

# litgen

## Contents

### Installation

- Installation or online usage
- Use litgen online
- Quickstart with litgen

### Bindings - Advanced Topics

- First steps
- litgen options
- Headers processing
- Headers amalgamation
- Generated code layout
- Names and types translation
- Functions
- Classes and structs
- Template classes and functions
- Enums
- Namespaces
- Manually add custom bindings
- Preprocessor and macros
- Ignore litgen warnings
- Postprocessing and preprocessing
- srcML - C++ parsing advices

### Litgen as C++ transformation tool

[Skip to main content](#)

- Simple C++ code transformations
- Optional: install srcML command line tool

### What is litgen

[litgen](#), also known as *Literate Generator*, is an automatic python bindings generator for humans who like nice code and APIs.

It can be used to bind C++ libraries into *documented* and *discoverable* python modules using [pybind11](#) or [nanobind](#).

It can also be used as C++ transformation and refactoring tool.

[Source code](#), [Documentation](#), [PyPI](#)

Although being relatively new (2022), litgen was battle tested on 20 different libraries totalling more than 100,000 lines of code, and it is the main driving force behind the python bindings for [Dear ImGui Bundle](#).

litgen puts a strong emphasis on emitting documented and discoverable code, thus providing a great experience for the final python user.

### srcML

litgen is based on [srcML](#), a multi-language parsing tool to convert source code into XML, with a developer centric approach: preprocessor statements are kept unprocessed, and all original text is preserved (including white space, comments and special characters).

### Full pdf version of this book

View or download the [full pdf](#) for this manual.

You may feed it into a LLM such as ChatGPT, so that it can help you with litgen related questions.

# Documented bindings

As an intro, here is an example when using bindings generated by litgen inside a python Integrated Development Environment (IDE):

```
imgui.push
f push_item_width(item_width)                                imgui_bundle.imgur
f push_id(str_id)                                            imgui_bundle.imgur
f push_font(font)                                           imgui_bundle.imgur
f push_tab_stop(tab_stop)                                     imgui_bundle.imgur
f push_clip_rect(clip_rect_min, clip_rect_max, intersect_w... imgui_bundle.imgur
f push_button_repeat(repeat)                                 imgui_bundle.imgur
f push_style_color(idx, col)                                imgui_bundle.imgur
f push_style_var(idx, val)                                 imgui_bundle.imgur
```

*Example of auto-completion in an IDE: all bindings are discoverable*

```
imgui.push_item_width(200.)

```

*Parameters type are accurately reproduced, and the function documentation is accessible*

In the example above, the bindings were generated from the following C++ function signature:

```
// Parameters stacks (current window)
IMGUI_API void PushItemWidth(float item_width); // push width of items for
```

And the generated code consists of two parts:

1. a python stub file, which contains the documentation and the function signatures, e.g.:

```
# Parameters stacks (current window)
# IMGUI_API void PushItemWidth(float item_width); /* original C++ si
def push_item_width(item_width: float) -> None.
```

[Skip to main content](#)

```
"""push width of items for common large "item+label" widgets. >0.0: width in pixels
pass
```

1. a C++ bindings file, which contains the actual bindings, e.g.:

```
m.def("push_item_width",
    ImGui::PushItemWidth,
    py::arg("item_width"),
    "push width of items for common large \"item+label\" widgets. >0.0: width in pixels
pass")
```

## Examples

More complete examples can be found online inside the [Dear ImGui Bundle](#) repository, for example:

- [imgui.h](#) header file that declares the API for [Dear ImGui](#) in a documented way
- [imgui.py](#) the corresponding python stub file which exposes the bindings in a documented way

## Compatibility

Being based on [srcML](#), litgen is compatible with C++14: your code can of course make use of more recent features (C++17, C++20 and C++23), but the API that you want to expose to python must be C++14 compatible.

## License

The [litgen project](#) is published under the [GNU General Public License, version 3](#).

Code generated by litgen is not subject to the GPL license, allowing you to use it freely in your projects without the obligations of GPLv3.

## Credits

Acknowledging the use of litgen in your project's documentation is a nice way to support the

[Skip to main content](#)

litgen is based on [srcML](#), a multi-language parsing tool that converts source code into XML with a developer-centric approach: preprocessor statements are kept unprocessed, and all original text is preserved (including whitespace, comments, and special characters). [srcML](#) is published under the [GNU General Public License, version 3.](#)

---

## Keep in Touch

We'd love to hear about how you're using litgen! If you are using it, please consider [sharing your experience and insights.](#)

## Contributing

Contributions and skilled contributors are welcome! See contributors oriented documentation inside [Build.md](#).

## Help the Project

If litgen has made a difference for you - especially in a commercial or research setting - please consider [making a donation](#). Your support goes a long way toward keeping the project alive and growing. Any contribution, no matter the size, is greatly appreciated!

## Technical Support

C++ is notorious for being hard to parse. As a consequence, the author makes no guarantee that the generator will work on all kinds of C++ constructs.

**Open Source Support:** If you need assistance in an open-source context, please [open an issue](#) in the repository.

**Professional Support:** For more in-depth help in a professional setting, feel free to reach out to the author via email for consulting inquiries.

---

# Installation or online usage

You can either use litgen online, or install it locally.

## Use litgen online

You do not to install anything. Simply:

1. Set some conversion options
2. Paste the API of the code for which you want to generate bindings
3. Generate the binding code (C++ and python stubs), which you can then copy and paste into your project.

[Access the online tool](#)

## Install litgen locally

If installing locally, you can integrate liten into your building process, and it can generate full C++ bindings files, as well as python stubs (API documentation and declaration).

Install litgen with pip:

```
pip install "litgen@git+https://github.com/pthom/litgen"
```

Then, follow the instructions in [Quickstart with litgen](#) section.

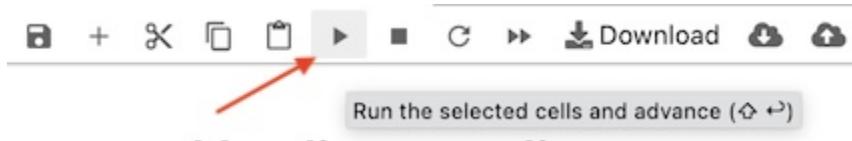
## Use litgen online

This page shows an example of conversion of C++ code into C++ bindings code and python stubs. You can open it in an online interactive environment ([mybinder](#)), where you can edit the code below.

1. To launch the online tool, click on the rocket icon at the top right of this page, or use [this link](#)

[Skip to main content](#)

3. Once it is launched, click on "Run Cell" to run each of the cells:



You can edit the C++ code as well as the options below. The generated code will be displayed under the cell, once you clicked "Run Cell". You will then be able to see:

- The generated python stubs (aka declarations)
- The generated C++ bindings code for pybind11
- The generated C++ bindings code for nanobind
- The generated glue code (if needed)

```
import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()

# This namespace will not be outputted as a python module
options.namespaces_root = ["MyNamespace"]
# All functions, modules and namespaces names are converted to snake case
options.python_convert_to_snake_case = True
# This is an API marker in the C++ code (for shared libraries code)
options.srcmlcpp_options.functions_api_prefixes = "^MY_API$"
# Also create bindings for functions that do not have the API marker
options.fn_exclude_non_api = False
# Optional comment that can be added to non API functions
options.fn_non_api_comment = ""
# "Box" immutable functions parameter when they should be modifiable
options.fn_params_replace_modifiable_immutable_by_boxed_regex = r".*"
# Which numeric and string preprocessor do we want to export
options.macro_define_include_by_name_regex = "^MY_"

code = """
// This namespace is not outputted as a submodule, since it is marked as Root (see options)
namespace MyNamespace
{
    // Multiplies a list of double by a given factor, returns updated list
    std::vector<double> MultiplyDoubles(const std::vector<double>& v, float k);

    // changes the value of the bool parameter (passed by modifiable reference)
    // (This function will use a BoxedBool in the python code, so that its value can be modified)
    void SwitchBoolValue(bool &v);

    // Standalone comment blocs are also exported

    // This function includes an API marker which will be ignored when generating the bindings
    MY_API int MySubstract(int a, int b); // eol doc is also included in bindings definition

    namespace MyMath // This namespace is not ignored and introduces a new python module
    {
        // div and mul: divide or multiply float numbers
        // (This comment concerns the two grouped
        // functions below, and will be exported as such)
        float Div(float a, float b); // Divide
        float Mul(float a, float b); // Multiply
    }

    // Stores Pixel coordinates
    struct Pixel
    {
        // Coordinates
        double x = 2., y = 3.;

        // Draw a pixel
    }
}
```

[Skip to main content](#)

```
    private:  
    double _Norm(); // this will not be exported as it is private  
};  
  
// This macro value be exported, as it matches the regex macro_define_include_b  
#define MY_VALUE 123  
}  
....  
  
litgen_demo.demo(  
    options, code, show_cpp=False, show_pydef=True, height=80  
)
```

Corresponding python decls (stub)

copy 

```
##### <generated_from:BoxedTypes> #####
class BoxedBool:
    value: bool
    def __init__(self, v: bool = False) -> None:
        pass
    def __repr__(self) -> str:
        pass
##### </generated_from:BoxedTypes> #####
def multiply_doubles(v: List[float], k: float) -> List[float]:
    """ Multiplies a list of double by a given factor, returns updated list"""
    pass

def switch_bool_value(v: BoxedBool) -> None:
    """ changes the value of the bool parameter (passed by modifiable reference)
        (This function will use a BoxedBool in the python code, so that its value can
        be modified)"""
    pass

# Standalone comment blocs are also exported

# This function includes an API marker which will be ignored when generating the
def my_substract(a: int, b: int) -> MY_API int:
    """ eol doc is also included in bindings doc"""
    pass

class Pixel:
    """ Stores Pixel coordinates"""
    # Coordinates
    x: float = 2.
    # Coordinates
    y: float = 3.

    def draw(self, i: Image) -> None:
        """ Draw a pixel"""
        pass

    def __init__(self, x: float = 2., y: float = 3.) -> None:
        """Auto-generated default constructor with named params"""
        pass

# This macro value be exported, as it matches the regex macro_define_include_by_
MY_VALUE = 123
```

[Skip to main content](#)

```

pass # (This corresponds to a C++ namespace. All method are static!)
"""This namespace is not ignored and introduces a new python module"""
# div and mul: divide or multiply float numbers
# (This comment concerns the two grouped
# functions below, and will be exported as such)
@staticmethod
def div(a: float, b: float) -> float:
    """ Divide"""
    pass
@staticmethod
def mul(a: float, b: float) -> float:
    """ Multiply"""
    pass

# </submodule my_math>

```

*pybind11 C++ binding code*

copy ↗

```

////////// <generated_from:BoxedTypes> //////////
auto pyClassBoxedBool =
    py::class_<BoxedBool>
        (m, "BoxedBool", "")
    .def_readwrite("value", &BoxedBool::value, "")
    .def(py::init<bool>(),
        py::arg("v") = false)
    .def("__repr__",
        &BoxedBool::__repr__)
;
////////// <generated_from:BoxedTypes> //////////

m.def("multiply_doubles",
    MyNamespace::MultiplyDoubles,
    py::arg("v"), py::arg("k"),
    "Multiplies a list of double by a given factor, returns updated list");

m.def("switch_bool_value",
    [](BoxedBool & v)
{
    auto SwitchBoolValue_adapt_modifiable_immutable = [](BoxedBool & v)
    {
        bool & v_boxed_value = v.value;

        MyNamespace::SwitchBoolValue(v_boxed_value);
    };

    SwitchBoolValue_adapt_modifiable_immutable(v);
},
.......
```

[Skip to main content](#)

```

m.def("my_subtract",
      MyNamespace::MySubtract,
      py::arg("a"), py::arg("b"),
      "eol doc is also included in bindings doc");

auto pyClassPixel =
    py::class_<MyNamespace::Pixel>
        (m, "Pixel", "Stores Pixel coordinates")
    .def(py::init<>([](
double x = 2., double y = 3.))
{
    auto r_ctor_ = std::make_unique<MyNamespace::Pixel>();
    r_ctor_->x = x;
    r_ctor_->y = y;
    return r_ctor_;
})
, py::arg("x") = 2., py::arg("y") = 3.
)
.def_readwrite("x", &MyNamespace::Pixel::x, "Coordinates")
.def_readwrite("y", &MyNamespace::Pixel::y, "Coordinates")
.def("draw",
     &MyNamespace::Pixel::Draw,
     py::arg("i"),
     "Draw a pixel")
;
m.attr("MY_VALUE") = 123;

{ // <namespace MyMath>
    py::module_ pyNsMyMath = m.def_submodule("my_math", "This namespace is not :
pyNsMyMath.def("div",
    MyNamespace::MyMath::Div,
    py::arg("a"), py::arg("b"),
    "Divide");

    pyNsMyMath.def("mul",
        MyNamespace::MyMath::Mul
}

```

*nanobind C++ binding code*

copy ↗

```

////////// <generated_from:BoxedTypes> //////////
auto pyClassBoxedBool =
    nb::class_<BoxedBool>
        (m, "BoxedBool", "")
    .def_rw("value", &BoxedBool::value, "")
    .def(nb::init<bool>(),
        nb::arg("v") = false)
    .def("__repr__",
        &BoxedBool::__repr__)
;

```

[Skip to main content](#)

```

m.def("multiply_doubles",
      MyNamespace::MultiplyDoubles,
      nb::arg("v"), nb::arg("k"),
      "Multiplies a list of double by a given factor, returns updated list");

m.def("switch_bool_value",
      [](BoxedBool & v)
      {
          auto SwitchBoolValue_adapt_modifiable_immutable = [](BoxedBool & v)
          {
              bool & v_boxed_value = v.value;

              MyNamespace::SwitchBoolValue(v_boxed_value);
          };

          SwitchBoolValue_adapt_modifiable_immutable(v);
      },
      nb::arg("v"),
      " changes the value of the bool parameter (passed by modifiable reference)\r\n"

m.def("my_substract",
      MyNamespace::MySubstract,
      nb::arg("a"), nb::arg("b"),
      "eol doc is also included in bindings doc");

auto pyClassPixel =
nb::class_<MyNamespace::Pixel>
    ("Pixel", "Stores Pixel coordinates")
.def("__init__", [](MyNamespace::Pixel * self, double x = 2., double y = 3.)
{
    new (self) MyNamespace::Pixel(); // placement new
    auto r_ctor_ = self;
    r_ctor_->x = x;
    r_ctor_->y = y;
},
nb::arg("x") = 2., nb::arg("y") = 3.
)
.def_rw("x", &MyNamespace::Pixel::x, "Coordinates")
.def_rw("y", &MyNamespace::Pixel::y, "Coordinates")
.def("draw",
      &MyNamespace::Pixel::Draw,
      nb::arg("i"),
      "Draw a pixel")
;
m.attr("MY_VALUE") = 123;

{ // <namespace MyMath>
nb::module_ pyNsMyMath = m.def_submodule("my_math", "This namespace is not :
pyNsMyMath.def("div",
      MyNamespace::MyMath::Div,
      nb::arg("a") nb::arg("b"))

```

[Skip to main content](#)

```
pyNsMyMath.def("mul",
    MvNamespace::MvMath::Mul.
```

C++ glue code

copy ↗

```
struct BoxedBool
{
    bool value;
    BoxedBool(bool v = false) : value(v) {}
    std::string __repr__() const { return std::string("BoxedBool(") + std::to_s(
};
```

## Quickstart with litgen

[litgen\\_template](#) is a template repository to build python bindings using [litgen](#), [pybind11](#) or [nanobind](#) and [scikit-build](#).

This template is based on [scikit\\_build\\_example](#).

## Usage for final users

Below are the instructions you would give to final users of your bindings. They are extremely short:

### First, install the package from source

```
git clone https://github.com/pthom/litgen_template.git && cd litgen_template
pip install -v .
```

### Then, use it from python

```
import daft_lib
daft_lib.add(1, 2)
```

(this template builds bindings for a C++ library called DaftLib, and publishes it as a python module called `daft_lib`)

[Skip to main content](#)

Of course, you could also publish your bindings to PyPI, and tell your users to install them with `pip install daft-lib`. This template provides tooling to make the publishing process easier, via [cibuildwheel](#)

## Autogenerate the binding code

### Install requirements

#### Create a virtual environment

```
python3 -m venv venv  
source venv/bin/activate
```

#### Install the requirements

```
pip install -r requirements-dev.txt
```

This will install [litgen](#) (the bindings generator), [pybind11](#) and [nanobind](#) (libraries to create C++ to Python bindings), [pytest](#) (for the tests), [black](#) (a code formatter), and [mypy](#) (static type checker for python).

See [requirements-dev.txt](#).

## Generate bindings

### Optionally, change the C++ code

- Change the C++ code (add functions, etc.) in [src/cpp\\_libraries/DaftLib](#)
- Adapt the generation options inside [tools/autogenerate\\_bindings.py](#)

### Optionally, switch to nanobind

[Skip to main content](#)

```
export LITGEN_USE_NANOBIND=ON
```

## Run the code generation via litgen

```
python tools/autogenerate_bindings.py
```

This will:

- Write the cpp binding code into [\\_pydef\\_pybind11/pybind\\_DaftLib.cpp](#) or [\\_pydef\\_nanobind/nanobind\\_DaftLib.cpp](#)
- Write the python stubs (i.e. typed declarations) inside [\\_stubs/daft\\_lib/\\_init\\_\\_.pyi](#).

*Tip: compare the [python stubs](#) with the [C++ header file](#) to see how close they are!*

## Customize bindings generation

[tools/autogenerate\\_bindings.py](#) is an example of how to drive the bindings generation via a script.

It will set a number of options, demonstrating a subset of litgen options: see the [litgen documentation](#) for more details.

The options in this script are heavily documented, and correspond to the documentation you can find in [DaftLib.h](#)

## Adapt for your own library

### Names, names, names

In this template repository:

- the C++ library is called [DaftLib](#)
- the native python module generated by pybind11 is called [\\_daft\\_lib](#)
- the python module which is imported by users is called [daft lib](#) (it imports and optionally

[Skip to main content](#)

- the pip package that can optionally be published to PyPI is called `daft-lib` (as Pypi does not allow dashes in package names)

You can change these names by running `change_lib_name.py` in the `tools/change_lib_name` folder.

## Structure of this template

### Bound C++ library

The C++ library `DaftLib` is stored inside `src/cpp_libraries/DaftLib/`

```
src/
└── cpp_libraries/
    └── DaftLib/
        ├── CMakeLists.txt
        ├── DaftLib.h
        └── cpp/
            └── DaftLib.cpp
```

### C++ binding code

The C++ binding code is stored inside `_pydef_pybind11/` (or `_pydef_nanobind/` if you use nanobind).

```
_pydef_pybind11/
└── module.cpp          # Main entry point of the python module
   └── pybind_DaftLib.cpp # File with bindings *generated by litgen*
```

### Python stubs

The python stubs are stored inside `_stubs/`

```
_stubs/
```

[Skip to main content](#)

```
└── __init__.py          # The python module (daft_lib) main entry point  
  └── py.typed           # (it imports and optionally adapts _daft_lib)  
                                # An empty file that indicates that the python mo
```

## Tooling for the bindings generation

```
tools/  
└── autogenerate_bindings.py  
└── change_lib_name/  
    └── change_lib_name.py
```

`tools/autogenerate_bindings.py` is the script that will generate the bindings using litgen.

`tools/change_lib_name/change_lib_name.py` is an optional utility that you can use once after cloning this template, in order to rename the libraries (e.g. from `DaftLib` to `MyLib`, `daft_lib` to `my_lib`, etc.)

## Compilation

```
└── CMakeLists.txt          # CMakeLists (used also by pip, via skbuild)  
└── litgen_cmake/  
    └── litgen_setup_module.cmake # litgen_setup_module() is a cmake function that  
                                    # the deployment of a python module with litgen  
└── requirements-dev.txt    # Requirements for development (litgen, pybind11,
```

## Deployment

`pyproject.toml` is used by pip and skbuild to build and deploy the package. It defines the name of the package, the version, the dependencies, etc.

## Continuous integration

Several github workflows are defined in `.github/workflows`:

```
.github/
```

[Skip to main content](#)

```
└── conda.yml      # Build the package with conda  
└── pip.yml        # Build the package with pip  
└── wheels.yml    # Build the wheels with cibuildwheel, and publish them to PyPI  
                  # (when a new release is created on github)
```

Note:

- cibuildwheel is configurable via options defined in the [pyproject.toml](#) file: see the [\[tool.cibuildwheel\]](#) section.
- it is also configurable via environment variables, see [cibuildwheel documentation](#)

## Tests

```
└── tests/daft_lib_test.py    # This is a list of python tests that will check  
└── pytest.ini
```

Those tests are run by cibuildwheel and by the pip CI workflow.

## Editable development mode

If you want to quickly iterate on the C++ code, and see the changes reflected in python without having to reinstall the package, you should use the python editable development mode.

### Setup editable mode

#### Step1: Install the package in editable mode

```
pip install -v -e . # -e stands for --editable, and -v stands for --verbose
```

#### Step 2: Create a standard C++ build directory

```
mkdir build && cd build  
cmake ..  
make # rebuild when you change the C++ code, and the changes will be reflected in py
```

[Skip to main content](#)

## Debug C++ bindings in editable mode

The [pybind\\_native\\_debug](#) executable provided in this template is a simple C++ program that can be used to debug the bindings in editable mode.

```
src/pybind_native_debug/
    ├── CMakeLists.txt
    ├── pybind_native_debug.cpp
    └── pybind_native_debug.py
```

Simply edit the python file [pybind\\_native\\_debug.py](#) by adding calls to the C++ functions you want to debug. Then, place breakpoints in the C++ code, and debug the C++ program.

## Development tooling

This template is ready to be used with additional tools:

- pre-commit
- ruff
- mypy
- pyright
- black
- pytest
- cibuildwheel

### pre-commit

[pre-commit](#) is a tool that allows you to run checks on your code before committing it. This template provides a default pre-commit configuration for it, but it is not active by default;

You can install pre-commit with:

[Skip to main content](#)

Then, you can activate the pre-commit hooks for your repository with:

```
pre-commit install
```

The pre-commit configuration file [.pre-commit-config.yaml](#), is configured with the following hooks:

- basic sanity checks: trailing-whitespace, end-of-file-fixer,check-yaml, check-added-large-files
- black: uncompromising Python code formatter
- ruff: fast Python linter and code formatter (only used for linting)
- mypy: static type checker for python

You can find more interesting hooks on the [pre-commit hooks repository](#), and for example add ruff, mypy, black, etc.

