pyproject_configs.Download*]] = None
If this project depends on external libraries downloadable from some URL specify it here

Type Optional[List[*Download*]]

ignore: bool = False
If True, skip this library; typically used in conjection with an override

Type bool

maven_lib_download: Optional[*robotpy_build.pyproject_configs.MavenLibDownload*] = None
If this project depends on external libraries stored in a maven repo specify it here

Type Optional[*MavenLibDownload*]

class robotpy_build.pyproject_configs.**SupportedPlatform**
Supported platforms for this project. Currently this information is merely advisory, and is used to generate error messages when platform specific downloads fail.

```
[tool.robotpy-build]
base_package = "my.package"
supported_platforms = [
    { os = "windows", arch = "x86-64" },
]
```

See also:

List of supported *platforms*

arch: Optional[str] = None
Platform architecture

Type Optional[str]

os: Optional[str] = None
Platform operating system name

Type Optional[str]

class robotpy_build.pyproject_configs.**WrapperConfig**
Configuration for building a C++ python extension module, optionally using autogenerated wrappers around existing library code.

```
[tool.robotpy-build.wrappers."PACKAGENAME"]
name = "package_name"
```

The PACKAGENAME above is a python package (eg "example.package.name"). A robotpy-build project can contain many different wrappers and packages.

depends: List[str] = []
List of robotpy-build library dependencies. This affects this wrapper library in the following ways:

- Any include file directories exported by the dependency will be added to the include path for any source files compiled by this wrapper
- It will be linked to any libraries the dependency contains
- The python module for the dependency will be imported in the _init{extension}.py file.

Type List[str]

download: Optional[List[*robotpy_build.pyproject_configs.Download*]] = None
If this project depends on external libraries downloadable from some URL specify it here

Type Optional[List[*Download*]]

extension: Optional[str] = None
Name of extension to build. If None, set to _{name}

Type Optional[str]

extra_includes: List[str] = []
List of extra include directories to export, relative to the project root.

Type List[str]

generate: Optional[List[Dict[str, str]]] = None

Specifies header files that autogenerated pybind11 wrappers will be created for. Simple C++ headers will most likely 'just work', but complex headers will need to have an accompanying *generation_data* file specified that can customize the autogenerated files.

List of dictionaries: each dictionary key is used for the function name of the initialization function, the value is the header that is being wrapped. The header is first looked for relative to the package, then relative to each include directory (including downloaded and extracted packages).

```
[tool.robotpy-build.wrappers."PACKAGENAME"]
generate = [
  { Name = "header.h" },
]
```

See also:

Automated C++ header wrapping

Type Optional[List[Dict[str, str]]]

generation_data: Optional[str] = None

Path to a single data.yml to use during code generation, or a directory of yaml files. If a directory, generation data will be looked up using the key in the generate dictionary.

These YAML files can be generated via the robotpy-build command line tool:

```
robotpy-build create-gen --write
```

See also:

Generator Customization

Type Optional[str]

ignore: bool = False

If True, skip this wrapper; typically used in conjection with an override.

Type bool

libinit: Optional[str] = None

Name of generated file that ensures the shared libraries and any dependencies are loaded. Defaults to `_init{extension}.py`

Generally, you should create an `__init__.py` file that imports this module, otherwise your users will need to do so.

Type Optional[str]

maven_lib_download: Optional[*robotpy_build.pyproject_configs.MavenLibDownload*] = None

If this project depends on external libraries stored in a maven repo specify it here.

Type Optional[*MavenLibDownload*]

name: str

Name that other projects/wrappers use in their 'depends' list

Type str

pp_defines: List[str] = []

Preprocessor definitions to apply when compiling this wrapper.

Type List[str]

sources: List[str] = []
Optional source files to compile. Path is relative to the root of the project.

Type List[str]

type_casters: Dict[str, List[str]] = {}
Specifies type casters that this package exports. robotpy-build will attempt to detect these types at generation time and include them in generated wrappers.

```
[tool.robotpy-build.wrappers."package-name".type_casters]
"namespace_type1_type_caster.h" = ["namespace::type1", .. ]
```

See also:

Type Casters

Type Dict[str, List[str]]

CHAPTER

ELEVEN

GENERATOR CUSTOMIZATION

Because robotpy-build's code generation is intended to be a semi-automated process (except for simple C++ code), a rich set of per-{class/function/parameter} configuration options can be specified in per-file YAML configurations.

