Python's asyncio in C++ for Terminal Graphics

2024/10/10 by Fernando Jose @ Meeting C++ 2024

Welcome!

- This talk is about a C++20 coroutines use case: using asyncio in C++.
- asyncio is Python's "library to write concurrent code with async/await syntax".
- The motivation was developing terminal visualizations for fun - I like programming puzzles, which I usually write in C++.

Disclaimer 🚨

I had no experience neither with C++20 coroutines nor async/await.

- There probably are low-hanging fruits in my solution.
- In fact, I'll discuss an open problem I've. Discussions and help are welcomed!

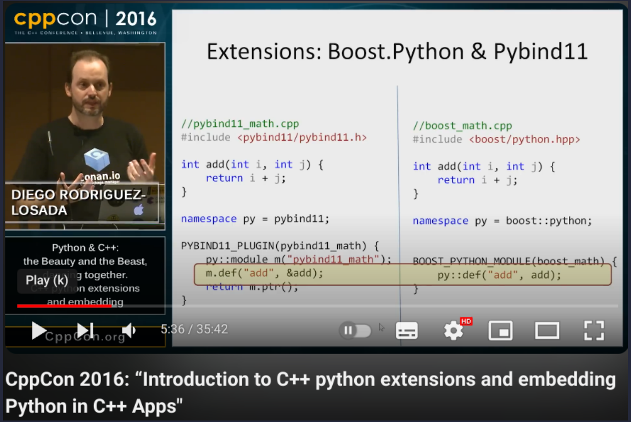
It's been a recreational project to learn how to use them, while addressing a personal "tooling need".

My goal is to share what I've managed to put together to get it working.

Outline

1. Using Python from C++ via pybind11.
2. The batgrl terminal graphics library.
3. Introduction to C++ & Python async interoperability. 3D Cubes.
4. Achieving `co_await` of `async def`s. Rock, Papers, Scissors.
5. Recapitulation.

Other Talks on C++ & Python Interoperability



Extensions: Boost.Python & Pybind11

```cpp
//pybind11_math.cpp
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

namespace py = pybind11;

PYBIND11_PLUGIN(pybind11_math) {
    py::module m("pybind11_math");
    m.def("add", &add);
    return m.ptr();
}
```

```cpp
//boost_math.cpp
#include <boost/python.hpp>

int add(int i, int j) {
    return i + j;
}

namespace py = boost::python;

BOOST_PYTHON_MODULE(boost_math) {
    py::def("add", add);
}
```

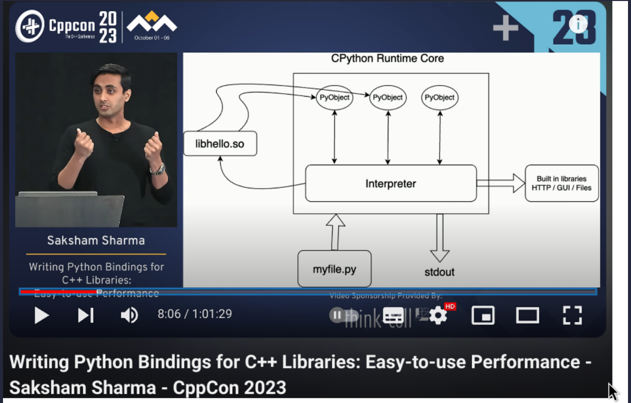
CppCon 2016: “Introduction to C++ python extensions and embedding Python in C++ Apps"

Other Talks on C++ & Python Interoperability

CppCon 2017: Jan Plhak “Pybind11 - Python on Steroids”

Important! In today's talk the focus is on using Python from C++.

Other Talks on C++ & Python Interoperability



Writing Python Bindings for C++ Libraries: Easy-to-use Performance - Saksham Sharma - CppCon 2023

Important! In today's talk the focus is on using Python from C++.