You may want to disable some checks in the [.pre-commit-config.yaml](#) file if you think this is too strict for your project.

## ruff: python linter and code formatter

[ruff](#) is a very fast python linter and code formatter. You can install it and run it with:

```
pip install ruff # install ruff (once)
ruff . # each time you want to check your python code
```

## mypy and pyright: static type checkers for python

[mypy](#) and [pyright](#) are static type checkers for python.

You can use either one of them, or both.

### mypy

```
pip install mypy # install mypy (once)
mypy # each time you want to check your python code
```

[Skip to main content](#)

mypy is configured via the [mypy.ini](#) file.

## pyright

```
pip install pyright # install pyright (once)
pyright # each time you want to check your python code
```

pyright is configured via the [pyrightconfig.json](#) file.

## black: python code formatter

[black](#) is a python code formatter.

```
pip install black # install black (once)
black . # each time you want to format your python code
```

## pytest: python tests

pytest is an easy-to-use python test framework.

```
pip install pytest # install pytest (once)
pytest # each time you want to run your python tests
```

It is configured via the [pytest.ini](#) file, and tests are stored in the [tests](#) folder.

## cibuildwheel: build wheels for all platforms

[ci-buildwheel](#) is a tool that allows you to build wheels for all platforms.

It is configured via the [pyproject.toml](#) file (see the [tool.cibuildwheel] section), and the [github workflow](#) file.

## run\_all\_checks

[tools/run\\_all\\_checks.sh](#) is a script you can run before committing or pushing. It will run a collection of checks (mypy, black, ruff, pytest).

## First steps

### Generate bindings for functions

Using litgen is straightforward. The simplest use case is:

1. import litgen
2. instantiate some options
3. convert some code into C++ binding code and python declarations (stubs)
4. use the generated code

Below we import litgen, define the C++ code we want to bind, and run `litgen.generate_code`

```
# Import litgen
import litgen

# Instantiate some options
options = litgen.LitgenOptions()
# Code for which we will emit bindings
cpp_code = """
    // Mathematic operations

    // Adds two integers
    int Add(int x, int y = 2);

    int Sub(int x, int y = 2); // Subtract two integers
"""

# Run the generator
generated_code = litgen.generate_code(options, cpp_code)
```

The generated code contains several elements:

[Skip to main content](#)

- `stub_code` contains the python declarations, including all comments. They enable to have a flawless code navigation and completion inside IDEs.
- `pydef_code` contains the C++ binding code. This code is specific to pybind11
- `glue_code`: optional additional C++ code that is emitted in specific advanced cases (more details later in this manual). Most of the time, it will be empty

Let's see the python declarations corresponding to the C++ code: they should be copied into a python stub file (\*.pyi) that describes the bindings offered by your library.

*Note: below, we import `litgen_demo` to be able to display code in this page. You will not need it: you can simply call `print(generated_code.stub_code)`*

```
from litgen.demo import litgen_demo

litgen_demo.demo(options, cpp_code, show_pydef=True)

# Below, you will see:
# - the original C++ code to the left
# - stub_code (i.e. the python stubs) to the right
# - pydef_code (i.e. the C++ binding code) at the bottom
# In this case, the glue code is empty
```

Original C++ decls

copy 

```
// Mathematic operations

// Adds two integers
int Add(int x, int y = 2);

int Sub(int x, int y = 2); // Subtracts two integers
```

- Corresponding python decls (stub)

copy 

```
# Mathematic operations

def add(x: int, y: int = 2) -> int:
    """ Adds two integers"""
    pass

def sub(x: int, y: int = 2) -> int:
    """ Subtract two integers"""
    pass
```

pybind11 C++ binding code

copy 

```
m.def("add",
       Add,
       py::arg("x"), py::arg("y") = 2,
       "Adds two integers");

m.def("sub",
       Sub,
       py::arg("x"), py::arg("y") = 2,
       "Subtract two integers");
```

nanobind C++ binding code

copy 

```
m.def("add",
       Add,
       nb::arg("x"), nb::arg("y") = 2,
       "Adds two integers");

m.def("sub",
       Sub,
       nb::arg("x"), nb::arg("y") = 2,
       "Subtract two integers");
```

## Binding a simple class or struct

For a simple class, public members and methods will be exposed. For a simple struct, a constructor

[Skip to main content](#)

```
options = litgen.LitgenOptions()
cpp_code = """
class FooClass {
    private:
        int mPriv = 0;
    public:
        FooClass(int v);
        int mPublic = 1;
        void ShowInfo();
};

struct FooStruc {
    int a = 1;
    void ShowInfo();
};
"""

litgen_demo.demo(options, cpp_code, show_pydef=True)
```

## Original C++ decls

copy 

```
class FooClass {
    private:
        int mPriv = 0;
    public:
        FooClass(int v);
        int mPublic = 1;
        void ShowInfo();
};

struct FooStruc {
    int a = 1;
    void ShowInfo();
};
```

## - Corresponding python decls (stub) -

copy 

```
class FooClass:
    def __init__(self, v: int) -> None:
        pass
    m_public: int = 1
    def show_info(self) -> None:
        pass

class FooStruc:
    a: int = 1
    def show_info(self) -> None:
        pass
    def __init__(self, a: int = 1) -> None:
        """Auto-generated default constructor
        pass
```

## pybind11 C++ binding code

copy 

```
auto pyClassFooClass =
py::class_<FooClass>
    (m, "FooClass", "")
    .def(py::init<int>(),
        py::arg("v"))
    .def_readwrite("m_public", &FooClass::mPublic, "")
    .def("show_info",
        &FooClass::ShowInfo)
;

auto pyClassFooStruc =
py::class_<FooStruc>
    (m, "FooStruc", "")
    .def(py::init<>([](
        int a = 1
    {
        auto r_ctor_ = std::make_unique<FooStruc>();
        r_ctor_->a = a;
        return r_ctor_;
    }))
    , py::arg("a") = 1
    )
    .def_readwrite("a", &FooStruc::a, "")
    .def("show_info",
        &FooStruc::ShowInfo)
;
```

[Skip to main content](#)

copy 

```
auto pyClassFooClass =
nb::class_<FooClass>
    (m, "FooClass", "")
.def(nb::init<int>(),
    nb::arg("v"))
.def_rw("m_public", &FooClass::mPublic, "")
.def("show_info",
    &FooClass::ShowInfo)
;

auto pyClassFooStruc =
nb::class_<FooStruc>
    (m, "FooStruc", "")
.def("__init__", [](FooStruc * self, int a = 1)
{
    new (self) FooStruc(); // placement new
    auto r_ctor_ = self;
    r_ctor_->a = a;
},
nb::arg("a") = 1
)
.def_rw("a", &FooStruc::a, "")
.def("show_info",
    &FooStruc::ShowInfo)
;
```

## Generator options

There are numerous generations options that can be set, and they are explained in details later in this manual: See [full list](#)

## About glue code

In some cases, some additional “glue code” will be emitted. This is the case for example when we want to bind a function that wants to modify a parameter whose type is immutable in python.

In this case, we will need to set one option

(`options.fn_params_replace_modifiable_immutable_by_boxed_regex`)

[Skip to main content](#)

```
options = litgen.LitgenOptions()
# This a regex which specifies that we want to adapt all functions with boxed types
options.fn_params_replace_modifiable_immutable_by_boxed_regex = r".*"
cpp_code = """
    // changes the value of the bool parameter (passed by modifiable reference)
    // (This function will use a BoxedBool in the python code, so that its value can
    void SwitchBoolValue(bool &v);
"""

# Run the generator
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy ↗

```
// changes the value of the bool  
// (This function will use a BoxedBool)  
void SwitchBoolValue(bool &v);
```

- Corresponding python decls (stub)

copy ↗

```
##### <generated_f  
class BoxedBool:  
    value: bool  
    def __init__(self, v: bool = False):  
        pass  
    def __repr__(self) -> str:  
        pass  
##### </generated_f
```

pybind11 C++ binding code

copy ↗

```
/////////// <generated_from:BoxedTypes> //////////  
auto pyClassBoxedBool =  
    py::class_<BoxedBool>  
        ("BoxedBool", "")  
        .def_readwrite("value", &BoxedBool::value, "")  
        .def(py::init<bool>(),  
            py::arg("v") = false)  
        .def("__repr__",  
            &BoxedBool::__repr__)  
;  
/////////// </generated_from:BoxedTypes> //////////  
  
m.def("switch_bool_value",  
    [](BoxedBool & v)  
    {  
        auto SwitchBoolValue_adapt_modifiable_immutable = [](BoxedBool & v)  
        {  
            bool & v_boxed_value = v.value;  
  
            SwitchBoolValue(v_boxed_value);  
        };  
  
        SwitchBoolValue_adapt_modifiable_immutable(v);  
    }.
```

[Skip to main content](#)

```
py::arg("v"),
" changes the value of the bool parameter (passed by modifiable reference)\\\"
```

*nanobind C++ binding code*

copy ↗

```
///////////////////      <generated_from:BoxedTypes>      //////////////////
auto pyClassBoxedBool =
nb::class_<BoxedBool>
(m, "BoxedBool", "")
.def_rw("value", &BoxedBool::value, "")
.def(nb::init<bool>(),
    nb::arg("v") = false)
.def("__repr__",
    &BoxedBool::__repr__)
;
///////////////////      </generated_from:BoxedTypes>      //////////////////

m.def("switch_bool_value",
[](BoxedBool & v)
{
    auto SwitchBoolValue_adapt_modifiable_immutable = [](BoxedBool & v)
    {
        bool & v_boxed_value = v.value;

        SwitchBoolValue(v_boxed_value);
    };

    SwitchBoolValue_adapt_modifiable_immutable(v);
},
nb::arg("v"),
" changes the value of the bool parameter (passed by modifiable reference)\\\"
```

*C++ glue code*

copy ↗

```
struct BoxedBool
{
    bool value;
    BoxedBool(bool v = false) : value(v) {}
    std::string __repr__() const { return std::string("BoxedBool(") + std::to_s
```

## litgen options

[Skip to main content](#)

# Note about `__regex` options

Many options in `litgen` end with the suffix `__regex`. These fields accept a **RegexOrMatcher**, i.e. either:

- a **regex string**
- or a **callable** (`Callable[[str], bool]`) that decides whether a given name matches

## Regex strings

Regex strings behave as usual in Python, except that an empty string `""` matches nothing:

- `".*"` → matches everything
- Multiple alternatives can be combined with `"|"` (regex OR)

Example: to match exactly `YourFunctionName` or any function ending in `_private`, you can use:  
`r"^\YourFunctionName$|_private$"`

*Tips:*

- Always prefix regex strings with `r` (raw string) to avoid escaping issues.
- Use anchors (`^` and `$`) when you need exact matches.

## Callables

Instead of a regex, you may also provide a callable, for example a lambda function that takes a single string argument (the name to match) and returns `True` if it matches, `False` otherwise.

Example: to match any name that ends with `_internal`, you can use:

```
options.fn_exclude_by_name__regex = lambda name: name.endswith("_internal")
```

This can be useful when patterns are easier to express with Python logic than with regex.

# litgen main options

Below, we show the content of litgen/options.py:

```
from codemanip import code_utils
from litgen.demo import litgen_demo

litgen_options_code = code_utils.download_url_content(
    "https://raw.githubusercontent.com/pthom/litgen/main/src/litgen/options.py"
)
litgen_demo.show_python_code(litgen_options_code, title="litgen/options.py")
```

copy


```
from __future__ import annotations

from enum import Enum
from typing import Any, Callable, List, TYPE_CHECKING

from codemanip import code_utils
from codemanip.code_replacements import RegexReplacementList
from codemanip.code_utils import RegexOrMatcher

from srcmlcpp import SrcmlcppOptions

from litgen.internal.template_options import TemplateFunctionsOptions, Template(
from litgen.internal.class_iterable_info import ClassIterablesInfos

if TYPE_CHECKING:
    from litgen.internal.adapted_types import AdaptedFunction, AdaptedClass
    from litgen.litgen_generator import GeneratedCodeType
```

```
class BindLibraryType(Enum):
    pybind11 = 1
    nanobind = 2
```

```
class LitgenOptions:
    """Numerous options to configure litgen code generation.

    (include / excludes, indentation, c++ to python translation settings, function
    adaptations, etc.)"""

    # -----
    # Note about __regex fields
    # -----
    # All variables ending with __regex are of type RegexOrMatcher, i.e. they can
    # - a regex string (str)
    # - or a callable (str -> bool) that decides whether a name matches
    #
    # Defaults:
    # - "" : matches nothing
    # - ".*": matches everything
    #
    # Regex usage tips:
    # - Combine alternatives with "|" (OR)
    #     e.g. r"ExactName$|_private$"
    # - Always use raw strings (prefix with "r") to avoid backslash issues
    #
```

[Skip to main content](#)

```

#
# -----
##### <bind library options>
#####
#
bind_library: BindLibraryType = BindLibraryType.pybind11

#####
#   <srcmlcpp options>
#####
# There are interesting options to set in SrcmlcppOptions (see srcmlcpp/srcmlcpp_options.py)
#
# Notably:
# * fill srcmlcpp_options.functions_api_prefixes: the prefix(es) that denote
# * also set LitgenOptions.fn_exclude_non_api=True if you want to exclude non-API functions
srcmlcpp_options: SrcmlcppOptions

#####
#   <Layout settings for the generated python stub code>
#####
#   <show the original location and or signature of elements as a comment>
original_location_flag_show = False
# if showing location, how many parent folders shall be shown
# (if -1, show the full path)
original_location_nb_parent_folders = 0
# If True, the complete C++ original signature will be show as a comment in
original_signature_flag_show = False
# Size of an indentation in the python stubs
python_indent_size = 4
python_ident_with_tabs: bool = False
# Insert as many empty lines in the python stub as found in the header file,
python_reproduce_cpp_layout: bool = True
# The generated code will try to adhere to this max length (if negative, the
python_max_line_length = 88
# Strip (remove) empty comment lines
python_strip_empty_comment_lines: bool = False
# Run black formatter
python_run_black_formatter: bool = False
python_black_formatter_line_length: int = 88

#####
#   <Layout settings for the C++ generated pydef code>
#####
# Spacing option in C++ code
cpp_indent_size: int = 4
cpp_indent_with_tabs: bool = False

#####
#   <Disable comments inclusion in C++ and Python>
#####
comments_excluded: bool = False

```

[Skip to main content](#)

```

#####
#   <names translation from C++ to python>
#####
# Convert variables, functions and namespaces names to snake_case (class, s1
python_convert_to_snake_case: bool = True
# List of code replacements when going from C++ to Python
# Notes:
# - by default, type_replacements is prefilled with standard_type_replacement
# type_replacements will be applied to all types (including class and enum)
# - by default, value_replacements is prefilled with standard_value_replacement
# - by default, comments_replacements is prefilled with standard_comments_replacement
# - by default, the others are empty
# - type_replacements, var_names_replacements and function_names_replacements
#   to modify the outputted python code
# - fn_parameters_type_replacements can be used to modify types when used as
type_replacements: RegexReplacementList # = cpp_to_python.standard_type_replacement
var_names_replacements: RegexReplacementList # = RegexReplacementList() by
namespace_names_replacements: RegexReplacementList # = RegexReplacementList()
function_names_replacements: RegexReplacementList # = RegexReplacementList()
value_replacements: RegexReplacementList # = cpp_to_python.standard_value_replacement
comments_replacements: RegexReplacementList # = cpp_to_python.standard_comments_replacement
macro_name_replacements: RegexReplacementList # = RegexReplacementList() by
fn_params_type_replacements: RegexReplacementList # = RegexReplacementList()

#####
#   <functions and method adaptations>
#####

# -----
# Exclude some functions
# -----
# Exclude certain functions and methods by a regex on their name
fn_exclude_by_name__regex: RegexOrMatcher = ""

# Exclude certain functions and methods by a regex on any of their parameters
# (those should be decorated type)
# For example:
#     options.fn_exclude_by_param_type__regex = "^\char\s*$|^unsigned\s+char"
# would exclude all functions having params of type "char *", "unsigned char"
# Note: this is distinct from `fn_params_exclude_types__regex` which removes
# from the function signature, but not the function itself.
fn_exclude_by_param_type__regex: RegexOrMatcher = ""

# Exclude function and methods by its name and signature
# For example:
#     options.fn_exclude_by_name_and_signature = {
#         "Selectable": "const char *, bool, ImGuiSelectableFlags, const ImVec2>
#     }
fn_exclude_by_name_and_signature: dict[str, str]

# -----
# Exclude some params by name or type

```

[Skip to main content](#)

```

# Remove some params from the python published interface. A param can only be
# in the C++ signature
fn_params_exclude_names_regex: RegexOrMatcher = ""
fn_params_exclude_types_regex: RegexOrMatcher = ""

# fn_exclude_non_api:
# if srcmlcpp_options.functions_api_prefixes is filled, and fn_exclude_non_api
# then only functions with an api marker will be exported.
fn_exclude_non_api: bool = True
# fn_non_api_comment:
# if fn_exclude_non_api is False, a comment can be added to non api functions
fn_non_api_comment: str = "(private API)"

# -----
# Templatized functions options
# -----
# Template function must be instantiated for the desired types.
# See https://pybind11.readthedocs.io/en/stable/advanced/functions.html#binding-templates
#
# fn_template_options:
#   of type Dict[TemplatedFunctionNameRegexStr (aka str), List[CppTypeNames]]
#
# For example,
# 1. This line:
#     options.fn_template_options.add_specialization(r"template^", ["int"])
#   would instantiate all template functions whose name end with "template"
# 2. This line:
#     options.fn_template_options.add_specialization(r".*", ["int", float"])
#   would instantiate all template functions (whatever their name) with "int"
# 3. This line:
#     options.fn_template_options.add_ignore(r".*")
#   would ignore all template functions (they will not be exported)
fn_template_options: TemplateFunctionsOptions
# if fn_template_decorate_in_stub is True, then there will be some
# decorative comments in the stub file, in order to visually group
# the generated functions together
fn_template_decorate_in_stub: bool = True

# -----
# Vectorize functions options (pybind11 only, not compatible with nanobind)
# -----
# Numeric functions (i.e. function accepting and returning only numeric parameters)
# i.e. they will accept numpy arrays as an input.
# See https://pybind11.readthedocs.io/en/stable/advanced/pycpp/numpy.html#vectorizing-functions
# and https://github.com/pybind/pybind11/blob/master/tests/test_numpy_vectorization.py
#
# * fn_vectorize_regex and fn_namespace_vectorize_regex contain a regexes
# on functions names + namespace names for which this transformation will be applied
#
# For example, to vectorize all function of the namespace MathFunctions, apply
#     options.fn_namespace_vectorize_regex: str = r"MathFunctions^$"
#     options.fn_vectorize_regex = r".*"
#

```

[Skip to main content](#)

```

#   (they can be empty, in which case the vectorized function will be a usual
fn_vectorize_regex: RegexOrMatcher = r""
fn_namespace_vectorize_regex: RegexOrMatcher = r""
fn_vectorize_prefix: str = ""
fn_vectorize_suffix: str = ""

# -----
# Return policy
# -----
# Force the function that match those regexes to use `pybind11::return_value<
# 
# Note:
#   you can also write "// py::return_value_policy::reference" as an end of
#   See packages/litgen/integration_tests/mylib/include/mylib/return_value_
fn_return_force_policy_reference_for_pointers_regex: RegexOrMatcher = ""
fn_return_force_policy_reference_for_references_regex: RegexOrMatcher = ""
#
# The callback below provides a flexible way to enforce the reference return
# It accepts litgen.internal.adapted_types.AdaptedFunction as a parameter.
# See an example in
#     src/litgen/tests/internal/adapted_types/function_policy_reference_cali
fn_return_force_policy_reference_callback: Callable[[AdaptedFunction], None] = None

# -----
# Force overload
# -----
# Force using py::overload for functions that matches these regexes
fn_force_overload_regex: RegexOrMatcher = ""
# Force using a lambda for functions that matches these regexes
# (useful when pybind11 is confused and gives error like
#   error: no matching function for call to object of type 'const detail::
fn_force_lambda_regex: RegexOrMatcher = ""

# -----
# C style buffers to py::array
# -----
#
# Signatures with a C buffer like this:
#   MY_API inline void add_inside_array(uint8_t* array, size_t array_si:
# may be transformed to:
#   void add_inside_array(py::array & array, uint8_t number_to_add)
#   def add_inside_array(array: numpy.ndarray, number_to_add: int) -> N
#
# It also works for templated buffers:
#   MY_API template<typename T> void mul_inside_array(T* array, size_t :
# will be transformed to:
#   void mul_inside_array(py::array & array, double factor)
#   def mul_inside_array(array: numpy.ndarray, factor: float) -> None
# (and factor will be down-casted to the target type)
#
# fn_params_buffer_replace_by_array_regexes contains a regex on functions na
# for which this transformation will be applied.
# Set it to "" to apply this to all functions. Set it to "None" to disable it.

```

[Skip to main content](#)

```

fn_params_replace_buffer_by_array__regex: RegexOrMatcher = r"""

# fn_params_buffer_types: list of numeric types that are considered as possible
# You can customize this list in your own options by removing items from it,
# but you *cannot* add new types or new synonyms (typedef for examples); similarly
# py::array and native relies on these *exact* names!
#
# By default, fn_params_buffer_types will contain those types:
fn_params_buffer_types: str = code_utils.join_string_by_pipe_char(
    [
        "uint8_t",
        "int8_t",
        "uint16_t",
        "int16_t",
        "uint32_t",
        "int32_t",
        "np_uint_l", # Platform dependent: "uint64_t" on *nixes, "uint32_t" on windows
        "np_int_l", # Platform dependent: "int64_t" on *nixes, "int32_t" on windows
        "float",
        "double",
        "long double",
        "long long",
    ]
)

# fn_params_buffer_template_types: list of templated names that are considered as possible
# By default, only template<typename T> or template<typename NumericType> are considered
fn_params_buffer_template_types: str = code_utils.join_string_by_pipe_char(
    [
        "nb", "size", "count", "total", "n"
    ] by default
fn_params_buffer_size_names__regex: RegexOrMatcher = code_utils.join_string_by_pipe_char(
    [
        code_utils.make_regex_var_name_contains_word("nb"),
        code_utils.make_regex_var_name_contains_word("size"),
        code_utils.make_regex_var_name_contains_word("count"),
        code_utils.make_regex_var_name_contains_word("total"),
        code_utils.make_regex_var_name_contains_word("n"),
    ]
)