Additionally, some headers are too complex for the autogenerator to completely process, so when this occurs you must manually specify required information in the YAML file.

Most files generated by robotpy-build are customizable.

Note: robotpy-build is designed for the RobotPy project and may contain defaults that aren't appropriate for all projects. If you find that you need more customization, file an issue on github and let's talk about it!

11.1 Location of customization file

In your `pyproject.toml`, you can specify either a single YAML file with customizations, or you can specify a directory that robotpy-build will search for YAML files.

Single file:

```
[tool.robotpy-build.wrappers."PACKAGENAME"]
generation_data = "gen/data.yml"
```

Multiple files:

```
[tool.robotpy-build.wrappers."PACKAGENAME"]
generation_data = "gen"
```

When a directory is specified, pybind11 will search for YAML files in the directory based on the header filename. In the above example, customization data for `header.h` could be specified in `gen/header.yml`.

11.2 Autogeneration

The default values for these YAML files can be generated via the robotpy-build command line tool:

```
robotpy-build create-gen --write
```

This can be a good way to get the boilerplate out of the way when you need to provide customizations.

11.3 Reference

The following strctures describe the dictionaries that are read from the YAML file. The toplevel structure is *HooksDataYaml*.

class robotpy_build.hooks_datacfg.**BufferData**

len: str
Name of the C++ length parameter. An out-only parameter, it will be set to the size of the python buffer, and will be returned so the caller can determine how many bytes were written

Type str

minsz: Optional[int] = None
If specified, the minimum size of the python buffer

Type Optional[int]

src: str
Name of C++ parameter that the buffer will use

Type str

type: *robotpy_build.hooks_datacfg.BufferType*
Indicates what type of python buffer is required

Type *BufferType*

class robotpy_build.hooks_datacfg.**BufferType**(*value*)
An enumeration.

IN = 'in'
The buffer must indicate that it is readable (such as bytes, or bytearray)

INOUT = 'inout'
The buffer must indicate that it readable or writeable (such as a bytearray)

OUT = 'out'
The buffer must indicate that it is writeable (such as a bytearray)

class robotpy_build.hooks_datacfg.**ClassData**

attributes: Dict[str, *robotpy_build.hooks_datacfg.PropData*] = {}

base_qualnames: Dict[str, str] = {}
Specify fully qualified names for the bases

Type Dict[str, str]

constants: List[str] = []
Extra constexpr to insert into the trampoline and wrapping scopes

Type List[str]

doc: Optional[str] = None
Docstring for the class

Type Optional[str]

doc_append: Optional[str] = None
Text to append to the (autoconverted) docstring

Type Optional[str]

enums: Dict[str, *robotpy_build.hooks_datacfg.EnumData*] = {}

force_depends: List[str] = []
If there are circular dependencies, this will help you resolve them manually. TODO: make it so we don't need this

Type List[str]

force_no_default_constructor: bool = False

force_no_trampoline: bool = False

force_type_casters: List[str] = []
Use this to bring in type casters for a particular type that may have been hidden (for example, with a typedef or definition in another file), instead of explicitly including the header. This should be the full namespace of the type.

Type List[str]

ignore: bool = False

ignored_bases: List[str] = []

is_polymorphic: bool = False

methods: Dict[str, *robotpy_build.hooks_datacfg.FunctionData*] = {}

nodelete: bool = False
If the object shouldn't be deleted by pybind11, use this. Disables implicit constructors.

Type bool

rename: Optional[str] = None
Set the python name of the class to this

Type Optional[str]

shared_ptr: bool = True
If the type was created as a shared_ptr (such as via std::make_shared) then pybind11 needs to be informed of this.

https://github.com/pybind/pybind11/issues/1215

One way you can tell we messed this up is if there's a double-free error and the stack trace involves a unique_ptr destructor

Type bool

subpackage: Optional[str] = None
If specified, put the class in a sub.pack.age. Ignored for functions attached to a class. When template parameters are used, must define subpackage on template instances instead

Type Optional[str]

template_inline_code: str = ''
If this is a template class, the specified C++ code is inserted into the template definition

Type str

template_params: Optional[List[str]] = None
If this is a template class, a list of the parameters if it can't be autodetected (currently can't autodetect). If there is no space in the parameter, then it is assumed to be a 'typename', otherwise the parameter is split by space and the first item is the type and the second parameter is the name (useful for integral templates)

Type Optional[List[str]]

trampoline_inline_code: Optional[str] = None
If this class has an associated trampoline, add this code inline at the bottom of the trampoline class. This is rarely useful.