Using Python Code from C++ via pybind11

- Embedding the interpreter (https://pybind11.readthedocs.io/en/stable/advanced/embedding.html)

```cpp
#include <pybind11/embed.h>
```

- Start the interpreter using pybind11's RAII class `scoped_interpreter`.

```cpp
namespace py = pybind11;
py::scoped_interpreter guard{};
```

Using a Python Library from C++ via pybind11

- Modules are imported with `module::import` as pybind11's `object`s, for examples

```
py::object faker = py::module::import("faker");
py::object plt   = py::module::import("matplotlib.pyplot");
py::object np    = py::module::import("numpy");
```

- Objects' attributes and methods are accesible via `attr` (string-based interface).

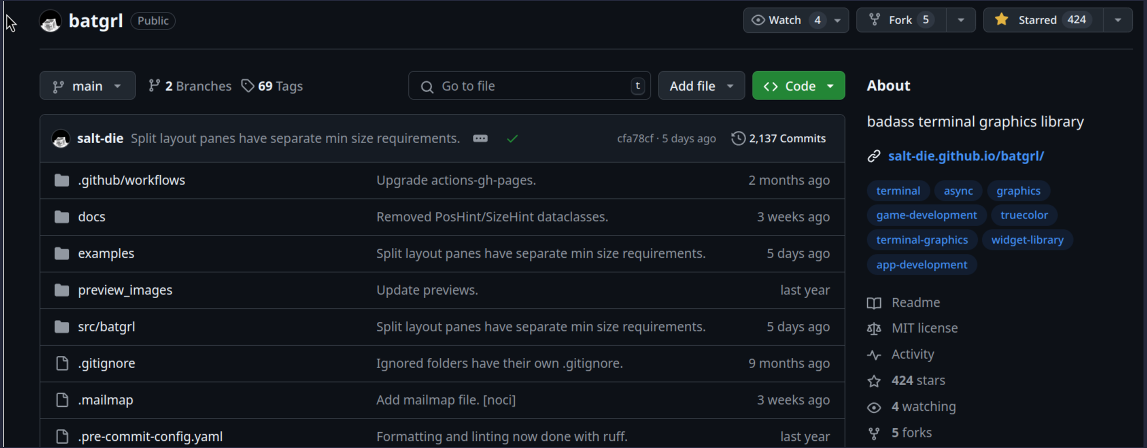
Example

```cpp
#include <pybind11/embed.h>

namespace py = pybind11;

int main() {
  py::scoped_interpreter guard{};
  py::object requests = py::module::import("requests");
  py::object response = requests.attr("get")("https://meetingcpp.com/2024/");
  py::print(response.attr("text"));
}
```

[finished with error]

```
terminate called after throwing an instance of 'pybind11::error_already_set'
  what():  ModuleNotFoundError: No module named 'requests'
```

The batgrl Terminal Graphics Library



salt-die Split layout panes have separate min size requirements. cfa78cf · 5 days ago 2,137 Commits

.github/workflows	Upgrade actions-gh-pages.	2 months ago
docs	Removed PosHint/SizeHint dataclasses.	3 weeks ago
examples	Split layout panes have separate min size requirements.	5 days ago
preview_images	Update previews.	last year
src/batgrl	Split layout panes have separate min size requirements.	5 days ago
.gitignore	Ignored folders have their own .gitignore.	9 months ago
.mailmap	Add mailmap file. [noci]	3 weeks ago
.pre-commit-config.yaml	Formatting and linting now done with ruff.	last year

About

badass terminal graphics library

salt-die.github.io/batgrl/

github.com/salt-die/batgrl. ~30k LoC. Based on asyncio.

Implementing a Python Library Interface in C++

Through the next slides

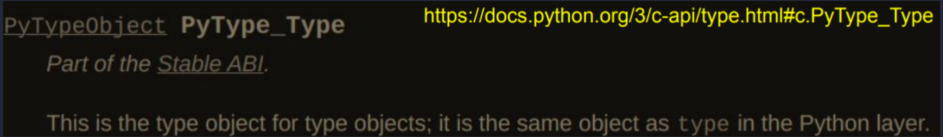
- a Python class is derived in C++
- to implement an abstract method,
- and make it compatible with `asyncio`.

The Python library shall then call this C++ code.