# -----
# C style arrays functions and methods parameters
# -----
#
# Signatures like
#     void foo_const(const int input[2])
# may be transformed to:
#     void foo_const(const std::array<int, 2>& input)      (c++ bound signature)
#     def foo_const(input: List[int]) -> None:             (python)
# fn_params_replace_c_array_const_by_std_array__regex contains a list of regular expressions
# for which this transformation will be applied.
# Set it to "" to apply this to all functions, set it to "None" to disable it.
"""

```

[Skip to main content](#)

```

# Signatures like
#     void foo_non_const(int output[2])
# may be transformed to:
#     void foo_non_const(BoxedInt & output_0, BoxedInt & output_1)
#     def foo_non_const(output_0: BoxedInt, output_1: BoxedInt) -> None
# fn_params_replace_c_array_modifiable_by_boxed_regex contains a list of regexes
# for which this transformation will be applied.
# Set it to r".*" to apply this to all functions, set it to "" to disable it.
fn_params_replace_c_array_modifiable_by_boxed_regex: RegexOrMatcher = r".*"
# (c_array_modifiable_max_size is the maximum number of params that can be modified)
fn_params_replace_modifiable_c_array_max_size = 10

# -----
# C style string list functions and methods parameters
# -----

# Signatures like
#     void foo(const char * const items[], int items_count)
# may be transformed to:
#     void foo(const std::vector<std::string>& const items[])
#     def foo(items: List[str]) -> None
# (python code)
# fn_params_replace_c_string_list_regexes contains a list of regexes on function
# for which this transformation will be applied.
# Set it to [r".*"] to apply this to all functions, set it to [] to disable it.
fn_params_replace_c_string_list_regex: RegexOrMatcher = r".*"

# -----
# Make "immutable python types" modifiable, when passed by pointer or reference
# -----
# adapt functions params that use non const pointers or reference to a type

# Signatures like
#     int foo(int* value)
# May be transformed to:
#     def foo(BoxedInt value) -> int
# (python code)
# So that any modification done on the C++ side can be seen from python.
# -----
# fn_params_adapt_modifiable_immutable_regexes contains a list of regexes on function
# Set it to r".*" to apply this to all functions. Set it to "" to disable it.
fn_params_replace_modifiable_immutable_by_boxed_regex: RegexOrMatcher = r""

# -----
# Make "mutable default parameters" behave like C++ default arguments
# (i.e. re-evaluate the default value each time the function is called)
# -----
# There is a common pitfall in Python when using mutable default values in functions:
# if the default value is a mutable object, then it is shared between all calls
# This is because the default value is evaluated only once, when the function is defined,
# and not each time the function is called.
# -----
# This is fundamentally different from C++ default arguments, where the default value
# is evaluated each time the function is called.

```

[Skip to main content](#)

```

# However, this is not guaranteed, especially when using nanobind!
#
# Recommended settings for nanobind:
#   fn_params_adapt.mutable.param.with.default_value_to autogenerated_name
#   fn_params_adapt.mutable.param.with.default.value_regex = r".*"
# (you may call options.use_nanobind() to set these options as well as the i
#
# -----
# Regex which contains a list of regexes on functions names for which this is
# by default, this is disabled (set it to r".*" to enable it for all functions)
fn_params_adapt.mutable.param.with.default_value_regex: RegexOrMatcher = r''

# if True, auto-generated named constructors will adapt mutable default parameters
fn_params_adapt.mutable.param.with.default_value_to autogenerated_named_ctor
# if True, a comment will be added in the stub file to explain the behavior
fn_params_adapt.mutable.param.with.default_value_add_comment: bool = True
# fn_params_adapt.mutable.param.with.default_value_fn_is_known_immutable_type
# may contain a user defined function that will determine if a type is considered immutable
# By default, all the types below are considered immutable in python:
#   "int|float|double|bool|char|unsigned char|std::string|..."
fn_params_adapt.mutable.param.with.default_value_fn_is_known_immutable_type: str = ''
# Same as above, but for values
fn_params_adapt.mutable.param.with.default_value_fn_is_known_immutable_value: str = ''

# -----
# Convert `const char* x = NULL` for Python passing None without TypeError
#
# -----
# Signatures like
#   void foo(const char* text = NULL)
# may be transformed to:
#   void foo(std::optional<std::string> text = std::nullopt)
# with a lambda function wrapping around original interface.
#
# NOTE: Enable this for nanobind.
fn_params_const_char_pointer_with_default_null: bool = True

# As an alternative, we can also add the modified value to the returned type
# of the function (which will now be a tuple)
#
# For example
#   int foo(int* value)
# May be transformed to:
#   def foo(int value) -> Tuple[int, bool]
# So that any modification done on the C++ side can be seen from python.
#
# fn_params_output_modifiable_immutable_to_return_regex contains a list of
# Set it to r".*" to apply this to all functions. Set it to "" to disable it.
fn_params_output_modifiable_immutable_to_return_regex: RegexOrMatcher = ""

# -----
# Custom adapters (advanced, very advanced and not documented here)
# fn_custom_adapters may contain callables of signature
#   f(adapted_function: AdaptedFunction) -> Optional[LambdaAdapter]
#
# -----
#fn_custom_adapters: List[Any]

```

[Skip to main content](#)

```

#####
#     <class, struct, and member adaptations>
#####

# Exclude certain classes and structs by a regex on their name
class_exclude_by_name_regex: RegexOrMatcher = ""

# Exclude certain members by a regex on their name
member_exclude_by_name_regex: RegexOrMatcher = ""

# Exclude members based on their type
member_exclude_by_type_regex: RegexOrMatcher = ""

# Exclude certain members by a regex on their name, if class or struct name
# For example:
#   options.member_exclude_by_name_and_class_regex = {
#       "ImVector": join_string_by_pipe_char([
#           r"^\Size$",
#           r"^\Capacity$",
#           ...
#       ])
#   }
member_exclude_by_name_and_class_regex: dict[str, RegexOrMatcher] = {}

# Make certain members read-only by a regex on their name
member_READONLY_by_name_regex: RegexOrMatcher = ""

# Make certain members read-only based on their type
member_READONLY_by_type_regex: RegexOrMatcher = ""

# class_create_default_named_ctor_regex / struct_create_default_named_ctor_
# regex giving the list of class & struct names for which we want to generate
# constructor for Python, when no default constructor is provided by C++
# (by default, this is active for all structs and not for the classes,
# in order for it to work, all struct members need to be default constructible
# they are not declared with a default value)
struct_create_default_named_ctor_regex: RegexOrMatcher = r".*"
class_create_default_named_ctor_regex: RegexOrMatcher = r""

# class_expose_protected_methods_regex:
# regex giving the list of class names for which we want to expose protected
# (by default, only public methods are exposed)
# If active, this will use the technique described at
# https://pybind11.readthedocs.io/en/stable/advanced/classes.html#binding-protected-members
class_expose_protected_methods_regex: RegexOrMatcher = ""

# class_override_virtual_methods_in_python_regex:
# regex giving the list of class names for which we want to be able to override
# from python.
# (by default, this is not possible)
# If active, this will use the technique described at
# https://pybind11.readthedocs.io/en/stable/advanced/classes.html#overriding-virtual-methods
# Note: if you want to override protected functions, also fill `class_expose_protected_methods_regex`
class_override_virtual_methods_in_python_regex: RegexOrMatcher = ""

# class_dynamic_attributes_regex:

```

[Skip to main content](#)

```

# the ones explicitly defined using class_::def_readwrite() or class_::def_()
# If active, this will use the technique described at
# https://pybind11.readthedocs.io/en/stable/classes.html#dynamic-attributes
class_dynamic_attributes__regex: RegexOrMatcher = ""

# class_deep_copy__regex & class_copy__regex:
# By default, structs and classes exported from C++ do not support (deep)copy
# However, if they do have a copy constructor (implicit or user defined),
# (deep)copy can be enabled by invoking this constructor.
# https://pybind11.readthedocs.io/en/stable/advanced/classes.html#deepcopy-
class_deep_copy__regex: RegexOrMatcher = ""
class_copy__regex: RegexOrMatcher = ""
# If class_copy_add_info_in_stub=True, the existence of __copy__ and __deepcopy__
# will be mentioned in the stub file.
class_copy_add_info_in_stub: bool = False

# iterable classes: if some cpp classes expose begin()/end()/size(), they can
# Make classes iterables by setting:
#     options.class_iterables_infos.add_iterable_class(python_class_name_re,
class_iterables_infos: ClassIterablesInfos

# class_held_as_shared__regex:
# Regex specifying the list of class names that should be held using std::shared_ptr
# (when using pybind11. This is unused for nanobind)
#
# **Purpose:**
# By default, pybind11 uses `std::unique_ptr` as the holder type for bound objects.
#
# **When to Use:**
# If your C++ code uses `std::shared_ptr` to manage instances of a class (e.g.
# or parameters), and you expose that class to Python, you need to ensure that
# the holder type for that class.
#
# **References:**
# - [pybind11 Documentation: Smart Pointers](https://pybind11.readthedocs.io/en/stable/basics.html#smart-pointers)
# - [Understanding Holder Types in pybind11](https://pybind11.readthedocs.io/en/stable/basics.html#holder-types)
class_held_as_shared__regex: RegexOrMatcher = ""

# class_custom_inheritance_callback:
# (advanced) A callback to customize the base classes used in generated bindings.
# The first parameter is the AdaptedClass, representing the C++ class being
# The second parameter is the GeneratedCodeType, indicating whether stub or
# An example usage can be found in: src/litgen/tests/option_class_custom_inheritance_callback.cpp
class_custom_inheritance_callback: Callable[[AdaptedClass, GeneratedCodeType], None] = None

# -----
# Templatized class options
# -----
# Template class must be instantiated for the desired types, and a new name
# See https://pybind11.readthedocs.io/en/stable/advanced/classes.html#binding-templatized-classes
#
# class_template_options enables to set this
#

```

[Skip to main content](#)

```

# 1. this call would instantiate some classes for types "int" and "const char"
#   MyClass<int> (cpp)  -> MyClassInt (python)
#
# -----
#   options.class_template_options.add_specialization(
#     class_name_regex=r"^\MyPrefix",                                # r"^\MyPrefix" => :
#     cpp_types_list=["int", "const char *"],                      # instantiated type
#     naming_scheme=TemplateNamingScheme.camel_case_suffix
#   )
# 2. this call would ignore all template classes:
#   options.class_template_options.add_ignore(r".*")
#   would ignore all template functions (they will not be exported)
class_template_options: TemplateClassOptions
# if class_template_decorate_in_stub is True, then there will be some
# decorative comments in the stub file, in order to visually group
# the generated classes together
class_template_decorate_in_stub: bool = True

# -----
# Adapt class members
# -----
# adapt class members which are a fixed size array of a numeric type:
#
# For example
#   struct Foo { int values[10]; };
# May be transformed to:
#   class Foo:
#     values: numpy.ndarray
#
# i.e. the member will be transformed to a property that points to a numpy array
# which can be read/written from python (this requires numpy)
# This is active by default.
member_numeric_c_array_replace__regex: RegexOrMatcher = r".*"

# member_numeric_c_array_types: list of numeric types that can be stored in
# for a class member which is a fixed size array of a numeric type
# - Synonyms (defined via. `typedef` or `using`) are allowed here
# - *don't* include char, *don't* include byte, those are not numeric!
# See https://numpy.org/doc/stable/reference/generated/numpy.chararray.html
member_numeric_c_array_types: str = code_utils.join_string_by_pipe_char(
    [
        "int",
        "unsigned int",
        "long",
        "unsigned long",
        "long long",
        "unsigned long long",
        "float",
        "double",
        "long double",
        "uint8_t",
        "int8_t",
        "uint16_t",
        "int16_t"
    ]
)

```

[Skip to main content](#)

```

        "int32_t",
        "uint64_t",
        "int64_t",
        "bool",
    ]
)

#####
# <namespace adaptations>
#####

# All C++ namespaces in this list will not be emitted as a submodule
# (i.e. their inner code will be placed in the root python module, or in the
# module)
namespaces_root: List[str]

# All C++ namespaces that match this regex will be excluded
# By default, any namespace whose name contains "internal" or "detail" will
namespace_exclude_regex: RegexOrMatcher = r"[Ii]nternal|[Dd]etail"

#####
# <enum adaptations>
#####

# Exclude certain enums by a regex on their name
enum_exclude_by_name_regex: RegexOrMatcher = ""
# Remove the typical "EnumName_" prefix from "C enum" values.
# For example, with the C enum:
#     enum MyEnum { MyEnum_A = 0, MyEnum_B };
# Values would be named "a" and "b" in python
enum_flag_remove_values_prefix: bool = True
# A specific case for ImGui, which defines private enums which may extend to
#     enum ImGuiMyFlags_ { ImGuiMyFlags_None = 0,...}; enum ImGuiMyFlagsPr:
enum_flag_remove_values_prefix_private: bool = False

# Skip count value from enums, for example like in:
#     enum MyEnum { MyEnum_A = 1, MyEnum_B = 1, MyEnum_COUNT };
enum_flag_skip_count: bool = True
# By default, all enums export rudimentary arithmetic ( r".*" matches any enum
# (and the enum will be a derivative of enum.IntEnum)
enum_make_arithmetic_regex: RegexOrMatcher = r".*"
# Indicate that the enumeration supports bit-wise operations
# (and the enum will be a derivative of enum.IntFlag or enum.Flag)
enum_make_flag_regex: RegexOrMatcher = r"""
# Export all entries of the enumeration into the parent scope.
enum_export_values: bool = False

#####
# <define adaptations>
#####

# Simple preprocessor defines can be exported as global variables, e.g.:
#     #define MY_VALUE 1
#     #define MY_FLOAT 1.5
#     #define MY_STRING "abc"

```

[Skip to main content](#)

```

# This is limited to *simple* defines (no param, string, int, float or hex etc)
# By default nothing is exported
macro_define_include_by_name__regex: RegexOrMatcher = ""

#####
#     <globals vars adaptations>
#####
# Global variable defines can be exported as global variables, e.g.:
# By default nothing is exported (still experimental)
globals_vars_include_by_name__regex: RegexOrMatcher = ""

#####
#     <post processing>
#####
# If you need to process the code after generation, fill these functions
postprocess_stub_function: Callable[[str], str] | None = None # run at the
postprocess_pydef_function: Callable[[str], str] | None = None

#####
#     <Sanity checks and utilities below>
#####
def check_options_consistency(self) -> None:
    # the only authorized type are those for which the size is known with certainty
    # * int and long are not acceptable candidates: use int8_t, uint_8t, int16_t, ...
    # * concerning float and doubles, there is no standard for fixed size floating point types
    #   float, double and long double and their various platforms implementations
    authorized_types = [
        "byte",
        "uint8_t",
        "int8_t",
        "uint16_t",
        "int16_t",
        "uint32_t",
        "int32_t",
        "np_uint_l", # Platform dependent: "uint64_t" on *nixes, "uint32_t" on windows
        "np_int_l", # Platform dependent: "int64_t" on *nixes, "int32_t" or "int64_t" on windows
        "float",
        "double",
        "long double",
        "long long",
    ]
    for buffer_type in self._fn_params_buffer_types_list():
        if buffer_type not in authorized_types:
            raise ValueError(
                f"""
                    options.build_types contains an unauthorized type: {buffer_type}
                    Authorized types are: { ", ".join(authorized_types) }
                """
            )

def _indent_cpp_spaces(self) -> str:
    space = "\t" if self.cpp_indent_with_tabs else " "
    return space * self.cpp_indent_size

```

[Skip to main content](#)

```

def _indent_python_spaces(self) -> str:
    space = "\t" if self.python_ident_with_tabs else " "
    return space * self.python_indent_size

def _fn_params_buffer_types_list(self) -> list[str]:
    return code_utils.split_string_by_pipe_char(self.fn_params_buffer_types)

def _fn_params_buffer_template_types_list(self) -> list[str]:
    return code_utils.split_string_by_pipe_char(self.fn_params_buffer_template_types)

def _member_numeric_c_array_types_list(self) -> list[str]:
    return code_utils.split_string_by_pipe_char(self.member_numeric_c_array_types)

def use_nanobind(self) -> None:
    self.bind_library = BindLibraryType.nanobind
    self.fn_params_const_char_pointer_with_default_null = True
    self.fn_params_adapt mutable_param_with_default_value_regex = r".*"
    self.fn_params_adapt mutable_param_with_default_value_to autogenerated_value

def __init__(self) -> None:
    # See doc for all the params at their declaration site (scroll up to the
    from litgen.internal import cpp_to_python

    self.srcmlcpp_options = SrcmlcppOptions()
    self.srcmlcpp_options.header_filter_preprocessor_regions = True

    self.type_replacements = cpp_to_python.standard_type_replacements()
    self.value_replacements = cpp_to_python.standard_value_replacements()
    self.comments_replacements = cpp_to_python.standard_comment_replacements()

    self.function_names_replacements = RegexReplacementList()
    self.var_names_replacements = RegexReplacementList()
    self.macro_name_replacements = RegexReplacementList()
    self.namespace_names_replacements = RegexReplacementList()
    self.fn_params_type_replacements = RegexReplacementList()

```

## srcmlcpp options

litgen is based on srcmlcpp, which also provides some options, via

`LitgenOptions.srcmlcpp_options`. They are available at [srcmlcpp/srcmlcpp\\_options.py](#)

```

litgen_options_code = code_utils.download_url_content(
    "https://raw.githubusercontent.com/pthom/litgen/main/src/srcmlcpp/srcmlcpp_options.py")
litgen_demo.show_python_code(litgen_options_code, title="srcmlcpp/srcmlcpp_options.py")

```

[Skip to main content](#)

copy 

```
Options for srcmlcpp. Read the doc near all options elements.  
=====  
from __future__ import annotations  
from typing import Callable, Optional  
  
from codemanip.code_utils import split_string_by_pipe_char, RegexOrMatcher  
from srcmlcpp.scrml_warning_settings import WarningType  
  
  
class SrcmlcppOptions:  
    #####  
    # <API prefixes for functions / API comment suffixes for classes>  
    #####  
  
    # Prefixes that denote exported dll functions.  
    # For example, you could use "MY_API" which would be defined as `__declspec`  
    # You can have several prefixes: separate them with a "|", for example: "MY_"  
    #  
    # If you filled SrcmlcppOptions.functions_api_prefixes, then those prefixes  
    # as specifiers for the return type of the functions.  
    functions_api_prefixes: str = ""  
  
    #####  
    # <Numbers parsing: resolve macros values>  
    #####  
  
    # List of named possible numbers or sizes (fill it if some number/sizes are  
    # For example it could store `{ "SPACE_DIMENSIONS" : 3 }` if the C++ code uses  
    named_number_macros: dict[str, int]  
  
    #####  
    # <Exclude certain regions based on preprocessor macros>  
    #####  
  
    # Set header_filter_preprocessor_regions to True if the header has regions  
    # that you want to exclude from the parsing, like this:  
    #     #ifdef SOME_RARE_OPTION  
    #             // code we want to exclude  
    #     #endif  
    #  
    # See srcmlcpp/filter_preprocessor_regions.py for more complete examples  
    header_filter_preprocessor_regions: bool = False  
    # If header_filter_preprocessor_regions is True,  
    # you need to also fill header_filter_acceptable_regex in order to accept  
    # inside header_guards (and other acceptable preprocessor defines you may set)  
    # Your regex can have several options: separate them with a "|".  
    # By default, all macros names ending with "_H", "HPP", "HXX" are considered
```

[Skip to main content](#)

```

#####
#   <Custom preprocess of the code>
#####

#
# If you need to preprocess the code before parsing, fill this function
#
code_preprocess_function: Optional[Callable[[str], str]] = None

#####
#   <Misc options>
#####

# Encoding of python and C++ files
encoding: str = "utf-8"

# Preserve empty lines, i.e. any empty line in the C++ code will be mentioned
# this is done by adding a dummy comment on the line.
preserve_empty_lines: bool = True

# flag_srcml_dump_positions: if True, code positions will be outputted in tI
flag_srcml_dump_positions: bool = True

# indentation used by CppElements str_code() methods (4 spaces by default)
indent_cpp_str: str = "    "

#####
#   <Verbose / Quiet mode>
#####

# if quiet, all warning messages are discarded (warning messages go to stderror)
flag_quiet: bool = False

# List of ignored warnings
ignored_warnings: list[WarningType]
# List of ignored warnings, identified by a part of the warning message
ignored_warning_parts: list[str]

# Show python callstack when warnings are raised
flag_show_python_callstack: bool = False

# if true, display parsing progress during execution (on stdout)
flag_show_progress: bool = False

#####
# Workaround for https://github.com/srcML/srcML/issues/1833
#####
fix_brace_init_default_value = True

#####
#   <End>
#####

```

[Skip to main content](#)

```

# See doc for all the params at their declaration site (scroll up!)
self.named_number_macros = {}
self.ignored_warnings = []
self.ignored_warning_parts = []

def functions_api_prefixes_list(self) -> list[str]:
    assert isinstance(self.functions_api_prefixes, str)
    return split_string_by_pipe_char(self.functions_api_prefixes)

def _int_from_str_or_named_number_macros(options: SrcmlcppOptions, int_str: Opt:
    if int_str is None:
        return None

    try:
        v = int(int_str)
        return v
    except ValueError:
        if int_str in options.named_number_macros:
            return options.named_number_macros[int_str]
        else:
            return None

```

## Headers processing

### Filtering header content

A C/C++ header can contains different zone, some of which are parts of the public API, and some of which may correspond to specific low-level options.

litgen (and srcmlcpp) can filter a header based on preprocessor `#ifdef / #ifndef` occurrences.

Let's look at an example header: its code is defined in the `cpp_code` variable below.