Type Optional[str]

typealias: List[str] = []
Extra 'using' directives to insert into the trampoline and the wrapping scope

Type List[str]

classmethod validate_attributes(*value*)

classmethod validate_enums(*value*)

classmethod validate_methods(*value*)

class robotpy_build.hooks_datacfg.**EnumData**

doc: Optional[str] = None
Set your own docstring for the enum

Type Optional[str]

doc_append: Optional[str] = None
Text to append to the (autoconverted) docstring

Type Optional[str]

ignore: bool = False
If set to True, this property is not made available to python

Type bool

rename: Optional[str] = None
Set the python name of this enum to the specified string

Type Optional[str]

subpackage: Optional[str] = None
If specified, put the enum in a sub.pack.age (ignored for enums that are part of classes)

Type Optional[str]

value_prefix: Optional[str] = None

values: Dict[str, *robotpy_build.hooks_datacfg.EnumValue*] = {}

class robotpy_build.hooks_datacfg.**EnumValue**

doc: Optional[str] = None
Docstring for the enum value

Type Optional[str]

doc_append: Optional[str] = None
Text to append to the (autoconverted) docstring

Type Optional[str]

ignore: bool = False
If set to True, this property is not made available to python

Type bool

rename: Optional[str] = None
Set the python name of this enum value to the specified string

Type Optional[str]

class robotpy_build.hooks_datacfg.**FunctionData**
Customize the way the autogenerator binds a function.

```
functions:
  # for non-overloaded functions, just specify the name + customizations
  name_of_non_overloaded_fn:
    # add customizations for function here

  # For overloaded functions, specify the name, but each overload
  # separately
  my_overloaded_fn:
    overloads:
      int, int:
        # customizations for `my_overloaded_fn(int, int)`
      int, int, int:
        # customizations for `my_overloaded_fn(int, int, int)`
```

buffers: List[*robotpy_build.hooks_datacfg.BufferData*] = []

cpp_code: Optional[str] = None
Use this code instead of the generated code

Type Optional[str]

doc: Optional[str] = None
Docstring for the function, will attempt to convert Doxygen docs if omitted

Type Optional[str]

doc_append: Optional[str] = None
Text to append to the (autoconverted) docstring for the function

Type Optional[str]

ifdef: Optional[str] = None
Generate this in an *#ifdef*

Type Optional[str]

ifndef: Optional[str] = None
Generate this in an *#ifndef*

Type Optional[str]

ignore: bool = False
If True, don't wrap this

Type bool

ignore_pure: bool = False
If True, don't wrap this, but provide a pure virtual implementation

Type bool

internal: bool = False
If True, prepends an underscore to the python name

Type bool

keepalive: Optional[List[Tuple[int, int]]] = None
Adds py::keep_alive<x,y> to the function. Overrides automatic keepalive support, which retains references passed to constructors. https://pybind11.readthedocs.io/en/stable/advanced/functions.html#keep-alive

Type Optional[List[Tuple[int, int]]]

no_release_gil: Optional[bool] = None
By default, robotpy-build will release the GIL whenever a wrapped function is called.

Type Optional[bool]

overloads: Dict[str, *robotpy_build.hooks_datacfg.FunctionData*] = {}

param_override: Dict[str, *robotpy_build.hooks_datacfg.ParamData*] = {}
Mechanism to override individual parameters

Type Dict[str, *ParamData*]

rename: Optional[str] = None
Use this to set the name of the function as exposed to python

Type Optional[str]

return_value_policy: *robotpy_build.hooks_datacfg.ReturnValuePolicy* = 'automatic'
https://pybind11.readthedocs.io/en/stable/advanced/functions.html#return-value-policies

Type *ReturnValuePolicy*

subpackage: Optional[str] = None
If specified, put the function in a sub.pack.age

Type Optional[str]

template_impls: Optional[List[List[str]]] = None
If this is a function template, this is a list of instantiations that you wish to provide. This is a list of lists, where the inner list is the template parameters for that function