Inheritance Interoperation - Code

```cpp
py::object app_class =
    py::module::import("batgrl.app").attr("App");

py::object metaclass = py::reinterpret_borrow<py::object>(
    (PyObject *) &PyType_Type);

module.attr("AnApp") =
    metaclass("AnApp", py::make_tuple(app_class), members);

// Later a specific "members" instance is shown;
// it's a dictionary with the class' attributes and/or methods.
```

Inheritance Interoperation - Doc

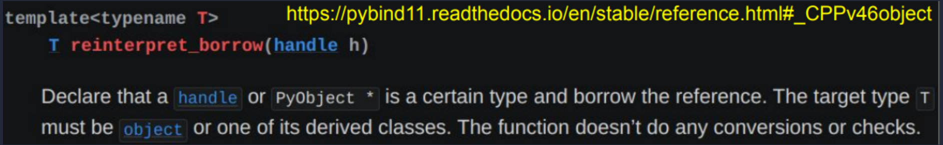


https://docs.python.org/3/c-api/type.html#c.PyType_Type

`PyTypeObject` **PyType_Type**

Part of the Stable ABI.

This is the type object for type objects; it is the same object as `type` in the Python layer.



https://pybind11.readthedocs.io/en/stable/reference.html#_CPPv46object

```
template<typename T>
    T reinterpret_borrow(handle h)
```

Declare that a `handle` or `PyObject *` is a certain type and borrow the reference. The target type `T` must be `object` or one of its derived classes. The function doesn't do any conversions or checks.

Thanks to Wenzel Jakob's explanation in https://github.com/pybind/pybind11/issues/1193#issuecomment-429451094.

Building a Sequence Diagram of a C++ batgrl App

```cpp
using namespace pybind11::literals;

int main() {
  py::scoped_interpreter guard{};
  py::object an_app_class =
      py::module_::import("embedded_module").attr("AnApp");
  py::object an_app = an_app_class("render_interval"_a=0.1)
  an_app.attr("run")();
}
```

`src/batgrl/app.py` model:

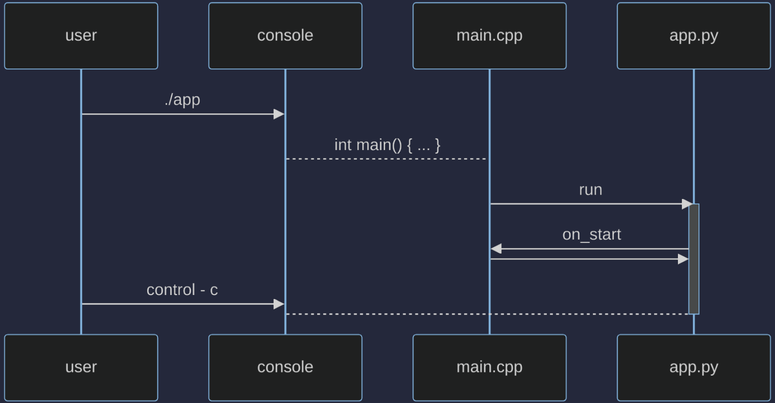
```python
def run(self):
  # ...
  asyncio.run(self._run_async())
```

Building a Sequence Diagram of a C++ batgrl App

`src/batgrl/app.py` model:

```python
def run(self):
  # ...
  asyncio.run(self._run_async())

async def _run_async(self):
  # ...
  async def auto_render():
    while True:
      # render ...
      await asyncio.sleep(self.render_interval)

  with # ...
    await asyncio.gather(self.on_start(), auto_render())
```

Sequence Diagram of a C++ batgrl App



Runtime + Source View

Awaitable for asyncio

```
class awaitable {
    std::future<void> future;

  public:
    awaitable(std::future<void>&& f) : future(std::move(f)) {}

    awaitable* await() { return this; }

    void next() {
        future.wait();
        throw py::stop_iteration{};
    }
};
```

Good for `async def` without suspension (including apps with `add_done_callback`).

Enable Async

```cpp
py::class_<awaitable> enable_async(py::module m) {
  return py::class_<awaitable>(m, "awaitable")
    .def(py::init<>())
    .def("__await__", &awaitable::await)
    .def("__next__",  &awaitable::next);
}
```

The `awaitable` is the return type used to define an `async def` in C++ that is called from Python using the binding.