```
cpp_code = """
#ifndef MY_HEADER_H    // This is an inclusion guard, it should not be filtered out
void Foo() {}

#ifndef ARCANE_OPTION
    // We are entering a zone that handle arcane options that should not be included
    void Foo2() {}
#else
    // this should also not be included in the bindings
    void Foo3() {}
#endif

#ifndef COMMON_OPTION
    // We are entering a zone for which we would like to publish bindings
    void Foo4();
#endif

#endif // #ifndef MY_HEADER_H
""";
```

Let's try to generate bindings for it:

```
import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
#ifndef MY_HEADER_H // This is an .
void Foo() {}

#ifndef ARCANEOPTION
// We are entering a zone that ha
void Foo2() {}

#else
// this should also not be includ
void Foo3() {}

#endif

#ifndef COMMONOPTION
// We are entering a zone for wh.
void Foo4();

#endif

#endif // #ifndef MY_HEADER_H
```

pybind11 C++ binding code

nanobind C++ binding code

We see that elements inside `#ifdef ARCANEOPTION` were not processed. However, elements inside `#ifdef COMMONOPTION` were not processed. Let's correct this by adjusting the options:

```
options = litgen.LitgenOptions()

# the default value for header_filter_acceptable_regex was
#     "__cplusplus|_h$_|_h$|_H$_|_H$|hpp$|HPP$|hxx$|HXX$"
# (which includes support for common header guards)
# Let's add support for COMMONOPTION
options.srcmlcpp_options.header_filter_acceptable_regex = "_H$|^COMMONOPTION$"
litgen_demo.demo(options, cpp_code)
```

- Corresponding python decls (stub)

copy ↗

```
# #ifndef MY_HEADER_H
def foo() -> None:
    pass
# #endif
```

[Skip to main content](#)

Original C++ decls

copy ↗

```
#ifndef MY_HEADER_H // This is an .

void Foo() {}

#ifndef ARCANEOPTION
    // We are entering a zone that has
    void Foo2() {}
#else
    // this should also not be included
    void Foo3() {}
#endif

#ifndef COMMONOPTION
    // We are entering a zone for which
    void Foo4();
#endif

#endif // #ifndef MY_HEADER_H
```

pybind11 C++ binding code

nanobind C++ binding code

- Corresponding python decls (stub)

copy ↗

```
# #ifndef MY_HEADER_H

def foo() -> None:
    pass

# #ifdef COMMONOPTION
#
def foo4() -> None:
    """ We are entering a zone for which
    pass
# #endif
#
# #endif
```

## API Prefixes

In a given header files, function can have an "API Prefix", that denotes whether they should be published or not in a shared library.

```
cpp_code = """
// This function has an API marker and is exported in a shared library
MY_API int add(int, int b);

// This function does not have an API marker, and is probably private
int mul(int a, int b);
"""
```

You can set the API marker in the options:

```
options = litgen.LitgenOptions()
options.srcmlcpp_options.functions_api_prefixes = "MY_API"
```

[Skip to main content](#)

Original C++ decls

copy 

```
// This function has an API marker and
MY_API int add(int, int b);

// This function does not have an API.
int mul(int a, int b);
```

pybind11 C++ binding code

nanobind C++ binding code

You can also decide to export non API function, with an optional comment.

```
options = litgen.LitgenOptions()
options.srcmlcpp_options.functions_api_prefixes = "MY_API"
options.fn_exclude_non_api = False
options.fn_non_api_comment = "Private API!"
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy 

```
// This function has an API marker and
MY_API int add(int, int b);

// This function does not have an API.
int mul(int a, int b);
```

pybind11 C++ binding code

nanobind C++ binding code

- Corresponding python decls (stub) -

copy 

```
def add(param_0: int, b: int) -> int
    """ This function has an API marker
    pass
```

## Header preprocessing

If you need to preprocess header code before the generation, you can create a function that transforms the source code, and store it inside

[Skip to main content](#)

For example:

```
def preprocess_change_int(code: str) -> str:
    return code.replace("int", "Int32") # This is a *very* dumb preprocessor

cpp_code = """
int add(int, int b);
"""

options = litgen.LitgenOptions()
options.srcmlcpp_options.code_preprocess_function = preprocess_change_int
generated_code = litgen.generate_code(options, cpp_code)
litgen_demo.show_cpp_code(generated_code.stub_code)
```

copy ↗

```
def add(param_0: Int32, b: Int32) -> Int32:
    pass
```

## Headers amalgamation

Litgen processes files individually, and if a subclass is defined in a different file than its parent, inherited members may not be correctly bound. Sometimes it is worthwhile to first generate an [Amalgamation Header](#) for a library before generating bindings for it. An amalgamation header is a single header file that includes all the public headers of a library.

## Amalgamation utility

`litgen` provides a utility function `write_amalgamate_header_file` to generate an Amalgamation header file. It is available in the `codemanip.amalgamated_header` module.

```
from codemanip import amalgamated_header
```

And its API is as follows:

[Skip to main content](#)

```

base_dir: str          # The base directory of the headers
local_includes_startwith: str # Only headers whose name begin with this string
include_subdirs: list[str] # Include only headers in these subdirectories

main_header_file: str      # The main header file
dst_amalgamated_header_file: str # The destination file

def write_amalgamate_header_file(options: AmalgamationOptions) -> None:
    ...

```

`write_amalgamate_header_file` takes an `AmalgamationOptions` object as an argument and generates an Amalgamation header file. It will include all the headers whose name starts with `local_includes_startwith` in the `base_dir` directory and its subdirectories given in `include_subdirs`.

Note: it will include any file only once: if a file was already included by another file, it will not be included again.

## A concrete example

Let's take an example with the [Hello ImGui](#) library.

This library is a C++ library that wraps the [Dear ImGui](#) library and provides additional functionalities. Its bindings are generated using the `litgen` library, and are available in [Dear ImGui Bundle](#).

It has a directory structure as shown below.

```

src
├── CMakeLists.txt
└── hello_imgui
    ├── CMakeLists.txt
    ├── app_window_params.h
    ├── hello_imgui.h          --> ( hello_imgui.h is the main header, included
    ├── imgui_window_params.h   it "#include" all other public API headers
    ├── ... (other headers)
    └── internal
        ├── borderless_movable.cpp
        ├── borderless_movable.h
        ├── clock_seconds.cpp
        ├── clock_seconds.h
        ├── ... (other headers and c++ files)
        └── ... (not part of the public API)

```

[Skip to main content](#)

And its main header file `hello_imgui.h` looks like this:

```
#pragma once

#if defined(__ANDROID__) && defined(HELLOIMGUI_USE SDL2)
// We need to include SDL, so that it can instantiate its main function under Android
#include "SDL.h" // This include should *not* be in the amalgamation header
#endif

#include "hello_imgui/dpi_aware.h"           // Only headers whose name begin with "hello_imgui"
#include "hello_imgui/hello_imgui_assets.h"    // "hello_imgui" should be included
#include "hello_imgui/hello_imgui_error.h"      // in the amalgamation header
#include "hello_imgui/hello_imgui_logger.h"
#include "hello_imgui/image_from_asset.h"
#include "hello_imgui/imgui_theme.h"
#include "hello_imgui/hello_imgui_font.h"
#include "hello_imgui/runner_params.h"
#include "hello_imgui/hello_imgui_widgets.h"

#include <string> // Other includes can be included as usual
#include <cstdint>
#include <cstddef>

... (other includes)
```

The code to generate the Amalgamation header file is as follows:

```
from codemanip import amalgamated_header

options = amalgamated_header.AmalgamationOptions()
options.base_dir = hello_imgui_src_dir
options.local_includes_startwith = "hello_imgui/"
options.include_subdirs = ["hello_imgui"]
options.main_header_file = "hello_imgui.h"
options.dst_amalgamated_header_file = PYDEF_DIR + "/hello_imgui_amalgamation.h" # The base directory of the header file
options.local_includes_startwith = "hello_imgui/" # Only headers whose name begin with "hello_imgui"
options.include_subdirs = ["hello_imgui"] # Include only headers in the subdirectories
options.main_header_file = "hello_imgui.h" # The main header file
options.dst_amalgamated_header_file = PYDEF_DIR + "/hello_imgui_amalgamation.h" # The base directory of the header file

amalgamated_header.write_amalgamate_header_file(options)
```

And the generated Amalgamation header file `hello_imgui_amalgamation.h` will look like this:

```
// THIS FILE WAS GENERATED AUTOMATICALLY. DO NOT EDIT.
```

```
////////////////////////////////////////////////////////////////
//          hello_imgui.h
////////////////////////////////////////////////////////////////
```

[Skip to main content](#)

```

// We need to include SDL, so that it can instantiate its main function under Android
#include "SDL.h"
#endif

////////////////////////////////////////////////////////////////
//           hello_imgui/dpi_aware.h included by hello_imgui.h
////////////////////////////////////////////////////////////////
#include "imgui.h"

namespace HelloImGui
{
... (content of hello_imgui/dpi_aware.h)
}
... (other includes)

```

## Generated code layout

There are numerous code layout options in the [options](#) file.

```

#####
#   <Layout settings for the generated python stub code>
#####
#   <show the original location and or signature of elements as a comment>
original_location_flag_show = False
# if showing location, how many parent folders shall be shown
# (if -1, show the full path)
original_location_nb_parent_folders = 0
# If True, the complete C++ original signature will be show as a comment in the
original_signature_flag_show = False
# Size of an indentation in the python stubs
python_indent_size = 4
python_ident_with_tabs: bool = False
# Insert as many empty lines in the python stub as found in the header file, keep
python_reproduce_cpp_layout: bool = True
# The generated code will try to adhere to this max length (if negative, this is
python_max_line_length = 80
# Strip (remove) empty comment lines
python_strip_empty_comment_lines: bool = False
# Run black formatter
python_run_black_formatter: bool = False
python_black_formatter_line_length: int = 88

#####
#   <Layout settings for the C++ generated pydef code>
#####
# Spacing option in C++ code

```

[Skip to main content](#)

```
cpp_indent_size: int = 4  
cpp_indent_with_tabs: bool = False
```

We demonstrate some of them below:

```
import litgen  
from litgen.demo import litgen_demo  
  
options = litgen.LitgenOptions()  
cpp_code = """  
    int add(int a, int b); // Adds two numbers  
"""  
  
options.cpp_indent_with_tabs = True          # The C++ code will be indented with  
options.cpp_indent_size = 1                  # one tab  
  
options.python_indent_size = 2                # The python code will be indented w.  
options.python_run_black_formatter = False   # (if black is disabled)  
  
options.original_signature_flag_show = True   # We will show the original C++ sign.  
  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy ↗

```
int add(int a, int b); // Adds two numbers
```

- Corresponding python decls (stub)

copy ↗

```
# int add(int a, int b);      /* orig */
def add(a: int, b: int) -> int:
    """ Adds two numbers"""
    pass
```

pybind11 C++ binding code

copy ↗

```
m.def("add",
       add,
       py::arg("a"), py::arg("b"),
       "Adds two numbers");
```

nanobind C++ binding code

copy ↗

```
m.def("add",
       add,
       nb::arg("a"), nb::arg("b"),
       "Adds two numbers");
```

## Names and types translation

There are numerous names and types translation options in the [options](#) file.

Below is a relevant extract from the options:

```
#####
# <names translation from C++ to python>
#####
# Convert variables, functions and namespaces names to snake_case (class, struct)
python_convert_to_snake_case: bool = True
# List of code replacements when going from C++ to Python
# Notes:
# - by default, type_replacements is prefilled with standard_type_replacements(),
#   type_replacements will be applied to all types (including class and enum names)
# - by default, value_replacements is prefilled with standard_value_replacements()
# - by default, comments_replacements is prefilled with standard_comments_replacements()
```

[Skip to main content](#)

```
# to modify the outputted python code
type_replacements: RegexReplacementList # = cpp_to_python.standard_type_replacements
var_names_replacements: RegexReplacementList # = RegexReplacementList() by default
namespace_names_replacements: RegexReplacementList # = RegexReplacementList()
function_names_replacements: RegexReplacementList # = RegexReplacementList() by default
value_replacements: RegexReplacementList # = cpp_to_python.standard_value_replacements
comments_replacements: RegexReplacementList # = cpp_to_python.standard_comment_replacements
macro_name_replacements: RegexReplacementList # = RegexReplacementList() by default
```

## Types replacements

`options.type_replacements` enables to change the way some types are exported.

Let's take an example with some C++ code.

In the example below, `MyPair` is a template class that should behave like a `std::pair`, and should be presented as a python `Tuple[int, int]`.

```
cpp_code = """
    MyPair<int, int> GetMinMax(std::vector<int>& values);
"""

```

If we convert it, we see that `std::vector<int>` is correctly interpreted as a `List[int]`, however `MyPair<int, int>` is not.

```
import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy 

`MyPair<int, int> GetMinMax(std::vector<int>& values);`

– Corresponding python decls (stub)

copy 

`def get_min_max(values: List[int]) -> None:`

pybind11 C++ binding code

+

[Skip to main content](#)

In order to account for `MyPair<int, int>`, we need to add replacements to

`options.type_replacements`:

```
options = litgen.LitgenOptions()
options.type_replacements.add_last_replacement(
    r"MyPair<(.*)\s*(.*)>", # this is a regex, with 2 captures
    r"Tuple[\1, \2]"          # and this is the replacement
)
litgen_demo.demo(options, cpp_code)
```

*Original C++ decls*

[copy ↗](#)

```
MyPair<int, int> GetMinMax(std::
```

*Corresponding python decls (stub)*

[copy ↗](#)

```
def get_min_max(values: List[int]) ->
    pass
```

*pybind11 C++ binding code*

+

*nanobind C++ binding code*

+

Note: by default `options.type_replacements` already contains standard replacements regexes.

See `standard_type_replacements()` inside `packages/litgen/internal/cpp_to_python.py`.

## Convert to snake case

The code below:

```
cpp_code = """
namespace MyNamespace
{
    struct MyClass
    {
        int AddNumber(int a, int b);

        int MultiplierRatio = 4;
    };
}
"""


```

By default, the bindings will convert CamelCase to snake\_case for functions, variables and

[Skip to main content](#)

```
import litgen

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
namespace MyNamespace
{
    struct MyClass
    {
        int AddNumber(int a, int b);
        int MultiplierRatio = 4;
    };
}
```

- Corresponding python decls (stub)

copy ↗

```
# <submodule my_namespace>
class my_namespace: # Proxy class to
    pass # (This corresponds to a C++
class MyClass:
    def add_number(self, a: int,
        pass

    multiplier_ratio: int = 4
    def __init__(self, multiplie
        """Auto-generated default
        pass

# </submodule my_namespace>
```

pybind11 C++ binding code

nanobind C++ binding code

However, you can set `options.python_convert_to_snake_case = False` to disable this.

```
options.python_convert_to_snake_case = False
litgen_demo.demo(options, cpp_code)
```

[Skip to main content](#)

Original C++ decls

copy ↗

```
namespace MyNamespace
{
    struct MyClass
    {
        int AddNumber(int a, int b);

        int MultiplierRatio = 4;
    };
}
```

pybind11 C++ binding code

nanobind C++ binding code

- Corresponding python decls (stub) -

copy ↗

```
# <submodule MyNamespace>
class MyNamespace: # Proxy class that
    pass # (This corresponds to a C++
class MyClass:
    def AddNumber(self, a: int, b):
        pass

    MultiplierRatio: int = 4
    def __init__(self, MultiplierRatio=4):
        """Auto-generated default constructor
        pass

# </submodule MyNamespace>
```

## Functions

There are numerous generations options that can be set in order to change function bindings options.

See [options.py](#): all the function related options begin wth `fn_` or `fn_params` (when they deal with function parameters)

## Exclude functions and/or params

Extract from [options.py](#), showing the related options:

```
#####
# <functions and method adaptations>
#####

# Exclude certain functions and methods by a regex on their name
fn_exclude_by_name_regex: str = ""
```

[Skip to main content](#)

```

# For example:
#     options.fn_exclude_by_param_type_regex = "^\char\s*$|^unsigned\s+char$|Ca"
# would exclude all functions having params of type "char *", "unsigned char", ...
#
# Note: this is distinct from `fn_params_exclude_types_regex` which removes pa
# from the function signature, but not the function itself.
fn_exclude_by_param_type_regex: str = ""

# -----
# Exclude some params by name or type
# -----
# Remove some params from the python published interface. A param can only be re
# in the C++ signature
fn_params_exclude_names_regex: str = ""
fn_params_exclude_types_regex: str = ""

```

**As an example, let's consider the code below, where we would want to:**

- exclude all functions beginning with "priv\_"
- exclude a function parameter if its type name starts with "Private"

```

import litgen
from litgen.demo import litgen_demo

cpp_code = """
void priv_SetOptions(bool v);

void SetOptions(const PublicOptions& options, const PrivateOptions& privateOptions)
"""

```

By default the generated code will be:

```

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)

```

Original C++ decls

copy 

```
void priv_SetOptions(bool v);  
void SetOptions(const PublicOptions&
```

- Corresponding python decls (stub)

copy 

```
def priv_set_options(v: bool) -> None:  
    pass  
  
def set_options(  
    options: PublicOptions,  
    private_options: PrivateOptions :  
) -> None:  
    pass
```

pybind11 C++ binding code

nanobind C++ binding code

But we can set some options to change this.

In the generated code below, look closely at the C++ binding code: you will see that it takes steps to generate a default value for the parameter of type `PrivateOptions`

```
options = litgen.LitgenOptions()  
options.fn_exclude_by_name_regex = "^priv_" # Exclude functions whose name begin  
options.fn_params_exclude_types_regex = "Private" # Exclude functions params whose  
  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

*Original C++ decls*

copy 

```
void priv_SetOptions(bool v);
void SetOptions(const PublicOptions&
```

*Corresponding python decls (stub)*

copy 

```
def set_options(options: PublicOptions):
    pass
```

*pybind11 C++ binding code*

copy 

```
m.def("set_options",
      [](const PublicOptions & options)
      {
          auto SetOptions_adapt_exclude_params = [] (const PublicOptions & options)
          {
              SetOptions(options, PrivateOptions());
          };

          SetOptions_adapt_exclude_params(options);
      },
      py::arg("options"));
```

*nanobind C++ binding code*

copy 

```
m.def("set_options",
      [](const PublicOptions & options)
      {
          auto SetOptions_adapt_exclude_params = [] (const PublicOptions & options)
          {
              SetOptions(options, PrivateOptions());
          };

          SetOptions_adapt_exclude_params(options);
      },
      nb::arg("options"));
```

## Return policy

See [relevant doc from pybind11](#):

Python and C++ use fundamentally different ways of managing the memory and lifetime of objects managed by them. This can lead to issues when creating bindings for functions that

[Skip to main content](#)

should take charge of the returned value and eventually free its resources, or if this is handled on the C++ side. For this reason, pybind11 provides a several return value policy annotations that can be passed to the `module_::def()` and `class_::def()` functions. The default policy is `return_value_policy::automatic`.

See [relevant doc from nanobind](#):

nanobind provides several return value policy annotations that can be passed to `module_::def()`, `class_::def()`, and `cpp_function()`...

## return\_value\_policy::reference

In the C++ code below, let's suppose that C++ is responsible for handling the destruction of the values returned by `MakeWidget` and `MakeFoo`: we do not want python to call the destructor automatically.

```
cpp_code = """  
Widget * MakeWidget();  
Foo& MakeFoo();  
"""
```

In that case, we can set `options.fn_return_force_policy_reference_for_pointers_regex` and/or `options.fn_return_force_policy_reference_for_references_regex`, and the generated pydef binding code, will set the correct return value policy.

```
options = litgen.LitgenOptions()  
options.fn_return_force_policy_reference_for_pointers_regex = r"^\w+Make"\noptions.fn_return_force_policy_reference_for_references_regex = r"^\w+Make"\nlitgen_demo.demo(options, cpp_code, show_pydef=True)
```

*Original C++ decls*

copy ↗

```
Widget * MakeWidget();
Foo& MakeFoo();
```

*Corresponding python decls (stub)*

copy ↗

```
def make_widget() -> Widget:
    pass
def make_foo() -> Foo:
    pass
```

*pybind11 C++ binding code*

copy ↗

```
m.def("make_widget",
      MakeWidget, py::return_value_policy::reference);

m.def("make_foo",
      MakeFoo, py::return_value_policy::reference);
```

*nanobind C++ binding code*

copy ↗

```
m.def("make_widget",
      MakeWidget, nb::rv_policy::reference);

m.def("make_foo",
      MakeFoo, nb::rv_policy::reference);
```

## Custom return value policy

If you annotate the function declaration with `return_value_policy::...`, then the generator will use this information:

```
cpp_code = """
Widget *MakeWidget(); // return_value_policy::take_ownership
"""

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy 

```
Widget *MakeWidget(); // return_value
```

- Corresponding python decls (stub)

copy 

```
def make_widget() -> Widget:  
    """ return_value_policy::take_ownership  
    pass
```

pybind11 C++ binding code

copy 

```
m.def("make_widget",  
      MakeWidget,  
      "return_value_policy::take_ownership",  
      py::return_value_policy::take_ownership);
```

nanobind C++ binding code

copy 

```
m.def("make_widget",  
      MakeWidget,  
      "return_value_policy::take_ownership",  
      nb::rv_policy::take_ownership);
```

## Handle mutable default param values

See ["Why are default values shared between objects?" in the Python FAQ](#)

There is a common pitfall in Python when using mutable default values in function signatures: if the default value is a mutable object, then it is shared between all calls to the function. This is because the default value is evaluated only once, when the function is defined, and not each time the function is called.

The Python code below shows this issue:

```

def use_elems(elems = []): # elems is a mutable default argument
    elems.append(1) # This will affect the default argument, for **all** subsequent
    print(elems)

use_elems() # will print [1]
use_elems() # will print [1, 1]
use_elems() # will print [1, 1, 1]

```

```

[1]
[1, 1]
[1, 1, 1]

```

This is fundamentally different from C++ default arguments, where the default value is evaluated each time the function is called.

For bound C++ functions, in most cases the default value still be reevaluated at each call. However, this is not guaranteed, especially when using nanobind!

In order to handle this, litgen provides a set of options that can be set to change the behavior of the generated code. By default, those options are disabled, and the default parameters values *might* be shared between calls (but your mileage may vary).