Type Optional[List[List[str]]]

classmethod validate_overloads(*value*)

virtual_xform: Optional[str] = None
Specify a transformation lambda to be used when this virtual function is called from C++. This inline code should be a lambda that has the same arguments as the original C++ virtual function, except the first argument will be a py::function with the python overload

cpp_code should also be specified for this to be useful

For example, to transform a function that takes an iostream into a function that returns a string:

```
cpp_code: |
  [](MyClass* self) {
    return "string";
  }
virtual_xform: |
  [](py::function fn, MyClass* self, std::iostream &is) {
     std::string d = py::cast(fn());
     is << d;
  }
```

Type Optional[str]

class robotpy_build.hooks_datacfg.**HooksDataYaml**

Format of the file in [tool.robotpy-build.wrappers."PACKAGENAME"] generation_data

attributes: Dict[str, *robotpy_build.hooks_datacfg.PropData*] = {}

Key is the attribute (variable) name

```
attributes:
  my_variable:
    # customizations here, see PropData
```

Type Dict[str, *PropData*]

classes: Dict[str, *robotpy_build.hooks_datacfg.ClassData*] = {}

Key is the class name

```
classes:
  CLASSNAME:
    # customizations here, see ClassData
```

Type Dict[str, *ClassData*]

enums: Dict[str, *robotpy_build.hooks_datacfg.EnumData*] = {}

Key is the enum name, for enums at global scope

```
enums:
  MyEnum:
    # customizations here, see EnumData
```

Type Dict[str, *EnumData*]

extra_includes: List[str] = []

Adds `#include <FILENAME>` directives to the top of the autogenerated C++ file, after autodetected include dependencies are inserted.

Type List[str]

extra_includes_first: List[str] = []

Adds `#include <FILENAME>` directives after robotpy_build.h is included, but before any autodetected include dependencies. Only use this when dealing with broken headers.

Type List[str]

functions: Dict[str, *robotpy_build.hooks_datacfg.FunctionData*] = {}

Key is the function name

```
functions:
  fn_name:
    # customizations here, see FunctionData
```

Type Dict[str, *FunctionData*]

inline_code: Optional[str] = None

Specify raw C++ code that will be inserted at the end of the autogenerated file, inside a function. This is useful for extending your classes or providing other customizations. The following C++ variables are available:

- `m` is the `py::module` instance

- `cls_CLASSNAME` are `py::class` instances
- ... lots of other things too

The trampoline class (useful for accessing protected items) is available at `{CLASSNAME}_Trampoline`

To see the full list, run a build and look at the generated code at `build/*/gensrc/**/*.cpp`

Recommend that you use the YAML multiline syntax to specify it:

```
inline_code: |
  cls_CLASSNAME.def("get42", []() { return 42; });
```

Type `Optional[str]`

strip_prefixes: List[str] = []

templates: Dict[str, *robotpy_build.hooks_datacfg.TemplateData*] = {}

Instantiates a template. Key is the name to give to the Python type.

```
templates:
  ClassName:
    # customizations here, see TemplateData
```

Type `Dict[str, *TemplateData*]`

classmethod validate_attributes(*value*)

classmethod validate_classes(*value*)

classmethod validate_enums(*value*)

classmethod validate_functions(*value*)

class robotpy_build.hooks_datacfg.**ParamData**

Various ways to modify parameters

array_size: Optional[int] = None

Force an array size

Type `Optional[int]`

default: Optional[str] = None

Default value for parameter

Type `Optional[str]`

force_out: bool = False

Force this to be an 'out' parameter

See also:

Out parameters

Type `bool`

ignore: bool = False

Ignore this parameter

Type `bool`

name: Optional[str] = None

Set parameter name to this

Type Optional[str]

x_type: Optional[str] = None
Change C++ type emitted

Type Optional[str]

class robotpy_build.hooks_datacfg.**PropAccess**(*value*)
An enumeration.

AUTOMATIC = 'auto'
Determine read/read-write automatically:

- If a struct/union, default to readwrite
- If a class, default to readwrite if a basic type that isn't a reference, otherwise default to readonly

READONLY = 'readonly'
Allow python users access to the value, but ensure it can't change. This is useful for properties that are defined directly in the class

READWRITE = 'readwrite'
Allows python users to read/write the value

class robotpy_build.hooks_datacfg.**PropData**

access: *robotpy_build.hooks_datacfg.PropAccess* = 'auto'
Python code access to this property

Type *PropAccess*

doc: Optional[str] = None
Docstring for the property (only available on class properties)

Type Optional[str]

doc_append: Optional[str] = None
Text to append to the (autoconverted) docstring

Type Optional[str]

ignore: bool = False
If set to True, this property is not made available to python

Type bool

rename: Optional[str]
Set the python name of this property to the specified string

Type Optional[str]

class robotpy_build.hooks_datacfg.**ReturnValuePolicy**(*value*)
See pybind11 documentation for what each of these values mean.