Refined from https://github.com/pybind/pybind11/pull/2663 - "add async_class and async_class::def_async" (closed)

https://peps.python.org/pep-0492/

PYBIND11_EMBEDDED_MODULE

```cpp
PYBIND11_EMBEDDED_MODULE(cubes, module) {
  enable_async(module);

  py::dict members;
  // members["on_start"] = ... detailed from next slide

  py::object app_class =
      py::module::import("batgrl.app").attr("App");
  py::object metaclass = py::reinterpret_borrow<py::object>(
      (PyObject *) &PyType_Type);
  module.attr("CubesApp") =
      metaclass("CubesApp", py::make_tuple(app_class), members);
}
```

C++ code inside PYBIND11_EMBEDDED_MODULE becomes available from Python.

3D Cubes - main

```cpp
using namespace pybind11::literals;

PYBIND11_EMBEDDED_MODULE(cubes, module) {
  py::dict members;
  members["on_start"] = py::cpp_function(
    [](py::object self) -> awaitable {
      py::object renderer_module =
          py::module::import("cube_renderer");
      py::object renderer_class =
          rendered_module.attr("CubeRenderer");
      py::object renderer =
          renderer_class("size_hint"_a=py::make_tuple(1., 1.));
      // continues on the next slide ...
    },
    py::is_method(py::none()));
}
```

3D Cubes - main

```cpp
PYBIND11_EMBEDDED_MODULE(cubes, module) {
  members["on_start"] = py::cpp_function(
    [](py::object self) -> awaitable {
      // ... continuing from the previous slide
      renderer.attr("cubes") = get_cubes();
      self.attr("add_widgets")(renderer);
      std::future future =
          std::async(std::launch::async, [](){});
      return awaitable(std::move(future));
    },
    py::is_method(py::none()));
}
```

3D Cubes - Coordinate Data

```cpp
#include <pybind11/numpy.h>

py::list get_cubes() {
    std::array colors{"c9842a", "f50707", "0040ff"};
    std::array cubes_xyz{0, 0, 0,
                         1, 1, 1,
                         2, 2, 2,
                       -1, -1, 1};
    py::list cubes;
    py::object Cube = py::module::import("cube").attr("Cube");
    for (size_t i = 0; i < cubes_xyz.size() / 3; ++i)
        cubes.append(Cube(
                          py::array(3, cubes_xyz.data() + 3*i),
                          colors[i % colors.size()]));
    return cubes;
}
```

3D Cubes - batgrl's Interactive Viewer

```
[t-rec]: Press Ctrl+D to end recording
➜ cubes git:(slides) ✗
```

Interaction powered by the `widgets.behaviors.GrabbableBehavior` class.

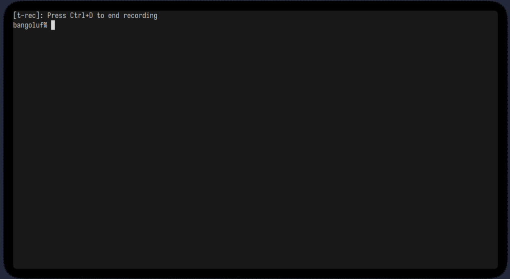
Rock Paper Scissors Intro

Toy program to use C++20 coroutines together with Python Tasks.

https://docs.python.org/3/library/asyncio-task.html

```
class Animation(Gadget):
  # ...
  play(self) -> asyncio.Task:
    # ...
    self._animation_task =
      asyncio.create_task(
        self._play_animation())
    return self._animation_task

  async def _play_animation(self):
    # simplifying, loop over frames:
    #   asyncio.sleep
    #   update frame index
```

batgrl's Animation is a sequence of frames and durations.



```
[t-rec]: Press Ctrl+D to end recording
bangoluf% 
```

Rock Paper Scissors

```cpp
PYBIND11_EMBEDDED_MODULE(rps, module) {
  py::dict members;
  members["on_start"] = py::cpp_function(
    [](py::object self) -> awaitable {
      // Background, image, and animations setup omitted.

      py::object animation = rps_library::next(animations);
      animation.attr("is_visible") = true;
      py::object animation_task = animation.attr("play")();

      std::future future =
          std::async(std::launch::async, [](){});
      return awaitable(std::move(future));
    }, py::is_method(py::none()));
}
```