Recommended settings for nanobind:

- `options.fn_params_adapt_mutable_param_with_default_value_to autogenerated_named_ctor = True`
- `options.fn_params_adapt_mutable_param_with_default_value_regex = r".*"`
- you may call `options.use_nanobind()` to set these options as well as the library to nanobind

There are a few related options that can be set to change the behavior of the generated code. See the extract from `options.py` below:

```

# -----
# Make "mutable default parameters" behave like C++ default arguments
# (i.e. re-evaluate the default value each time the function is called)
# -----
# Regex which contains a list of regexes on functions names for which this trans-
# by default, this is disabled (set it to r".*" to enable it for all functions)
fn_params_adapt_mutable_param_with_default_value_regex: str = r""
# if True, auto-generated named constructors will adapt mutable default parameters

```

[Skip to main content](#)

```

fn_params_adaptMutableParamWithDefaultValue__addComment: bool = True
# fn_params_adaptMutableParamWithDefaultValue_fn_isKnownImmutableType
# may contain a user defined function that will determine if a type is considered immutable
# By default, all the types below are considered immutable in python:
#     "int|float|double|bool|char|unsigned char|std::string|..."
fn_params_adaptMutableParamWithDefaultValue_fn_isKnownImmutableType: Callable[[Any], bool]
# Same as above, but for values
fn_params_adaptMutableParamWithDefaultValue_fn_isKnownImmutableValue: Callable[[Any], bool]

```

If those options are active, litgen will by default wrap the parameter into an

`Optional[Parameter_type]`, and then check if the passed value is `None`. This step will be done for parameters that have a default value which is mutable.

In the example below, `use_elems` signature in Python becomes `use_elems(elemss: Optional[List[int]] = None)`, and the generated code will check if `elemss` is `None`, and if so, create a new list.

```

cpp_code = """
void use_elems(const std::vector<int> &elemss = {}) {
    elemss.push_back(1);
    std::cout << elemss.size() << std::endl;
}

options = litgen.LitgenOptions()
options.fn_params_adaptMutableParamWithDefaultValue_regex = r".*"
litgen_demo.demo(options, cpp_code, show_pydef=False)
"""

```

Original C++ decls

copy ↗

```

void use_elems(const std::vector<int> &elemss = {}) {
    elemss.push_back(1);
    std::cout << elemss.size() << std::endl;
}

```

- Corresponding python decls (stub)

copy ↗

```

def use_elems(elemss: Optional[List[int]] = None):
    """Python bindings defaults:
        If elemss is None, then its default value is []
    """
    pass

```

pybind11 C++ binding code

+

nanobind C++ binding code

+

Below is a more advanced example, where we use an inner struct inside another struct: the autogenerated default constructor with named params will use a wrapped `Optional` type. You can see that the behavior is nicely explained in the generated stub, as an help for the user.

[Skip to main content](#)

```

cpp_code = """
struct Inner {
    int a = 0;
    Inner(int _a) : a(_a) {}
};

struct SomeStruct {
    Inner inner = Inner(42);
};

options = litgen.LitgenOptions()
options.fn_params_adapt.mutable_param_with_default_value_regex = r".*"
options.fn_params_adapt.mutable_param_with_default_value_to autogenerated_named_ct
litgen_demo.demo(options, cpp_code, show_pydef=False)
"""

```

*Original C++ decls*

copy ↗

```

struct Inner {
    int a = 0;
    Inner(int _a) : a(_a) {}
};

struct SomeStruct {
    Inner inner = Inner(42);
};

```

*Corresponding python decls (stub)*

copy ↗

```

class Inner:
    a: int = 0
    def __init__(self, _a: int) -> None:
        pass

class SomeStruct:
    inner: Inner = Inner(42)
    def __init__(self, inner: Optional[Inner] = None):
        """Auto-generated default constructor
        If inner is None, then it is initialized to a new Inner object
        """
        self.inner = inner

```

*Python bindings defaults:*  
*If inner is None, then it is initialized to a new Inner object*

pass

*pybind11 C++ binding code*

+

*nanobind C++ binding code*

+

## Handle modifiable immutable params

Some C++ functions may use a modifiable input/output parameter, for which the corresponding type in python is **immutable** (e.g. its is a numeric type, or a string).

[Skip to main content](#)

```
void SwitchBool(bool* inOutFlag);
```

In python, a function with the following signature **can not** change its parameter value, since bool is immutable:

```
def switch_bool(in_out_v: bool) -> None:  
    pass
```

litgen offers two ways to handle those situations:

- by using boxed types
- by adding the modified value to the function output

## Using boxed types

You can decide to replace this kind of parameters type by a "Boxed" type: this is a simple class that encapsulates the value, and makes it modifiable.

Look at the example below where a `BoxedBool` is created:

- its python signature is given in the stub
- its C++ declaration is given in the glue code
- the C++ binding code handle the conversion between `bool *` and `BoxedBool`

```
cpp_code = "void SwitchBool(bool* inOutFlag);"  
options = litgen.LitgenOptions()  
options.fn_params_replace_modifiable_immutable_by_boxed_regex = r".*"\n# "Box" all  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy ↗

```
void SwitchBool(bool* inOutFlag);
```

- Corresponding python decls (stub)

copy ↗

```
#####
class BoxedBool:
    value: bool
    def __init__(self, v: bool = False):
        pass
    def __repr__(self) -> str:
        pass
#####

```

```
def switch_bool(in_out_flag: BoxedBool):
    pass
```

pybind11 C++ binding code

copy ↗

```
////////// <generated_from:BoxedTypes> //////////
auto pyClassBoxedBool =
    py::class_<BoxedBool>
        (m, "BoxedBool", "")
    .def_readwrite("value", &BoxedBool::value, "")
    .def(py::init<bool>(),
        py::arg("v") = false)
    .def("__repr__",
        &BoxedBool::__repr__)
;
////////// </generated_from:BoxedTypes> //////////

m.def("switch_bool",
    [](BoxedBool & inOutFlag)
{
    auto SwitchBool_adapt_modifiable_immutable = [](BoxedBool & inOutFlag)
    {
        bool * inOutFlag_boxed_value = & (inOutFlag.value);

        SwitchBool(inOutFlag_boxed_value);
    };

    SwitchBool_adapt_modifiable_immutable(inOutFlag);
},      py::arg("in_out_flag"));
```

nanobind C++ binding code

copy ↗

[Skip to main content](#)

```

/////////// <generated_from:BoxedTypes> //////////
auto pyClassBoxedBool =
    nb::class_<BoxedBool>
        (m, "BoxedBool", "")
    .def_rw("value", &BoxedBool::value, "")
    .def(nb::init<bool>(),
        nb::arg("v") = false)
    .def("__repr__",
        &BoxedBool::__repr__)
;
/////////// </generated_from:BoxedTypes> //////////

m.def("switch_bool",
    [](BoxedBool & inOutFlag)
{
    auto SwitchBool_adapt_modifiable_immutable = [](BoxedBool & inOutFlag)
    {
        bool * inOutFlag_boxed_value = & (inOutFlag.value);

        SwitchBool(inOutFlag_boxed_value);
    };

    SwitchBool_adapt_modifiable_immutable(inOutFlag);
},      nb::arg("in_out_flag"));

```

C++ glue code

[copy ↗](#)

```

struct BoxedBool
{
    bool value;
    BoxedBool(bool v = false) : value(v) {}
    std::string __repr__() const { return std::string("BoxedBool(") + std::to_s
};
```

## Adding the modified value to the function output

Let's say that we have a C++ function that modifies a string, and returns a bool that indicates whether it was modified:

```
bool UserInputString(std::string* inOutStr);
```

We can ask litgen to add the modified string to the output of the function.

[Skip to main content](#)

Look at the example below:

- the python function returns a `Tuple[bool, str]`
- the C++ binding code adds a lambda that does the necessary transformation

```
cpp_code = "bool UserInputString(std::string* inOutStr);"
options = litgen.LitgenOptions()
options.fn_params_output_modifiable_immutable_to_return_regex = r".*"
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy 

```
bool UserInputString(std::string* in|
```

- Corresponding python decls (stub)

copy 

```
def user_input_string(in_out_str: st  
pass
```

pybind11 C++ binding code

copy 

```
m.def("user_input_string",  
    [](std::string inOutStr) -> std::tuple<bool, std::string>  
    {  
        auto UserInputString_adapt_modifiable_immutable_to_return = [](std::str:  
        {  
            std::string * inOutStr_adapt_modifiable = & inOutStr;  
  
            bool r = UserInputString(inOutStr_adapt_modifiable);  
            return std::make_tuple(r, inOutStr);  
        };  
  
        return UserInputString_adapt_modifiable_immutable_to_return(inOutStr);  
    },  
    py::arg("in_out_str"));
```

nanobind C++ binding code

copy 

```
m.def("user_input_string",  
    [](std::string inOutStr) -> std::tuple<bool, std::string>  
    {  
        auto UserInputString_adapt_modifiable_immutable_to_return = [](std::str:  
        {  
            std::string * inOutStr_adapt_modifiable = & inOutStr;  
  
            bool r = UserInputString(inOutStr_adapt_modifiable);  
            return std::make_tuple(r, inOutStr);  
        };  
  
        return UserInputString_adapt_modifiable_immutable_to_return(inOutStr);  
    },  
    nb::arg("in_out_str"));
```

# C style function params

## Immutable C array param

If a function uses a param whose type is `const SomeType[N]`, then it will be translated automatically, and the C++ binding code will handle the necessary transformations.

```
cpp_code = "void foo(const int v[2]);"
options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy 

```
void foo(const int v[2]);
```

- Corresponding python decls (stub)

copy 

```
def foo(v: List[int]) -> None:  
    pass
```

pybind11 C++ binding code

copy 

```
m.def("foo",  
      [](const std::array<int, 2>& v)  
      {  
          auto foo_adapt_fixed_size_c_arrays = [](const std::array<int, 2>& v)  
          {  
              foo(v.data());  
          };  
  
          foo_adapt_fixed_size_c_arrays(v);  
      },  
      py::arg("v"));
```

nanobind C++ binding code

copy 

```
m.def("foo",  
      [](const std::array<int, 2>& v)  
      {  
          auto foo_adapt_fixed_size_c_arrays = [](const std::array<int, 2>& v)  
          {  
              foo(v.data());  
          };  
  
          foo_adapt_fixed_size_c_arrays(v);  
      },  
      nb::arg("v"));
```

## Modifiable C array param

If a function uses a param whose type is `SomeType[N] v`, then litgen will understand that any value inside `v` can be modified, and it will emit code where a C++ signature like this:

```
void foo(int v[2]);
```

[Skip to main content](#)

```
def foo(v_0: BoxedInt, v_1: BoxedInt) -> None:  
    pass
```

```
cpp_code = "void foo(int v[2]);"  
options = litgen.LitgenOptions()  
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
void foo(int v[2]);
```

- Corresponding python decls (stub)

copy ↗

```
##### <generated_f  
class BoxedInt:  
    value: int  
    def __init__(self, v: int = 0) :-  
        pass  
    def __repr__(self) -> str:  
        pass  
##### </generated_
```

```
def foo(v_0: BoxedInt, v_1: BoxedInt  
    pass
```

pybind11 C++ binding code

+

nanobind C++ binding code

+

C++ glue code

+

## C style string list

If a pair of function params look like `const char * const items[], int item_count`, it will be transformed into a python `List[str]`:

```
cpp_code = "void PrintItems(const char * const items[], int item_count);"  
options = litgen.LitgenOptions()  
options.fn_params_replace_c_string_list_regex = r".*" # apply to all function names  
litgen_demo.demo(options, cpp_code)
```

[Skip to main content](#)

Original C++ decls

copy 

```
void PrintItems(const char * const i
```

- Corresponding python decls (stub)

copy 

```
def print_items(items: List[str]) ->
    pass
```

+  
pybind11 C++ binding code

+  
nanobind C++ binding code

## C style variadic string format

If a function uses a pair of parameters like `char const* const format, ...`, then litgen will transform it into a simple python string.

```
cpp_code = "void Log(LogLevel level, char const* const format, ...);"
options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy 

```
void Log(LogLevel level, char const*
```

- Corresponding python decls (stub)

copy 

```
def log(level: LogLevel, format: str
    pass
```

+  
pybind11 C++ binding code

+  
nanobind C++ binding code

## Passing numeric buffers to numpy

### Simple numeric buffers

If a function uses a pair (a more) of parameters which look like `(double *values, int count)`, or `(const float* values, int nb)` (etc.). then litgen can transform this parameter into a numpy

[Skip to main content](#)

Let's see an example with this function:

```
void PlotXY(const float *xValues, const float *yValues, size_t how_many);
```

We would like it to be published as:

```
def plot_xy(x_values: np.ndarray, y_values: np.ndarray) -> None:  
    pass
```

We will need to tell litgen:

- Which function are concerned (options.fn\_params\_replace\_buffer\_by\_array\_regex)
- The name of the the “count” param if it is not a standard one (count, nb, etc)

*Note: if you look at the pybind11 C++ binding code, you will see that litgen handles the transformation, and ensures that the types are correct.*

```
cpp_code = """  
void PlotXY(const float *xValues, const float *yValues, size_t how_many);  
"""  
  
options = litgen.LitgenOptions()  
options.fn_params_replace_buffer_by_array_regex = r"^\w+Plot"  
options.fn_params_buffer_size_names_regex += "|how_many"  
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
void PlotXY(const float *xValues, co
```

– Corresponding python decls (stub)

copy ↗

```
def plot_xy(x_values: np.ndarray, y_`  
    pass
```

pybind11 C++ binding code

+

nanobind C++ binding code

+

[Skip to main content](#)

## Template numeric buffers

If a template function uses a pair of parameters whose signature looks like `(const T* values, int count)`, then it can be transformed into a numpy array.

In the example below, we would like the following C++ function:

```
template<typename NumberType> void PlotXY(Color color, const NumberType *xValues, co
```

To be published as:

```
def plot_xy(color: Color, x_values: np.ndarray, y_values: np.ndarray) -> None:  
    pass
```

For this we need to:

- Set which function names are concerned  
(`options.fn_params_replace_buffer_by_array_regex`)
- Optionally, add the name of the template param (`options.fn_params_buffer_template_types`)

*Note: if you look at the generated pybind11 C++ binding code, you will see that it handles all numeric types. This is a very efficient way to transmit numeric buffers of all types to python*

```
cpp_code = """  
    template<typename NumberType>  
    void PlotXY(Color color, const NumberType *xValues, const NumberType *yValues,  
    """  
  
options = litgen.LitgenOptions()  
options.fn_params_replace_buffer_by_array_regex = r"^\w+Plot"  
options.fn_params_buffer_template_types += "|NumberType"  
litgen_demo.demo(options, cpp_code, height=80)
```

Original C++ decls

copy ↗

```
template<typename NumberType>
void PlotXY(Color color, const N
```

- Corresponding python decls (stub)

copy ↗

```
def plot_xy(color: Color, x_values: |  
    pass
```

pybind11 C++ binding code

nanobind C++ binding code

+

+

## Vectorize functions

See relevant portion of the [pybind11 manual](#). **This feature is not available with nanobind**

Within litgen, you can set:

- Which namespaces are candidates for vectorization (options.fn\_namespace\_vectorize\_\_regex.  
Set it to `r".*"` for all namespaces)
- Which function names are candidates for vectorization
- Which optional suffix or prefix will be added to the vectorized functions

```
cpp_code = """
namespace MathFunctions
{
    double fn1(double x, double y);
    double fn2(double x);
}
"""

options = litgen.LitgenOptions()
options.fn_namespace_vectorize__regex = "^\$MathFunctions\$"
options.fn_vectorize__regex = r".*"
options.fn_vectorize_suffix = "_v"
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
namespace MathFunctions
{
    double fn1(double x, double y);
    double fn2(double x);
}
```

- Corresponding python decls (stub) -

copy ↗

```
# <submodule math_functions>
class math_functions: # Proxy class
    pass # (This corresponds to a C++
@staticmethod
def fn1(x: float, y: float) -> float:
    pass
@staticmethod
def fn1_v(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    pass
@staticmethod
def fn2(x: float) -> float:
    pass
@staticmethod
def fn2_v(x: np.ndarray) -> np.ndarray:
    pass

# </submodule math_functions>
```

pybind11 C++ binding code +

nanobind C++ binding code +

## Accepting args and kwargs

Relevant portion of the [pybind11 manual](#) and the [nanobind manual](#)

litgen will automatically detect signatures with params which look like `(py::args args, const py::kwargs& kwargs)` or `(nb::args args, const nb::kwargs& kwargs)` and adapt the python stub signature accordingly.

```

cpp_code = """
void generic_pybind(py::args args, const py::kwargs& kwargs)
{
    /// .. do something with args
    // if (kwargs)
        /// .. do something with kwargs
}

void generic_nanobind(nb::args args, const nb::kwargs& kwargs)
{
    /// .. do something with args
    // if (kwargs)
        /// .. do something with kwargs
}
"""

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)

```

*Original C++ decls*

copy ↗

```

void generic_pybind(py::args args, c
{
    /// .. do something with args
    // if (kwargs)
        /// .. do something with kwa
}

void generic_nanobind(nb::args args,
{
    /// .. do something with args
    // if (kwargs)
        /// .. do something with kwa
}

```

*pybind11 C++ binding code*

*nanobind C++ binding code*

*- Corresponding python decls (stub)*

copy ↗

```

def generic_pybind(*args, **kwargs) :
    pass

def generic_nanobind(*args, **kwargs):
    pass

```

## Force overload

*Relevant portion of the [pybind11 manual](#) and the [nanobind manual](#)*

[Skip to main content](#)

## Automatic overload

If litgen detect two overload, it will add a call to `py::overload_cast` automatically:

```
cpp_code = """  
void foo(int x);  
void foo(double x);  
"""  
  
options = litgen.LitgenOptions()  
litgen_demo.demo(options, cpp_code)
```

*Original C++ decls*

`copy`

```
void foo(int x);  
void foo(double x);
```

*Corresponding python decls (stub)*

`copy`

```
@overload  
def foo(x: int) -> None:  
    pass  
@overload  
def foo(x: float) -> None:  
    pass
```

*pybind11 C++ binding code*

*nanobind C++ binding code*

## Manual overload

However, in some cases, you might want to add it manually: use

`options.fn_force_overload_regex`

```
cpp_code = """  
void foo2(int x);  
"""  
  
options = litgen.LitgenOptions()  
options.fn_force_overload_regex += r"|^foo2$"  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

[Skip to main content](#)

*Original C++ decls*

copy ↗

```
void foo2(int x);
```

*Corresponding python decls (stub)*

copy ↗

```
def foo2(x: int) -> None:  
    pass
```

*pybind11 C++ binding code*

copy ↗

```
m.def("foo2",  
      py::overload_cast<int>(foo2), py::arg("x"));
```

*nanobind C++ binding code*

copy ↗

```
m.def("foo2",  
      nb::overload_cast<int>(foo2), nb::arg("x"));
```

## Force usage of a lambda function

In some rare cases, the usage of `py::overload_cast` might not be sufficient to discriminate the overload. In this case, you can tell litgen to disambiguate it via a lambda function. Look at the pybind C++ binding code below:

```
cpp_code = """  
void foo3(int x);  
"""  
  
options = litgen.LitgenOptions()  
options.fn_force_lambda_regex += r"|^foo3$"  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

*Original C++ decls*

copy 

```
void foo3(int x);
```

- *Corresponding python decls (stub)*

copy 

```
def foo3(x: int) -> None:  
    pass
```

*pybind11 C++ binding code*

copy 

```
m.def("foo3",  
      [](int x)  
      {  
          auto foo3_adapt_force_lambda = [](int x)  
          {  
              foo3(x);  
          };  
  
          foo3_adapt_force_lambda(x);  
      },  
      py::arg("x"));
```

*nanobind C++ binding code*

copy 

```
m.def("foo3",  
      [](int x)  
      {  
          auto foo3_adapt_force_lambda = [](int x)  
          {  
              foo3(x);  
          };  
  
          foo3_adapt_force_lambda(x);  
      },  
      nb::arg("x"));
```

## Classes and structs

### Exclude members and classes

Sometimes, you may want to exclude classes and/or members from the bindings. Look at the example below for instructions.

[Skip to main content](#)

```

import litgen
from litgen.demo import litgen_demo

cpp_code = """
    class FooDetails // A class that we want to exclude from the bindings
    {
        // ...
    };

    struct Foo
    {
        int X = 0, Y = 1;

        FooDetails mDetails = {}; // A member that we would want to exclude from the bindings
    };
"""

options = litgen.LitgenOptions()
options.class_exclude_by_name_regex = r"Details$"
options.member_exclude_by_name_regex = r"Details$"
litgen_demo.demo(options, cpp_code)

```

*Original C++ decls*

copy ↗

```

class FooDetails // A class that we want to exclude from the bindings
{
    // ...
};

struct Foo
{
    int X = 0, Y = 1;

    FooDetails mDetails = {};// A member that we would want to exclude from the bindings
};

```

*pybind11 C++ binding code*

*nanobind C++ binding code*

*Corresponding python decls (stub)*

copy ↗

```

class Foo:
    x: int = 0
    y: int = 1

def __init__(self, x: int = 0, y: int = 1):
    """Auto-generated default constructor
    pass

```

## Default constructor with named params

litgen will automatically generate a default constructor with named params for structs. For classes,

[Skip to main content](#)

only if the class/struct does not provide a default constructor.