AUTOMATIC = 'automatic'

AUTOMATIC_REFERENCE = 'automatic_reference'

COPY = 'copy'

MOVE = 'move'

REFERENCE = 'reference'

REFERENCE_INTERNAL = 'reference_internal'

TAKE_OWNERSHIP = 'take_ownership'

class robotpy_build.hooks_datacfg.**TemplateData**

Instantiates a template as a python type. To customize the class, add it to the `classes` key and specify the template type.

Code to be wrapped:

```
template <typename T>
class MyClass {};
```

To bind `MyClass<int>` as the python class `MyIntClass`, add this to your YAML:

```
classes:
  MyClass:
    template_params:
    - T

templates:
  MyIntClass:
    qualname: MyClass
    params:
    - int
```

doc: Optional[str] = None

Set the docstring for the template instance

Type Optional[str]

doc_append: Optional[str] = None

Text to append to the (autoconverted) docstring for the template instance

Type Optional[str]

params: List[str]

Template parameters to use

Type List[str]

qualname: str

Fully qualified name of instantiated class

Type str

subpackage: Optional[str] = None

If specified, put the template instantiation in a sub.pack.age

Type Optional[str]

CHAPTER

TWELVE

TROUBLESHOOTING

12.1 Compilation errors

12.1.1 error: invalid use of incomplete type

pybind11 requires you to use complete types when binding an object. Typically this means that somewhere in the header there was a forward declaration:

```
class Foo;
```

Foo is defined in some header, but not the header you're scanning. To fix it, tell the generator to include the header:

```
extra_includes:
- path/to/Foo.h
```

12.1.2 error: 'SomeEnum' has not been declared

Often this is referring to a parameter type or a default argument. You can use the 'param_override' setting for that function to fix it. For example, if you had the following C++ code:

```
void fnWithEnum(SomeEnum e);
```

And `SomeEnum` was actually `somens::SomeEnum` but robotpy-build wasn't able to automatically determine that. You could fix it like so:

```
functions:
  fnWithEnum:
    param_override:
      # e is the name of the param
      e:
        # x_type tells the generator to force the type to the specified string
        x_type: "somens::SomeEnum"
```

12.1.3 error: static assertion failed: Cannot use an alias class with a non-polymorphic type

pybind11 has a static assert that occurs when an trampoline class is specified for a class that is a subclass of some other class, but none of its bases have any virtual functions. This is to prevent

robotpy-build generates trampoline classes to allow python code to override virtual functions in C++ classes. robotpy-build isn't always able to detect when a trampoline isn't appropriate.

When this error occurs, you can force robotpy-build to turn off trampoline classes for a specific type:

```
classes:
  X:
    force_no_trampoline: true
```

12.1.4 error: 'rpygen/__SomeClassName.hpp' file not found

This occurs when robotpy-build has created a trampoline class for a child class, and it is trying to include the header for the parent trampoline class. There are several reasons why this might happen.

Sometimes this error occurs because the parent lives in a different package that you didn't declare a dependency on in `pyproject.toml`. Add the dependency and the parent trampoline class should be found.

If the base class is polymorphic in a way that robotpy-build wasn't able to detect, you can force it to be polymorphic:

```
classes:
  X:
    is_polymorphic: true
```

Unfortunately, sometimes the base isn't a polymorphic type and you can't change it. In this case you can turn off the trampoline class for the child class:

```
classes:
  X:
    force_no_trampoline: true
```

12.2 Runtime errors

12.2.1 ImportError: dynamic module does not define module export function (PyInit__XXX)

Sometimes this exhibits itself as `unresolved external symbol PyInit__XXX`.

This error indicates that you compiled a Python C++ module without actually defining a module. Most likely, you forgot to to add a file which contains these contents:

```
#include <rpygen_wrapper.hpp>

RPYBUILD_PYBIND11_MODULE(m) {
    initWrapper(m);
}
```

You can of course put other content in here if needed.

CHAPTER

THIRTEEN

SUPPORT

If you run into a problem with robotpy-build that you think is a bug, or perhaps there is something wrong with the documentation or just too difficult to do, please feel free to file bug reports on the github issue tracker.

The developers can also be reached at the RobotPy Gitter channel.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X