Rock Paper Scissors - Next Animation

```cpp
py::object next(std::array<py::object, 3> animations) {
    std::random_device rd{};
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> randint02(0, 2);
    py::object animation = animations[randint02(gen)];
    return animation;
}
```

Rock Paper Scissors with asyncio.Task.add_done_callback

```
// PYBIND11_EMBEDDED_MODULE with members["on_start"] ...
  py::object animation = rps_library::next(animations);
  py::object animation_task = animation.attr("play")();
  animation.attr("is_visible") = true;
  animation_task.attr("add_done_callback")(py::cpp_function(
    [animations,previous_animation=animation](py::object) {
      previous_animation.attr("is_visible") = false;
      py::object animation = rps_library::next(animations);
      animation.attr("is_visible") = true;
      animation.attr("play")().attr("add_done_callback")(py::cpp_function(
        [animations,previous_animation=animation](py::object) {
          previous_animation.attr("is_visible") = false;
          py::object animation = rps_library::next(animations);
          animation.attr("is_visible") = true;
          // ...
```

From `add_done_callback` to `co_await`

Using `asyncio.Task` in C++ and deferring with a callback:

```
py::object animation_task = animation.attr("play")();
animation_task.attr("add_done_callback")(...
  animation.attr("is_visible") = false;
  next_animation = ...
)
```

Including 20's `<coroutine>` header it'd be cool to instead be able to write:

```
co_await animation.attr("play")();
animation.attr("is_visible") = false;
next_animation = ...
```

Awaitable for asyncio (w/ promise)

```cpp
class awaitable {
    std::future<void> future;
  public:
    awaitable(std::future<void>&& f) : future(std::move(f)) {}
    struct promise_type {
      awaitable get_return_object() {
          return awaitable(promise.get_future());
      }
      std::suspend_never initial_suspend() { return {}; }
      std::suspend_never final_suspend()   { return {}; }
      void return_void() { }
      void unhandled_exception() {
          promise.set_exception(std::current_exception());
      }
     private:
      std::promise<void> promise;
    };
    // await and next ...
};
```

Synchronization Update

Until now, the awaitable's await and next were each called once per run, since on_start returned without suspending. So, this was OK:

```
void next() {
    future.wait();
    throw py::stop_iteration{};
}
```

With co_await, Python repeatedly calls next, until the coroutine's finished.

```
void next() {
    using namespace std::chrono_literals;
    if (future.wait_for(3ms) == std::future_status::ready)
        throw py::stop_iteration{};
}
```

py awaiter

```cpp
py_awaiter operator co_await(py::object py_object) {
    return py_awaiter(py_object);
}
```

```cpp
class py_awaiter {
    py::object task;
  public:
    py_awaiter(py::object py_object) : task(py_object) { }

    void await_suspend(std::coroutine_handle<> handle) const {
        task.attr("add_done_callback")(py::cpp_function(
            [handle](py::object /* future */) {
                handle.resume();
            }));
    }

    // continues on the next slide
```

py awaiter and operator co_await

```cpp
class py_awaiter {
  public:
    // ... continuing py_awaiter class from the previous slide

    bool await_ready() const {
        return task.attr("done")().cast<bool>();
    }

    py::object await_resume() const {
        if (PyErr_Occurred())
            throw py::error_already_set();
        return task.attr("result")();
    }
};
```

co_await asyncio.attr("sleep")(s)

```cpp
class py_awaiter {
    py::object task;

  public:
    py_awaiter(py::object py_object) {
        if (py_object.attr("__class__")
                     .attr("__name__").cast<std::string>()
            == "coroutine") {
            py::object asyncio = py::module_::import("asyncio");
            task = asyncio.attr("create_task")(py_object);
        } else {
            task = py_object;
        }
    }
    // await_ready, await_suspend, and await_resume from the two previous
    // slides
};
```

Summary & Conclusions

- Leverage Python from C++, in contrast with the other typical direction.
- C++ customizable coroutines fit into existing asynchronous controllers, even in other PLs.
- Async/await syntax motivation in comparison with callbacks function arguments.

- I think programming puzzles + (terminal) graphics are good fun.