See example below:

```
cpp_code = """
enum class Options { A, B };

// A constructor with named params will be created for FooClass,
// since options.class_create_default_named_ctor_regex was filled
class FooClass {
public:
    Options options = Options::A;
    int a = 1;
};

// A constructor with named params will be created for FooStruct
struct FooStruct { int X = 0, Y = 1; };

// FooStruct has a default constructor, so no constructor with named params will
// be created
struct FooStruct2 {
    FooStruct2();
    int X = 0, Y = 1;
};
"""

options = litgen.LitgenOptions()
# options.struct_create_default_named_ctor_regex = r".*"
options.class_create_default_named_ctor_regex = r".*"
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
enum class Options { A, B };

// A constructor with named parameters
// since options.class_create_default()
class FooClass {
public:
    Options options = Options::A;
    int a = 1;
};

// A constructor with named parameters
struct FooStruct { int X = 0, Y = 1 };

// FooStruct has a default constructor
struct FooStruct2 {
    FooStruct2();
    int X = 0, Y = 1;
};
```

- Corresponding python decls (stub) -

copy ↗

```
class Options(enum.IntEnum):
    a = enum.auto() # (= 0)
    b = enum.auto() # (= 1)

class FooClass:
    """ A constructor with named parameters
    since options.class_create_default()
    """
    options: Options = Options.a
    a: int = 1
    def __init__(self, options: Options):
        """Auto-generated default constructor
        pass

class FooStruct:
    """ A constructor with named parameters
    x: int = 0
    y: int = 1
    def __init__(self, x: int = 0, y: int = 1):
        """Auto-generated default constructor
        pass

class FooStruct2:
    """ FooStruct has a default constructor
    def __init__(self) -> None:
        pass
    x: int = 0
    y: int = 1
```

pybind11 C++ binding code

nanobind C++ binding code

+

+

## Expose protected member functions

Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)

Exposing protected member functions requires the creation of a “Publicist” helper class. litgen enables to automate this:

[Skip to main content](#)

```
cpp_code = """  
class A {  
protected:  
    int foo() const { return 42; }  
};  
"""  
  
options = litgen.LitgenOptions()  
options.class_expose_protected_methods__regex = "^A$"  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

## Original C++ decls

copy 

```
class A {  
protected:  
    int foo() const { return 42; }  
};
```

## - Corresponding python decls (stub)

copy 

```
class A:  
    def __init__(self) -> None:  
        """Autogenerated default con  
        pass  
  
    # <protected_methods>  
    def foo(self) -> int:  
        pass  
    # </protected_methods>
```

## pybind11 C++ binding code

copy 

```
auto pyClassA =  
    py::class_<A>  
        (m, "A", "")  
    .def(py::init<>()) // implicit default constructor  
    .def("foo",  
        &A_publicist::foo)  
;
```

## nanobind C++ binding code

copy 

```
auto pyClassA =  
    nb::class_<A>  
        (m, "A", "")  
    .def(nb::init<>()) // implicit default constructor  
    .def("foo",  
        &A_publicist::foo)  
;
```

## C++ glue code

copy 

```
// helper type for exposing protected functions  
class A_publicist : public A  
{  
public:  
    using A::foo;  
};
```

[Skip to main content](#)

# Overriding virtual methods in Python

*Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)*

In order to override virtual methods in Python, you need to create of a trampoline class, which can be a bit cumbersome.

litgen can automate this: look at the pybind11 binding code, and at the glue code below.

```
cpp_code = """  
class Animal {  
public:  
    virtual ~Animal() {}  
    virtual std::string go(int n_times) = 0;  
};  
""";  
  
options = litgen.LitgenOptions()  
options.class_override_virtual_methods_in_python_regex = "^.+Animal$"  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

## Original C++ decls

copy 

```
class Animal {
public:
    virtual ~Animal() { }
    virtual std::string go(int n);
};
```

## - Corresponding python decls (stub)

copy 

```
class Animal:
    def go(self, n_times: int) -> str
    pass
def __init__(self) -> None:
    """Autogenerated default constructor
    pass
```

## pybind11 C++ binding code

copy 

```
auto pyClassAnimal =
    py::class_<Animal, Animal_trampoline>
        (m, "Animal", "")
    .def(py::init<>()) // implicit default constructor
    .def("go",
        &Animal::go, py::arg("n_times"))
    ;
```

## nanobind C++ binding code

copy 

```
auto pyClassAnimal =
    nb::class_<Animal, Animal_trampoline>
        (m, "Animal", "")
    .def(nb::init<>()) // implicit default constructor
    .def("go",
        &Animal::go, nb::arg("n_times"))
    ;
```

## C++ glue code

copy 

```
// helper type to enable overriding virtual methods in python
class Animal_trampoline : public Animal
{
public:
    using Animal::Animal;

    std::string go(int n_times) override
    {
        PYBIND11_OVERRIDE_PURE_NAME(
            std::string, // return type
```

[Skip to main content](#)

```

        go, // function name (c++)
        n_times // params
    );
}
};

```

Note: in the case of nanobind, the glue code will differ a bit. It is shown below:

```

options.bind_library = litgen.BindLibraryType.nanobind
generated_code = litgen.generate_code(options, cpp_code)
litgen_demo.show_cpp_code(generated_code.glue_code, "Glue code with nanobind")

```

*Glue code with nanobind*

copy ↗

```

// helper type to enable overriding virtual methods in python
class Animal_trampoline : public Animal
{
public:
    NB_TRAMPOLINE(Animal, 1);

    std::string go(int n_times) override
    {
        NB_OVERRIDE_PURE_NAME(
            "go", // function name (python)
            go, // function name (c++)
            n_times // params
        );
    }
};

```

## Combining virtual functions and inheritance

*Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)*

```
cpp_code = """
    class Animal {
public:
    virtual std::string go(int n_times) = 0;
    virtual std::string name() { return "unknown"; }
};

    class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += bark() + " ";
        return result;
    }
    virtual std::string bark() { return "woof!"; }
};
"""

options = litgen.LitgenOptions()
options.class_override_virtual_methods_in_python_regex = "^Animal$|^Dog$"
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

## Original C++ decls

copy ↗

```
class Animal {
public:
    virtual std::string go(int n);
    virtual std::string name();
};

class Dog : public Animal {
public:
    std::string go(int n_times) {
        std::string result;
        for (int i=0; i<n_times;)
            result += bark() + " ";
        return result;
    }
    virtual std::string bark() {
    };
}
```

## - Corresponding python decls (stub)

copy ↗

```
class Animal:
    def go(self, n_times: int) -> str:
        pass
    def name(self) -> str:
        pass
    def __init__(self) -> None:
        """Autogenerated default constructor
        pass

class Dog(Animal):
    def go(self, n_times: int) -> str:
        pass
    def bark(self) -> str:
        pass
    def __init__(self) -> None:
        """Autogenerated default constructor
        pass
```

## pybind11 C++ binding code

copy ↗

```
auto pyClassAnimal =
py::class_<Animal, Animal_trampoline>
    (m, "Animal", "")
.def(py::init<>()) // implicit default constructor
.def("go",
    &Animal::go, py::arg("n_times"))
.def("name",
    &Animal::name)
;

auto pyClassDog =
py::class_<Dog, Animal, Dog_trampoline>
    (m, "Dog", "")
.def(py::init<>()) // implicit default constructor
.def("go",
    &Dog::go, py::arg("n_times"))
.def("bark",
    &Dog::bark)
;
```

## nanobind C++ binding code

copy ↗

[Skip to main content](#)

```

auto pyClassAnimal =
    nb::class_<Animal, Animal_trampoline>
        (m, "Animal", "")
    .def(nb::init<>()) // implicit default constructor
    .def("go",
        &Animal::go, nb::arg("n_times"))
    .def("name",
        &Animal::name)
;

auto pyClassDog =
    nb::class_<Dog, Animal, Dog_trampoline>
        (m, "Dog", "")
    .def(nb::init<>()) // implicit default constructor
    .def("go",
        &Dog::go, nb::arg("n_times"))
    .def("bark",
        &Dog::bark)
;

```

C++ glue code

copy ↗

```

// helper type to enable overriding virtual methods in python
class Animal_trampoline : public Animal
{
public:
    using Animal::Animal;

    std::string go(int n_times) override
    {
        PYBIND11_OVERRIDE_PURE_NAME(
            std::string, // return type
            Animal, // parent class
            "go", // function name (python)
            go, // function name (c++)
            n_times // params
        );
    }
    std::string name() override
    {
        PYBIND11_OVERRIDE_NAME(
            std::string, // return type
            Animal, // parent class
            "name", // function name (python)
            name // function name (c++)
        );
    }
};

```

[Skip to main content](#)

```
// helper type to enable overriding virtual methods in python
class Dog_trampoline : public Dog
{
public:
```

Note: in the case of nanobind, the glue code will differ a bit. It is shown below:

```
options.bind_library = litgen.BindLibraryType.nanobind
generated_code = litgen.generate_code(options, cpp_code)
litgen_demo.show_cpp_code(generated_code.glue_code, "Glue code with nanobind")
```

## Glue code with nanobind

copy ↗

```
// helper type to enable overriding virtual methods in python
class Animal_trampoline : public Animal
{
public:
    NB_TRAMPOLINE(Animal, 2);

    std::string go(int n_times) override
    {
        NB_OVERRIDE_NAME(
            "go", // function name (python)
            go, // function name (c++)
            n_times // params
        );
    }
    std::string name() override
    {
        NB_OVERRIDE_NAME(
            "name", // function name (python)
            name // function name (c++)
        );
    }
};
```

```
// helper type to enable overriding virtual methods in python
class Dog_trampoline : public Dog
{
public:
    NB_TRAMPOLINE(Dog, 3);

    std::string go(int n_times) override
    {
        NB_OVERRIDE_NAME(
            "go", // function name (python)
            go, // function name (c++)
            n_times // params
        );
    }
    std::string bark() override
    {
        NB_OVERRIDE_NAME(
            "bark", // function name (python)
            bark // function name (c++)
        );
    }
    std::string name() override
    {
```

[Skip to main content](#)

```

        name // function name (c++)
    );
}
};
```

# Operator overloading

litgen is able to automatically transform C++ numerical operators into their corresponding dunder function in Python.

## Overloading addition, subtraction, etc.

See example below:

```

cpp_code = """
    struct IntWrapper
    {
        int value;
        IntWrapper(int v) : value(v) {}

        // arithmetic operators
        IntWrapper operator+(IntWrapper b) { return IntWrapper{ value + b.value };
        IntWrapper operator-(IntWrapper b) { return IntWrapper{ value - b.value };

        // Unary minus operator
        IntWrapper operator-() { return IntWrapper{ -value }; }

        // Comparison operator
        bool operator<(IntWrapper b) { return value < b.value; }

        // Two overload of the += operator
        IntWrapper operator+=(IntWrapper b) { value += b.value; return *this; }
        IntWrapper operator+=(int b) { value += b; return *this; }

        // Two overload of the call operator, with different results
        int operator()(IntWrapper b) { return value * b.value + 2; }
        int operator()(int b) { return value * b + 3; }
    };
"""

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code, height=60)
```

Original C++ decls

copy ↗

```
struct IntWrapper
{
    int value;
    IntWrapper(int v) : value(v)

    // arithmetic operators
    IntWrapper operator+(IntWrapper v) : value + v
    IntWrapper operator-(IntWrapper v) : value - v

    // Unary minus operator
    IntWrapper operator-() {
        return IntWrapper(-value);
    }

    // Comparison operator
    bool operator<(IntWrapper v) {
        return value < v.value;
    }

    // Two overload of the + operator
    IntWrapper operator+=(IntWrapper v) {
        return IntWrapper(value + v.value);
    }
    IntWrapper operator+=(int v) {
        return IntWrapper(value + v);
    }

    // Two overload of the call operator
    int operator()(IntWrapper v) {
        return value + v.value;
    }
    int operator()(int v) {
        return value + v;
    }
};
```

- Corresponding python decls (stub)

copy ↗

```
class IntWrapper:
    value: int
    def __init__(self, v: int) -> None:
        pass

    # arithmetic operators
    def __add__(self, b: IntWrapper) -> IntWrapper:
        pass
    @overload
    def __sub__(self, b: IntWrapper) -> IntWrapper:
        pass

    @overload
    def __neg__(self) -> IntWrapper:
        """ Unary minus operator """
        pass

    def __lt__(self, b: IntWrapper) -> bool:
        """ Comparison operator """
        pass

    # Two overload of the += operator
    @overload
    def __iadd__(self, b: IntWrapper) -> IntWrapper:
        pass
    @overload
    def __iadd__(self, b: int) -> IntWrapper:
        pass

    # Two overload of the call operator
    @overload
    def __call__(self, b: IntWrapper) -> int:
        pass
    @overload
    def __call__(self, b: int) -> int:
        pass
```

pybind11 C++ binding code

nanobind C++ binding code

[Skip to main content](#)

## Overloading comparisons with the spaceship operator

```
cpp_code = """
    struct Point
    {
        int x;
        int y;
        auto operator<=>(const Point&) const = default;
    };
"""

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
struct Point
{
    int x;
    int y;
    auto operator<=>(const Point&)
};
```

- Corresponding python decls (stub) -

copy ↗

```
class Point:
    x: int
    y: int
    def __lt__(self, param_0: Point)
        """(C++ auto return type)
    """
    pass
    def __le__(self, param_0: Point)
        """(C++ auto return type)
    """
    pass
    def __eq__(self, param_0: Point)
        """(C++ auto return type)
    """
    pass
    def __ge__(self, param_0: Point)
        """(C++ auto return type)
    """
    pass
    def __gt__(self, param_0: Point)
        """(C++ auto return type)
    """
    pass
    def __init__(self, x: int = int(
        """Auto-generated default co
    pass
```

pybind11 C++ binding code

+

nanobind C++ binding code

+

## Dynamic attributes

Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)

The python class `Pet` below will be able to pick up new attributes dynamically:

[Skip to main content](#)

```

cpp_code = """
struct Pet {
    Pet(const std::string &name) : name(name) { }
    std::string name;
};

options = litgen.LitgenOptions()
options.class_dynamic_attributes_regex = "^\$Pet$"
litgen_demo.demo(options, cpp_code, show_pydef=True)
"""

```

*Original C++ decls*

copy 

```

struct Pet {
    Pet(const std::string &name) : name(name) { }
    std::string name;
};

```

*Corresponding python decls (stub)*

copy 

```

class Pet:
    def __init__(self, name: str) ->
        pass
    name: str

```

*pybind11 C++ binding code*

copy 

```

auto pyClassPet =
    py::class_<Pet>
        (m, "Pet", py::dynamic_attr(), "")
    .def(py::init<const std::string &>(),
        py::arg("name"))
    .def_readwrite("name", &Pet::name, "")
;

```

*nanobind C++ binding code*

copy 

```

auto pyClassPet =
    nb::class_<Pet>
        (m, "Pet", nb::dynamic_attr(), "")
    .def(nb::init<const std::string &>(),
        nb::arg("name"))
    .def_rw("name", &Pet::name, "")
;

```

## Copy and Deep copy

[Skip to main content](#)

See below for instructions on how to add support for copy and deepcopy.

```
cpp_code = """  
    struct Foo { std::vector<int> values; };  
    struct Foo2 {  
        Foo foo1 = Foo();  
        Foo foo2 = Foo();  
    };  
....  
  
options = litgen.LitgenOptions()  
options.class_copy_regex = "^Foo$|^Foo2$"  
options.class_deep_copy_regex = "^Foo2$"  
options.class_copy_add_info_in_stub = True  
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
struct Foo { std::vector<int> va  
struct Foo2 {  
    Foo foo1 = Foo();  
    Foo foo2 = Foo();  
};
```

- Corresponding python decls (stub) -

copy ↗

```
class Foo:  
....  
    (has support for copy.copy)  
....  
    values: List[int]  
def __init__(self, values: List[  
    """Auto-generated default co  
    pass  
class Foo2:  
....  
    (has support for copy.copy and co  
....  
    foo1: Foo = Foo()  
    foo2: Foo = Foo()  
def __init__(self, foo1: Foo = Fo  
    """Auto-generated default co  
    pass
```

pybind11 C++ binding code

+

nanobind C++ binding code

+

## Iterable classes

It is possible to make custom container classes iterable in python. See example below.

[Skip to main content](#)

```
cpp_code = """
class MyContainer
{
public:
    float operator[](int idx);

    // We need to have defined begin(), end(), and size() to make the class iterable
    iterator begin(); // This function is excluded from the bindings (see options.f
    iterator end(); // This function is excluded from the bindings (see options.f
    size_t size(); // This function is also published as __len__
private:
    std::vector<float> values;
};

"""

options = litgen.LitgenOptions()
options.class_iterables_infos.add_iterable_class("MyContainer", "float")
options.fn_exclude_by_name_regex = "^begin$|^end$"
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

## Original C++ decls

copy 

```
class MyContainer
{
public:
    float operator[](int idx);

    // We need to have defined begin
    iterator begin(); // This function
    iterator end();   // This function
    size_t size();    // This function
private:
    std::vector<float> values;
};
```

## - Corresponding python decls (stub)

copy 

```
class MyContainer:
    def __getitem__(self, idx: int):
        pass

    # We need to have defined begin()
    def size(self) -> int:
        """ This function is also published as __len__
        pass
    def __init__(self) -> None:
        """Autogenerated default constructor
        pass
    def __iter__(self) -> Iterator[float]:
        pass
    def __len__(self) -> int:
        pass
```

## pybind11 C++ binding code

copy 

```
auto pyClassMyContainer =
    py::class_<MyContainer>
        (m, "MyContainer", "")
    .def(py::init<>()) // implicit default constructor
    .def("__getitem__",
        &MyContainer::operator[], py::arg("idx"))
    .def("size",
        &MyContainer::size, "This function is also published as __len__")
    .def("__iter__", [](const MyContainer &v) { return py::make_iterator(v.begin()); })
    .def("__len__", [](const MyContainer &v) { return v.size(); })
;
```

## nanobind C++ binding code

copy 

```
auto pyClassMyContainer =
    nb::class_<MyContainer>
        (m, "MyContainer", "")
    .def(nb::init<>()) // implicit default constructor
    .def("__getitem__",
        &MyContainer::operator[], nb::arg("idx"))
    .def("size",
        &MyContainer::size, "This function is also published as __len__")
    .def("__iter__", [](const MyContainer &v) { ... })
```

[Skip to main content](#)

```
.def("__len__", [](const MyContainer &v) { return v.size(); })  
;
```

## Numeric C array members

If a struct/class stores a numeric C array member, it will be exposed as a *modifiable* numpy array.

```
cpp_code = """  
struct Foo { int values[4]; };  
"""  
  
options = litgen.LitgenOptions()  
# options.member_numeric_c_array_replace_regex = r".*" # this is the default  
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy 

```
struct Foo { int values[4]; };
```

- Corresponding python decls (stub) -

copy 

```
class Foo:  
    values: np.ndarray # ndarray[ty]  
    def __init__(self) -> None:  
        """Auto-generated default con  
        pass
```

pybind11 C++ binding code

copy 

```
auto pyClassFoo =  
    py::class_<Foo>  
        (m, "Foo", "")  
    .def(py::init<>()) // implicit default constructor  
    .def_property("values",  
        [](Foo &self) -> pybind11::array  
    {  
        auto dtype = pybind11::dtype(pybind11::format_descriptor<int>::format());  
        auto base = pybind11::array(dtype, {4}, {sizeof(int)});  
        return pybind11::array(dtype, {4}, {sizeof(int)}, self.values, base);  
    }, [](Foo& self) {},  
    "")  
;
```

nanobind C++ binding code

copy 

```
auto pyClassFoo =  
    nb::class_<Foo>  
        (m, "Foo", "")  
    .def(nb::init<>()) // implicit default constructor  
    .def_prop_ro("values",  
        [](Foo &self) -> nb::ndarray<int, nb::numpy, nb::shape<4>, nb::c_contig>,  
    {  
        return self.values;  
    },  
    "")  
;
```

## Inner class or enum

[Skip to main content](#)

```

cpp_code = """
    struct Foo
    {
        enum class Choice { A, B };
        int HandleChoice(Choice value = Choice::A) { return 0; }
    };
"""

options = litgen.LitgenOptions()
litgen_demo.demo(options, cpp_code)

```

*Original C++ decls*

copy ↗

```

struct Foo
{
    enum class Choice { A, B };
    int HandleChoice(Choice value
};
```

*Corresponding python decls (stub)*

copy ↗

```

class Foo:
    class Choice(enum.IntEnum):
        a = enum.auto() # (= 0)
        b = enum.auto() # (= 1)
    def handle_choice(self, value: Foo.Choice):
        pass
    def __init__(self) -> None:
        """Auto-generated default constructor
        pass
```

*pybind11 C++ binding code*

*nanobind C++ binding code*

+

+

## Template classes and functions

litgen provides advanced support for template classes and functions. Refer to the examples below.

## Template Functions

*Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)*

litgen can instantiate template functions for a customizable range of types.

[Skip to main content](#)

## Export template functions with an @overload decorator

Consider the example below. If we try to generate code from it, litgen will complain that this template function is unhandled:

```
cpp_code = """
    template<typename T> T MaxValue(const std::vector<T>& values);
"""

import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
generated_code = litgen.generate_code(options, cpp_code)
```

```
Warning: (LitgenTemplateFunctionIgnore) Ignoring template function MaxValue. You might
While parsing a "function_decl", corresponding to this C++ code:
Position:2:5
```

```
    template<typename T> T MaxValue(const std::vector<T>& values);
    ^
```

```
Warning: (LitgenTemplateFunctionIgnore) Ignoring template function MaxValue. You might
While parsing a "function_decl", corresponding to this C++ code:
Position:2:5
```

```
    template<typename T> T MaxValue(const std::vector<T>& values);
    ^
```

If we add some information about how we want to specialize the function, then litgen will correctly output the bindings, and it will add an `@overload` decorator to the python functions.

```
options.fn_template_options.add_specialization("^MaxValue$", ["int", "float"], add_
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
template<typename T> T MaxValue(
```

- Corresponding python decls (stub) -

copy ↗

```
# -----  
#      <template specializations for  
@overload  
def max_value(values: List[int]) ->  
    pass
```

```
@overload  
def max_value(values: List[float]) ->  
    pass  
#      </template specializations fo  
# -----
```

pybind11 C++ binding code

+

nanobind C++ binding code

+

## Export template functions with a suffix

Instead of using `@overload`, we can give different names to the python functions:

```
cpp_code = """  
    template<typename T> voi LogValue(const std::string& label, const T& value);  
"""  
  
options = litgen.LitgenOptions()  
options.fn_template_options.add_specialization("^LogValue$", ["int", "float"], add_  
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy ↗

```
template<typename T> void LogValue
```

- Corresponding python decls (stub)

copy ↗

```
# -----  
#     <template specializations for  
def log_value_int(label: str, value:  
    pass
```

```
def log_value_float(label: str, value:  
    pass  
#     </template specializations for  
# -----
```

pybind11 C++ binding code

nanobind C++ binding code

+

+

## Template classes

### Introduction

Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)

litgen handles template classes instantiation in a sophisticated way.

In the example below, we set the following options:

#### type replacements

We set an option for type name replacements, so that `ImGuiConfig` will be exposed as `Config` in python:

```
options.type_replacements.add_last_replacement(r"ImGui([A-Z][a-zA-Z0-9]*)", r"\
```

#### class specialization

- We tell litgen to instantiate `ImVector` for `ImGuiConfig`, `float *`, and `int32_t`.

[Skip to main content](#)

```

options.class_template_options.add_specialization(
    "ImVector", # which class do we want to specialize
    ["ImGuiConfig", "float *", "int32_t"], # for which types
    ["Int32=uint32_t"] # With which synonyms
)

```

## Notes

- The member `Foo::Configs` will be exposed with the correct python type (`ImGuiConfig`)
- The member `Foo::IntValues` will not be published, since `ImVector<int>` is not published
- litgen will emit a warning about the missing specialization for `int`*

## Example of template class instantiation

```

cpp_code = """
    struct ImGuiConfig { /* implementation not shown here */ };

    template<typename T>
    struct ImVector
    {
        // Implementation not shown here
        private:
            T* data;
    };

    struct Foo
    {
        ImVector<ImGuiConfig> Configs; // This member will be added to the bindings
        ImVector<int> IntValues;      // This member will be excluded from the bindings
    };
"""

options = litgen.LitgenOptions()
options.type_replacements.add_last_replacement(r"ImGui([A-Z][a-zA-Z0-9]*)", r"\1")
options.class_template_options.add_specialization(
    "ImVector", # which class do we want to specialize
    ["ImGuiConfig", "float *", "int32_t"], # for which types
    ["Int32=uint32_t"], # With which synonyms
)
litgen_demo.demo(options, cpp_code)

# Note: the warnings below are normal, since we did not specialize ImVector<int> (t

```

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

Warning: (Undefined) Excluding template type ImVector<int> because its specialization  
 While parsing a "type", corresponding to this C++ code:  
 Position:16:9

### Original C++ decls

copy ↗

```
struct ImGuiConfig { /* implementation */ }

template<typename T>
struct ImVector
{
    // Implementation not shown
private:
    T* data;
};

struct Foo
{
    ImVector<ImGuiConfig> Config;
    ImVector<int> IntValues;
};
```

### Corresponding python decls (stub)

copy ↗

```
class Config:
    # implementation not shown here
    def __init__(self) -> None:
        """Auto-generated default constructor

    # -----
    # <template specializations for
class ImVector_Config: # Python specialization
    # Implementation not shown here
    def __init__(self) -> None:
        """Auto-generated default constructor

class ImVector_float_ptr: # Python specialization
    # Implementation not shown here
    def __init__(self) -> None:
        """Auto-generated default constructor

class ImVector_int32_t: # Python specialization
    # Implementation not shown here
    def __init__(self) -> None:
        """Auto-generated default constructor
```

[Skip to main content](#)

```
#      </template specializations fo
# -----
+-----+
pybind11 C++ binding code
nanobind C++ binding code
```

## Suppress template class warnings

You can ask litgen to ignore the warnings concerning the missing specialization:

```
# tell litgen to ignore warnings that contain "Excluding template type ImVector<int>
options.srcmlcpp_options.ignored_warning_parts.append("Excluding template type ImVe
# the following line emits a warning that is ignored
generated_code = litgen.generate_code(options, cpp_code)
```

## Enums

*Relevant portion of the [pybind11 manual](#) and of the [nanobind manual](#)*

litgen handles `enum` and `enum class` enums.

## Classic C enums

See example below:

- litgen automatically remove the standard prefix from enum values: `Foo::Foo_a` is exposed as `Foo.a`
- comments about values are preserved in the stub
- litgen automatically handles the enum numbering and outputs the values as a comment in the stub
- if some macro values are used (like `MY_VALUE`), we can set their value
- Computed values are also correctly exposed (see `Foo.d`)

[Skip to main content](#)

```

cpp_code = """
// Doc about Foo
// On several lines
enum Foo
{
    Foo_a, // This is a

    // And this is b and c's comment
    Foo_b,
    Foo_c = MY_VALUE,

    Foo_d = Foo_a | Foo_b + Foo_c, // And a computed value

    Foo_e = 4,

    Foo_count, // And this is count: by default this member is suppressed
};

"""

import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
options.srcmlcpp_options.named_number_macros = {"MY_VALUE": 256}
# options.enum_flag_skip_count = False # Uncomment this to generate a definition for
litgen_demo.demo(options, cpp_code)

```

Original C++ decls

copy ↗

```

// Doc about Foo
// On several lines
enum Foo
{
    Foo_a, // This is a

    // And this is b and c's comment
    Foo_b,
    Foo_c = MY_VALUE,

    Foo_d = Foo_a | Foo_b + Foo_c, /.

    Foo_e = 4,

    Foo_count, // And this is count:
};

```

– Corresponding python decls (stub) –

copy ↗

```

class Foo(enum.IntEnum):
    """ Doc about Foo
    On several lines
    """
    a = enum.auto() # (= 0) # This .

    # And this is b and c's comment
    b = enum.auto() # (= 1)
    c = enum.auto() # (= 256)

    d = enum.auto() # (= Foo.a | Foo
    e = enum.auto() # (= 4)

```

pybind11 C++ binding code

+

[Skip to main content](#)

# C++ enums: enum class

enum class is also supported, see example below:

```
cpp_code = """
enum class Foo
{
    A,
    B,
    C = MY_VALUE,
    D = A | B + C,
    E = 4,
    F
};

options = litgen.LitgenOptions()
options.srcmlcpp_options.named_number_macros = {"MY_VALUE": 256}
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy 

```
enum class Foo
{
    A,
    B,
    C = MY_VALUE,
    D = A | B + C,
    E = 4,
    F
};
```

pybind11 C++ binding code

nanobind C++ binding code

- Corresponding python decls (stub) -

copy 

```
class Foo(enum.IntEnum):
    a = enum.auto() # (= 0)
    b = enum.auto() # (= 1)
    c = enum.auto() # (= 256)
    d = enum.auto() # (= A | B + C)
    e = enum.auto() # (= 4)
    f = enum.auto() # (= 5)
```

## Arithmetic enums

Use `options.enum_make_arithmetic_regex` to make the enum arithmetic in python, so that it can be converted to a number in python.

[Skip to main content](#)

```

cpp_code = """
    enum class Foo { A, B, C};
"""

options = litgen.LitgenOptions()
options.enum_make_arithmetic_regex = "^Foo$"
litgen_demo.demo(options, cpp_code, show_pydef=True)

```

*Original C++ decls*



```
enum class Foo { A, B, C};
```

*Corresponding python decls (stub)*



```

class Foo(enum.IntEnum):
    a = enum.auto() # (= 0)
    b = enum.auto() # (= 1)
    c = enum.auto() # (= 2)

```

*pybind11 C++ binding code*



```

auto pyEnumFoo =
    py::enum_<Foo>(m, "Foo", py::arithmetic(), "")
        .value("a", Foo::A, "")
        .value("b", Foo::B, "")
        .value("c", Foo::C, "");

```

*nanobind C++ binding code*



```

auto pyEnumFoo =
    nb::enum_<Foo>(m, "Foo", nb::is_arithmetic(), "")
        .value("a", Foo::A, "")
        .value("b", Foo::B, "")
        .value("c", Foo::C, "");

```

## Namespaces

It is possible to define a “root” namespace, for which no submodule is emitted. Thanks to the option below, the “Main” namespace will not be outputted as a submodule. However its content is exported.

[Skip to main content](#)

```
options.namespaces_root = ["Main"]
```

For other namespace, litgen will output them as python submodules. If a namespace occurs twice in the C++ code, its content will grouped in the Python stubs

*Note: in the python stub file, a namespace is shown as a fake class (although it is really a python submodule). This avoids the creation of an additional stub file, while maintaining a perfect code completion inside IDEs!*

In the example below:

- the namespace "details" is ignored
- any anonymous namespace is ignored
- occurrences of the namespace "Inner" are grouped
- namespaces names are converted to snake\_case

```

cpp_code = """

void FooRoot(); // A function in the root namespace

namespace details // This namespace should be excluded (see options.namespace_exclude)
{
    void FooDetails();
}

namespace // This anonymous namespace should be excluded
{
    void LocalFunction();
}

// This namespace should not be outputted as a submodule,
// since it is present in options.namespaces_root
namespace Main
{
    // this is an inner namespace (this comment should become the namespace documentation)
    namespace Inner
    {
        void FooInner();
    }

    // This is a second occurrence of the same inner namespace
    // The generated python module will merge these occurrences
    // (and this comment will be ignored, since the Inner namespace already has documentation)
    namespace Inner
    {
        void FooInner2();
    }
}
"""

import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
options.namespaces_root = ["Main"]
# options.namespace_exclude_regex = r"[Ii]nternal|[Dd]etail" # this is the default
options.python_run_black_formatter = True
# options.python_convert_to_snake_case = False # uncomment this in order to keep the original names
litgen_demo.demo(options, cpp_code)

```

Original C++ decls

copy ↗

```
void FooRoot(); // A function in the root namespace

namespace details // This namespace is present in options
{
    void FooDetails();
}

namespace // This anonymous name is present in options
{
    void LocalFunction();
}

// This namespace should not be present in options
// since it is present in options
namespace Main
{
    // this is an inner namespace
    namespace Inner
    {
        void FooInner();
    }

    // This is a second occurrence of the same name
    // The generated python module will have two methods
    // (and this comment will be present in the generated code)
    namespace Inner
    {
        void FooInner2();
    }
}
```

pybind11 C++ binding code

nanobind C++ binding code

- Corresponding python decls (stub)

copy ↗

```
def foo_root() -> None:
    """ A function in the root namespace
    pass

# <submodule inner>
class inner: # Proxy class that implements the interface
    pass # (This corresponds to a C++ class)
    """ this is an inner namespace (and this is present in options)
    @staticmethod
    def foo_inner() -> None:
        pass
    @staticmethod
    def foo_inner2() -> None:
        pass

# </submodule inner>
```

## Manually add custom bindings

Litgen normally generates bindings automatically from C++ headers, but sometimes you may want to extend the API with extra methods or functions that are not present in the C++ code.

`LitgenOptions.custom_bindings` lets you do this **without modifying your C++ headers**.

You can attach:

[Skip to main content](#)

- free functions to a **C++ namespace** (shown as a Python submodule),
- free functions to the **main module**.

Each injection consists of:

- **stub code** (Python declarations added to the generated `.pyi` file),
- **pydef code** (C++ binding code inserted into the generated binding .cpp file, using pybind11 or nanobind syntax).

Placeholders are available inside `pydef_code`:

Placeholder	Role	Expansion (example)
<code>LG_CLASS</code>	The current bound C++ class object	<code>pyNsRootNs_ClassFoo</code> (of type <code>nb::class_</code> or <code>py::class_</code> )
<code>LG_SUBMODULE</code>	The current submodule (for a namespace)	<code>pyNsRootNs</code> (a <code>nb::module_</code> or <code>py::module_</code> )
<code>LG_MODULE</code>	The main Python module object	<code>m</code> (the top-level module)

These placeholders are automatically replaced by litgen during C++ binding code generation. You can safely use them inside your `pydef_code` snippets without declaring them yourself.

Note about `pydef_code` syntax:

- You can use either **pybind11** or **nanobind** syntax in `pydef_code`, depending on which backend you are using.
- Since the `pydef_code` is inserted into a function, limit yourself to statements that are valid **inside a function**: no function/class definitions, no `#include`. Use **lambdas** for small helpers.
- When writing lambdas, **fully qualify C++ types** if the class/namespace isn't open (e.g. `const RootNs::Foo& self`).
- Argument helpers differ by backend (`py::arg` vs `nb::arg`). Use the one matching your active backend.

# Example C++ code

This page demonstrates how to add custom bindings to the following C++ code.

```
cpp_code = """  
namespace RootNs  
{  
    struct Foo  
    {  
        int mValue = 0;  
    };  
}  
"""
```

We will be adding custom bindings to the class `RootNs::Foo`, the namespace `RootNs`, and to the main module.

## Custom bindings for a class

`options.custom_bindings.add_custom_bindings_to_class(qualified_class, stub_code, pydef_code)`, lets us extend the generated Python bindings with extra methods, properties, or static methods.

### Args:

- `qualified_class`: Fully qualified C++ class name (e.g. `"RootNs::Foo"`).
- `stub_code`: Python stub declarations to be inserted into the generated stub (".`.pyi`") file. These should be written in normal Python syntax with type annotations.
- `pydef_code`: Custom binding code in C++ (pybind11/nanobind syntax). You can use the placeholder `LG_CLASS` to refer to the bound `py::class_` / `nb::class_` object.

```

import litgen
options = litgen.LitgenOptions()

options.custom_bindings.add_custom_bindings_to_class(
    qualified_class="RootNs::Foo",
    stub_code=''''
        def get_value(self) -> int:
            """Get the value"""

        ...
        def set_value(self, value: int) -> None:
            """Set the value"""

        ...
    ''',
    pydef_code=''''
        LG_CLASS.def("get_value", [])(const RootNs::Foo& self){ return self.mValue;
        LG_CLASS.def("set_value", [])(RootNs::Foo& self, int value){ self.mValue = v
    ''',
)

```

## Custom bindings for C++ namespace / Python submodule

`options.custom_bindings.add_custom_bindings_to_submodule(qualified_namespace, stub_code, pydef_code)`, lets us extend the generated Python bindings with extra functions.

### Args:

- `qualified_namespace`: Fully qualified C++ namespace name (e.g. "RootNs").
- `stub_code`: Python stub declarations to be inserted into the generated stub ("pyi") file. These should be written in normal Python syntax with type annotations. **Functions here should be decorated with `@staticmethod`**. Explanation for this: in stubs, namespaces are represented as proxy classes. Thus, functions must be declared as `@staticmethod` to indicate they are module-level, not instance methods.
- `pydef_code`: Custom binding code in C++ (pybind11/nanobind syntax). You can use the placeholder `LG_SUBMODULE` to refer to the bound submodule object.

### Namespace stubs appear as a proxy class.

When declaring functions in a C++ namespace (Python submodule) in the stub, mark them with `@staticmethod`. These are module-level functions, not instance methods.

```
options.custom_bindings.add_custom_bindings_to_submodule(
    qualified_namespace="RootNs",
    stub_code='''
@staticmethod # We **must** use @staticmethod here
def foo_namespace_function() -> int:
    """A custom function in the submodule"""
    ...
    ''',
    pydef_code='''
// Example of adding a custom function to the submodule
LG_SUBMODULE.def("foo_namespace_function", [](){ return 53; });
''',
)
```

## Custom bindings for the main module

`options.custom_bindings.add_custom_bindings_to_main_module(stub_code, pydef_code)`, lets us extend the generated Python bindings with extra functions in the main module.

### Args:

- `stub_code`: Python stub declarations to be inserted into the generated stub (".*.pyi*") file. These should be written in normal Python syntax with type annotations.
- `pydef_code`: Custom binding code in C++ (pybind11/nanobind syntax). You can use the placeholder `LG_MODULE` to refer to the bound module object.

```
options.custom_bindings.add_custom_bindings_to_main_module(
    stub_code='''
def global_function() -> int:
    """A custom function in the main module"""
    ...
    ''',
    pydef_code='''
// Example of adding a custom function to the main module
LG_MODULE.def("global_function", [](){ return 64; });
''',
)
```

# Generated code (with custom bindings)

We may now call `litgen.generate_code` to generate the bindings, which will include our custom additions. The generated code is shown below.

```
from litgen.demo import litgen_demo  
litgen_demo.demo(options, cpp_code)
```

Original C++ decls

copy 

```
namespace RootNs  
{  
    struct Foo  
    {  
        int mValue = 0;  
    };  
}
```

- Corresponding python decls (stub) -

copy 

```
# <submodule root_ns>  
class root_ns: # Proxy class that i  
    pass # (This corresponds to a C  
class Foo:  
    m_value: int = 0  
    def __init__(self, m_value:  
        """Auto-generated default  
        pass  
  
    def get_value(self) -> int:  
        """Get the value""""  
        ...  
    def set_value(self, value: int):  
        """Set the value""""  
        ...  
  
    @staticmethod # We **must** use  
    def foo_namespace_function() ->  
        """A custom function in the :  
        ...  
# </submodule root_ns>  
  
def global_function() -> int:  
    """A custom function in the main  
    ...
```

pybind11 C++ binding code

+

nanobind C++ binding code

+

[Skip to main content](#)

## Note about ordering:

If you call `add_custom_code_to_*` multiple times for the same target (class/namespace/module), snippets are emitted **in the order they were added**.

# Preprocessor and macros

## Export macro values into the python module

Some `#define` macros can be exported:

Simple preprocessor defines can be exported as global variables, e.g.:

```
#define MYLIB_VALUE 1
#define MYLIB_FLOAT 1.5
#define MYLIB_STRING "abc"
#define MYLIB_HEX_VALUE 0x00010009
```

This is limited to *simple* defines.

You can also apply a simple renaming to the macro value: see example below.

```
cpp_code = """
#define MYLIB_VALUE 1
#define MYLIB_FLOAT 1.5
#define MYLIB_STRING "abc"
#define MYLIB_HEX_VALUE 0x00010009
"""

import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
options.macro_define_include_by_name_regex = "^MYLIB_"
# Suppress the "MYLIB_" prefix:
options.macro_name_replacements.add_first_replacement(r"^MYLIB_( [A-Z_]*)", r"\1")
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy ↗

```
#define MYLIB_VALUE 1
#define MYLIB_FLOAT 1.5
#define MYLIB_STRING "abc"
#define MYLIB_HEX_VALUE 0x00010009
```

- Corresponding python decls (stub)

copy ↗

```
VALUE = 1
FLOAT = 1.5
STRING = "abc"
HEX_VALUE = 0x00010009
```

pybind11 C++ binding code

copy ↗

```
m.attr("VALUE") = 1;
m.attr("FLOAT") = 1.5;
m.attr("STRING") = "abc";
m.attr("HEX_VALUE") = 0x00010009;
```

nanobind C++ binding code

copy ↗

```
m.attr("VALUE") = 1;
m.attr("FLOAT") = 1.5;
m.attr("STRING") = "abc";
m.attr("HEX_VALUE") = 0x00010009;
```

## Set numeric macro values

Sometimes, it is necessary to tell litgen the value of certain numeric macros. In the example below, the member `values` can be exposed as a numpy array, but litgen needs to know its size.

We set it via the option:

```
options.srcmlcpp_options.named_number_macros["MY_COUNT"] = 256
```

```

cpp_code = """
#define MY_COUNT 256

struct Foo
{
    int values[MY_COUNT];
};

"""

options = litgen.LitgenOptions()
options.srcmlcpp_options.named_number_macros["MY_COUNT"] = 256

litgen_demo.demo(options, cpp_code)

```

*Original C++ decls*

copy ↗

```

#define MY_COUNT 256

struct Foo
{
    int values[MY_COUNT];
};

```

*Corresponding python decls (stub)*

copy ↗

```

class Foo:
    values: np.ndarray # ndarray[ty]
    def __init__(self) -> None:
        """Auto-generated default constructor
        pass

```

*pybind11 C++ binding code*

*nanobind C++ binding code*

## Ignore litgen warnings

When processing code, litgen may emit some warnings when it encounters code that it cannot process. If possible, you can change the options in order to handle those warnings.

## Example code with potential warnings

However, there may be some warnings that you want to ignore. See the example below, where we use the code from a file name [10\\_05\\_10\\_sample\\_code.h](#).

Let's load it and display its content:

[Skip to main content](#)

```
import litgen
from litgen.demo import litgen_demo

with open("10_05_10_sample_code.h") as f:
    cpp_code = f.read()

litgen_demo.show_cpp_code(cpp_code)
```

copy ↗

```
// This will trigger a warning:
// Ignoring template function. You might need to set LitgenOptions.fn_template
// While parsing a "function_decl", corresponding to this C++ code:
// Position:4:1
//         template<typename T> void MyOperation(T value);
template<typename T> void MyOperation(T value);

struct Foo {};

// This will generate a warning:
// operators are supported only when implemented as a member functions
// And this operator will not be exported
bool operator==(const Foo& v1, const Foo& v2);

// litgen does not support C-style function parameters
// This function will trigger a warning: "Can't use a function_decl as a param"
int CallOperation(int (*functionPtr)(int, int), int a, int b) {
    return (*functionPtr)(a, b);
}
```

## Warnings emitted by litgen

Now, let's process it with litgen, and see that we encounter many warnings:

```
options = litgen.LitgenOptions()
generated_code = litgen.generate_code_for_file(options, "10_05_10_sample_code.h")
```

```

Warning: (SrcmlcppIgnoreElement) A cpp element of type "function" was stored as Cpp
Warning: (SrcmlcppIgnoreElement) Can't use a function_decl as a param.
While parsing a "function_decl", corresponding to this C++ code:
10_05_10_sample_code.h:21:19
    int CallOperation(int (*functionPtr)(int, int), int a, int b) {
        ^
Warning: (LitgenBlockElementException) Warning: (LitgenBlockElementException) operat
While parsing a "function_decl", corresponding to this C++ code:
10_05_10_sample_code.h:16:1
    bool operator==(const Foo& v1, const Foo& v2);
    ^
Warning: (LitgenTemplateFunctionIgnore) Ignoring template function MyOperation. You
While parsing a "function_decl", corresponding to this C++ code:
10_05_10_sample_code.h:7:1
    template<typename T> void MyOperation(T value);
    ^
Warning: (LitgenTemplateFunctionIgnore) Ignoring template function MyOperation. You
While parsing a "function_decl", corresponding to this C++ code:
10_05_10_sample_code.h:7:1
    template<typename T> void MyOperation(T value);
    ^

```

## Suppress the warnings

Once you studied the warnings, and once you know they are harmless, you can suppress them by filling `options.srcmlcpp_options.ignored_warning_parts`.

```

options = litgen.LitgenOptions()
options.srcmlcpp_options.ignored_warning_parts.append("Can't use a function_decl as")
options.srcmlcpp_options.ignored_warning_parts.append(
    "operators are supported only when implemented as a member functions")
options.srcmlcpp_options.ignored_warning_parts.append("Ignoring template function M")
# The next line does not emit warnings anymore
generated_code = litgen.generate_code_for_file(options, "10_05_10_sample_code.h")

# The generated code will include only what was correctly converted
litgen_demo.show_python_code(generated_code.stub_code)

```

copy ↗

```
##### <generated_from:10_05_10_sample_code.h> #####
class Foo:
    def __init__(self) -> None:
        """Auto-generated default constructor"""
        pass

##### </generated_from:10_05_10_sample_code.h> #####
#####
```

## Postprocessing and preprocessing

### Header preprocessing

If you need to preprocess header code before the generation, you can create a function that transforms the source code, and store it inside

```
options.srcmlcpp_options.code_preprocess_function
```

For example:

```
def preprocess_change_int(code: str) -> str:
    return code.replace("int", "Int32") # This is a *very* dumb preprocessor

cpp_code = """
int add(int, int b);
"""

import litgen
from litgen.demo import litgen_demo

options = litgen.LitgenOptions()
options.srcmlcpp_options.code_preprocess_function = preprocess_change_int
litgen_demo.demo(options, cpp_code)
```

[Skip to main content](#)

*Original C++ decls*

copy ↗

```
int add(int, int b);
```

*Corresponding python decls (stub)*

copy ↗

```
def add(param_0: Int32, b: Int32) ->  
    pass
```

*pybind11 C++ binding code*

*nanobind C++ binding code*

+

+

## Post-processing of the stub and pydef files

You can also post-process the stub and pydef files, i.e. apply a function that will be called after the code is generated. Below is a very dumb example:

```
cpp_code = """
int AnswerToTheUltimateQuestionOfLife_TheUniverse_AndEverything() { return 42; }
"""

from codemanip import code_utils

def postprocess_stub(code: str) -> str:
    return (
        code_utils.unindent_code(
            """
# Copyright(c) 2023 – Pascal Thomet
#     Yes, I claim the copyright on this magnificent function.
#     ...At least, I tried...
"""
        )
        + code
    )

def postprocess_pydef(code: str) -> str:
    return (
        code_utils.unindent_code(
            """
// Copyright(c) 2023 – Pascal Thomet
//     Yes, I claim the copyright on this magnificent function.
//     ...At least, I tried...
"""
        )
        + code
    )

options = litgen.LitgenOptions()
options.postprocess_stub_function = postprocess_stub
options.postprocess_pydef_function = postprocess_pydef
litgen_demo.demo(options, cpp_code, show_pydef=True)
```

Original C++ decls

copy 

```
int AnswerToTheUltimateQuestionOfLife();
```

- Corresponding python decls (stub) -

copy 

```
# Copyright(c) 2023 – Pascal Thomet
# Yes, I claim the copyright on this magnificent function.
# ...At least, I tried...
```

```
def answer_to_the_ultimate_question_of_life():
    pass
```

pybind11 C++ binding code

copy 

```
// Copyright(c) 2023 – Pascal Thomet
// Yes, I claim the copyright on this magnificent function.
// ...At least, I tried...
m.def("answer_to_the_ultimate_question_of_life_the_universe_and_everything",
      AnswerToTheUltimateQuestionOfLife_TheUniverse_AndEverything);
```

nanobind C++ binding code

copy 

```
// Copyright(c) 2023 – Pascal Thomet
// Yes, I claim the copyright on this magnificent function.
// ...At least, I tried...
m.def("answer_to_the_ultimate_question_of_life_the_universe_and_everything",
      AnswerToTheUltimateQuestionOfLife_TheUniverse_AndEverything);
```

## srcML - C++ parsing advices

srcML parses C++ 14 code into XML:

- it will *never* fail to construct an XML tree from C++ code
- it will *always* regenerate the exact same original C++ code from the XML tree

However, there are corner cases where the XML tree is not what you would expect. See some gotchas below.

# srcML

Don't use `={} as function's default parameter value`

See [related issue](#)

```
void Foo(int v = {} );
```

is parsed as a declaration statement, whereas

```
void Foo(int v = 0 );
```

is correctly parsed as a function declaration.

## Note about mixing auto return and API markers

Mixing API marker and auto return type is not supported.

Such functions will not be parsed correctly!

```
MY_API auto my_modulo(int a, int b)
MY_API auto my_pow(double a, double b) -> double
```

# Python bindings

## Note about function arrow-return type notation:

Arrow return notation function are correctly exported to python, including their return type. For example,

[Skip to main content](#)

Will result in:

```
def my_pow(a: float, b: float) -> float:  
pass
```

## Note about function inferred return type notation:

Functions with an inferred return type are correctly exported to python, however the published return type is unknown and will be marked as "Any" For example,

```
auto my_pow(double a, double b)
```

Will result in:

```
def my_pow(a: float, b: float) -> Any:  
pass
```

# srcmlcpp: C++ code parsing

litgen provides three separate python packages, srcmlcpp is one of them:

- [codemanip](#): a python package to perform *textual* manipulations on C++ and Python code. See [code\\_utils.py](#)
- [srcmlcpp](#): a python package that build on top of srcML in order to interpret the XML tree produced by srcML as a tree of python object resembling a C++ AST.
- [Litgen](#): a python package that generates python bindings from C++ code.

[srcmlcpp](#) will transform C++ source into a tree of Python objects (descendants of [CppElement](#)) that reflect the C++ AST.

This tree is used by litgen to generate the python bindings. It may also be used to perform automatic C++ code transformations.

# Transform C++ code into a CppElement tree

Given the C++ code below:

```
code = """
// A Demo struct
struct Foo
{
    const int answer(int *v=nullptr); // Returns the answer
};
"""
"""
```

srcmlcpp can produce a tree of `CppClass` with this call:

```
import srcmlcpp

options = srcmlcpp.SrcmlcppOptions()
cpp_unit = srcmlcpp.code_to_cpp_unit(options, code)
```

`cpp_unit` is then a tree of Python object (descendants of `CppClass`) that represents the source code.

Here is what it looks like under a debugger:

```
✓ └─ cpp_unit = {CppUnit} \n// A Demo struct\nstruct Foo\n{\npublic: // <default_access_type>\rThe root node is always a CppUnit\n    \n    > └─ block_children = {[list: 3] [CppEmptyLine(), CppStruct(cpp_element_comments=CppElementComments(comment_on_previous_lines='A Demo struct', comme...}\n        > └─ 0 = {CppEmptyLine}\n        \n        ✓ 1 = {CppClassStruct} // A Demo struct\n        \n        > └─ block_children = {[list: 1] [CppClassPublicProtectedPrivate(block_children=[CppClassFunctionDecl(cpp_element_comments=CppElementComments(comment_on_...}\n            > └─ 0 = {CppClassPublicProtectedPrivate} public:// <default_access_type>\n                \n                > └─ access_type = {str} 'public'\n                \n                > └─ block_children = {[list: 1] [CppClassFunctionDecl(cpp_element_comments=CppElementComments(comment_on_previous_lines='', comment_end_of_line='...}\n                    > └─ 0 = {CppClassFunctionDecl} const int answer(int * v = nullptr); // Returns\n                        > └─ cpp_element_comments = {CppElementComments} CppElementComments(comment_on_previous_lines='', comment_end_of_line='Returns\n                            > └─ is_auto_decl = {bool} False\n                            > └─ name = {str} 'answer'\n                    \n                    > └─ parameter_list = {CppClassParameterList} int * v = nullptr\n                    \n                    > └─ parameters = {[list: 1] [CppClassParameter(decl=CppDecl(cpp_el...}\n                        > └─ 0 = {CppClassParameter} int * v = nullptr\n                            \n                            > └─ decl = {CppClassDecl} int * v = nullptr\n                                > └─ cpp_element_comments = {CppElementComments} CppElementComments(comment_on_previous_lines='', comment_end_of_line='...}\n                                > └─ cpp_type = {CppClassType} int *\n                                    > └─ init = {str} 'nullptr'\n                                    > └─ name = {str} 'v'\n                                    > └─ range = {str} ''\n                                \n                                > └─ srcml_element = {Element: 3} <Element '{http://www...}\nEach `srcml_element` is an xml tree produced by srcML, that enables to reproduce an exact copy of the original source code of the element\n                                > └─ srcml_element = {Element: 1} <Element '{http://www...}\n                                    > └─ template_name = {str} ''\n                                    > └─ template_type = {NoneType} None\n                                    > └─ __len__ = {int} 1\n                            \n                            > └─ srcml_element = {Element: 1} <Element '{http://www...}\n                                > └─ specifiers = {[list: 0] []}\n                            \n                            > └─ srcml_element = {Element: 3} <Element '{http://www...}\n                                > └─ template = {NoneType} None\n                                > └─ type = {CppClassType} const int\n                                > └─ __len__ = {int} 1\n                            \n                            > └─ srcml_element = {Element: 2} <Element '{http://www...}\n                                > └─ type = {str} 'default'\n                                > └─ __len__ = {int} 1\n                            \n                            > └─ srcml_element = {Element: 1} <Element '{http://www...}\n                                > └─ cpp_element_comments = {CppClassComments} CppElementComments(comment_on_previous_lines='A Demo struct', comment_end_of_line='')\n                                    > └─ comment_end_of_line = {str} ''\n                                    > └─ comment_on_previous_lines = {str} 'A Demo struct'\n                                    > └─ name = {str} 'Foo'\n                            \n                            > └─ srcml_element = {Element: 2} <Element '{http://www...}\n                                > └─ super_list = {NoneType} None\n                                > └─ template = {NoneType} None\n                            \n                            > └─ 2 = {CppEmptyLine}\nComments on top or end of line comments are part of the parsed tree, so they can be used in generated code or in generated documentation\nEmpty lines are also part of the parsed tree, so they can be used in generated documentation or code
```

## Transform a CppElement tree into C++ code

### Transformation to source code from a tree of CppTypelement

CppTypelement provides a method `str_code()` that can output the C++ code it contains. It is close to the original source code (including comments), but can differ a bit.

#### Note

Any modification applied to the AST tree by modifying the CppElements objects (CppMethod, CppConstruct, etc.) will be visible using this method.

[Skip to main content](#)

```
from litgen.demo import litgen_demo  
litgen_demo.show_cpp_code(cpp_unit.str_code())
```

copy ↗

```
// A Demo struct  
struct Foo  
{  
public: // <default_access_type>  
    const int answer(int *v = nullptr); // Returns the answer  
};
```

## “Verbatim” transformation from tree to code

You can obtain the verbatim source code (i.e. the exact same source code that generated the tree), with a call to `str_code_verbatim()`.

### Note

- This will call the srcML executable using the srcml xml tree stored inside `cpp_unit.srcml_xml`, which guarantees to return the same source code
- Any modification applied to the AST tree by modifying the `CppClassElement` python objects (CppUnit, CppStruct, etc.) will not be visible using this method

```
print(cpp_unit.str_code_verbatim())
```

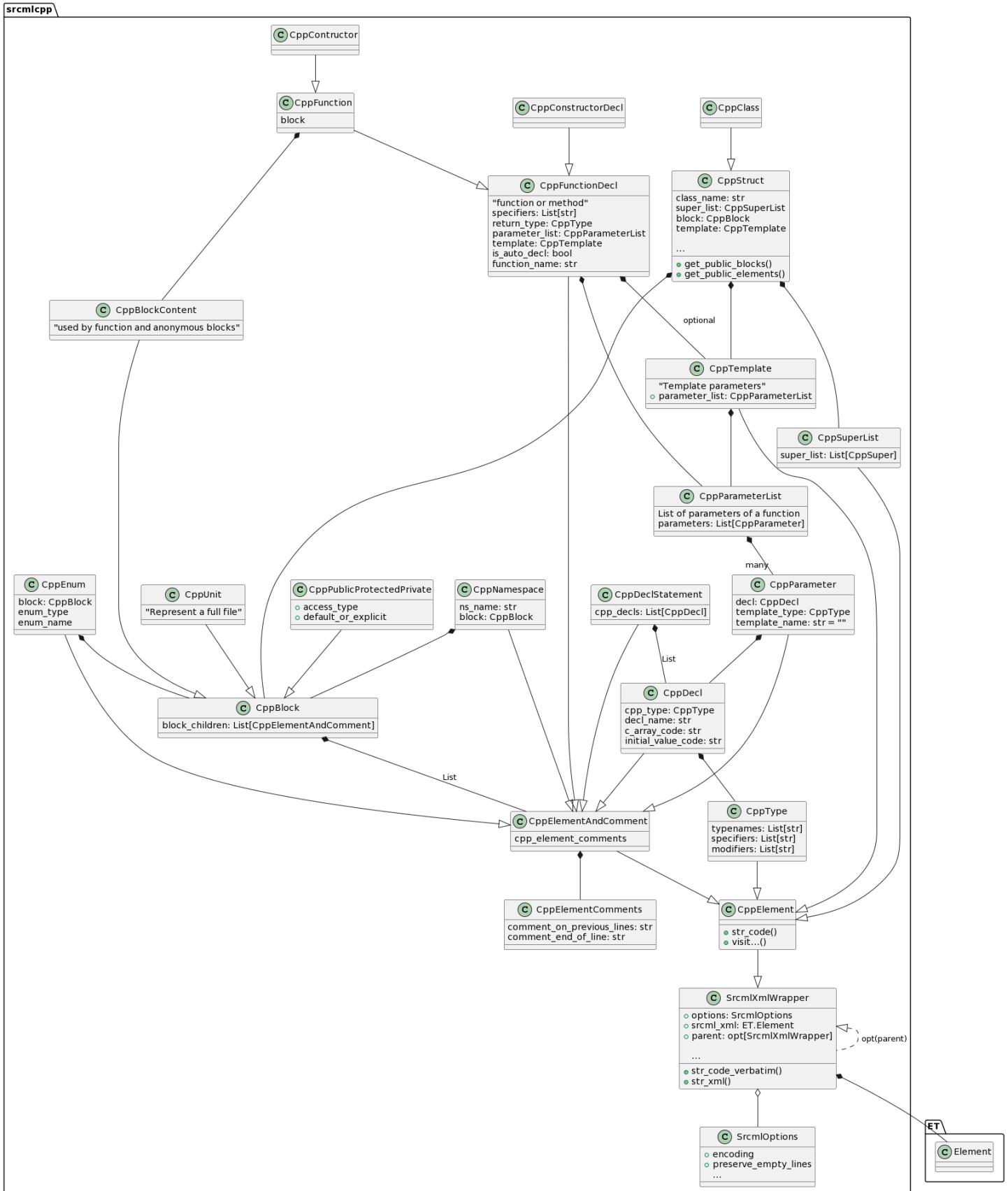
```
// A Demo struct  
struct Foo  
{  
    const int answer(int *v=nullptr); // Returns the answer  
};
```

[Skip to main content](#)

# CppClass types

When parsing C++ code, it will be represented by many python objects, that represents different C++ tokens.

See the diagram below for more information:

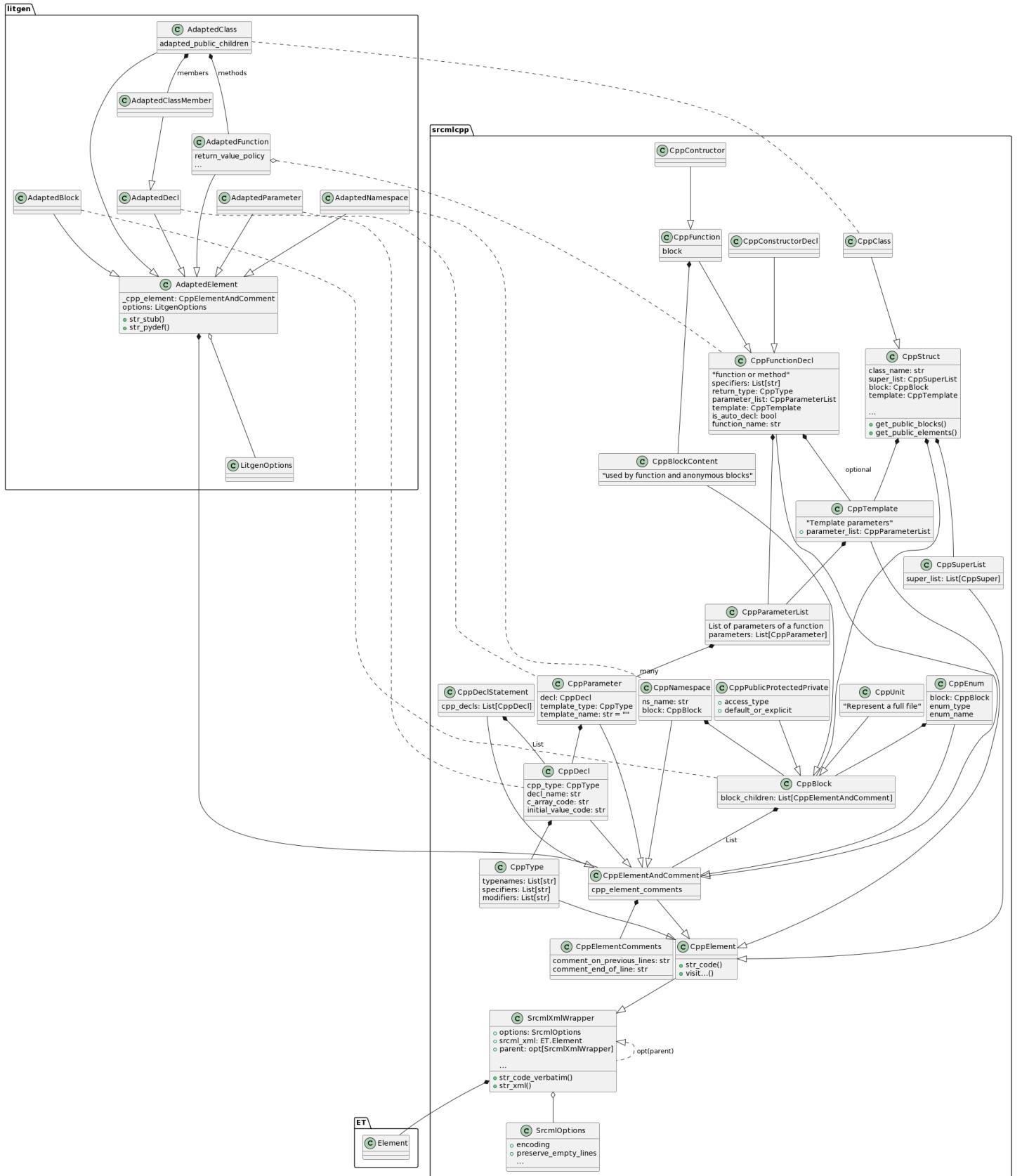


[Skip to main content](#)

# litgen and srcmlcpp

For information, when litgen transform C++ code into python bindings, it will transform the `CppClass` tree into a tree of `AdaptedElement`.

See diagram below:



[Skip to main content](#)

# Simple C++ code transformations

## Visiting the CppElement tree

Let's transform the following code into a tree of `CppClass`:

```
code = """
int AddNumbers(const MyModule_Class& c);
"""

import srcmlcpp

options = srcmlcpp.SrcmlcppOptions()
cpp_unit = srcmlcpp.code_to_cpp_unit(options, code)
```

We can then "visit" this tree, and for example log the type of all encountered elements:

```
def visitor_log_info(cpp_element: srcmlcpp.CppElement, event: srcmlcpp.CppElementsV
    if event == srcmlcpp.CppElementsVisitorEvent.OnElement:
        print(" " * depth + cpp_element.short_cpp_element_info())

    cpp_unit.visit_cpp_breadth_first(visitor_log_info)
```

```
CppClass
    CppEmptyLine
    CppFunctionDecl name=AddNumbers
        CppType name=int
        CppParameterList
            CppParameter
                CppDecl name=c
                    CppType name=MyModule_Class
    CppEmptyLine
```

## Transforming the CppElement tree

Let's suppose we want to apply a mass source code transformation where:

[Skip to main content](#)

- all functions names should be transformed to "snake\_case" (with a warning on top)
- all types whose name start with `MyModule_` should be replaced by `MyModule::` (i.e. we want to add a namespace)

In our example,

```
int AddNumbers(const MyModule_Class& c);
```

should be transformed to

```
// Was changed to snake_case!
int add_numbers(const MyModule::Class& c);
```

We can achieve this by defining `my_refactor_visitor` this way:

```
from srcmlcpp import CppFunctionDecl, CppType # import the types we want to apply
from codemanip import code_utils # for to_snake_case

def change_function_to_snake_case(cpp_function: CppFunctionDecl):
    """Change a function name to snake_case"""
    cpp_function.function_name = code_utils.to_snake_case(cpp_function.function_name)
    cpp_function.cpp_element_comments.comment_on_previous_lines += "Was changed to"

def make_my_module_namespace(cpp_type: CppType):
    """If a type starts with MyModule_, replace it by MyModule::"""

    def change_typename(typename: str):
        if typename.startswith("MyModule_"):
            return typename.replace("MyModule_", "MyModule::")
        else:
            return typename

    cpp_type.typenames = [change_typename(typename) for typename in cpp_type.typename]

def my_refactor_visitor(cpp_element: srcmlcpp.CppElement, event: srcmlcpp.CppElementEvent):
    """Our visitor will apply the transformation"""
    if event == srcmlcpp.CppElementsVisitorEvent.OnElement:
        if isinstance(cpp_element, CppFunctionDecl):
            change_function_to_snake_case(cpp_element)
        elif isinstance(cpp_element, CppType):
            make_my_module_namespace(cpp_element)
```

[Skip to main content](#)

Let's run this visitor and see its output:

```
# Let's visit the code
cpp_unit.visit_cpp_breadth_first(my_refactor_visitor)
# And print the refactored code
from litgen.demo import litgen_demo

litgen_demo.show_cpp_code(cpp_unit.str_code())
```

copy ↗

```
//Was changed to snake case!
int add_numbers(const MyModule::Class & c);
```

Note: `str_code_verbatim` still retains the original source code

```
litgen_demo.show_cpp_code(cpp_unit.str_code_verbatim())
```

copy ↗

```
int AddNumbers(const MyModule_Class& c);
```

## Optional: install srcML command line tool

[srcML](#) can also be used as a command line tool to generate XML representations of source code. It is used by litgen to generate the bindings.

You do not need to install srcML if you are using `litgen`, but it might be useful to have it installed to inspect the generated XML files.

## Either install srcML from pre-compiled binaries

You can download a pre-compiled version at [srcML.org](#)

[Skip to main content](#)

For example, on ubuntu 20.04:

```
wget http://131.123.42.38/lmcrs/v1.0.0/srcml_1.0.0-1_ubuntu20.04.deb  
sudo dpkg -i srcml_1.0.0-1_ubuntu20.04.deb
```

## Or build srcML from source

*Note: the build instructions in srcML repository are a bit out of date, which is why these instructions are provided here. It uses a fork of srcML that fixes some compilation issues on the develop branch*

### Install required packages

On ubuntu:

```
sudo apt-get install libarchive-dev antlr libxml2-dev libxslt-dev libcurl4-openssl-
```

On macOS:

```
brew install antlr2 boost
```

### Clone, build and install srcML

```
git clone https://github.com/pthom/srcML.git -b develop_fix_build  
mkdir -p build && cd build  
cmake ../srcML && make -j  
sudo make install
```