C++ APIs on Python

Austin Bingham
Roxar Software Solutions
austin.bingham@emerson.com

Alright gentlemen! We need to do mixed C++ and Python development.

Python calling C++

~~Python calling C++~~

C++ calling Python

zipmeme

Concept

Python extension module

C/C++ library

C-API

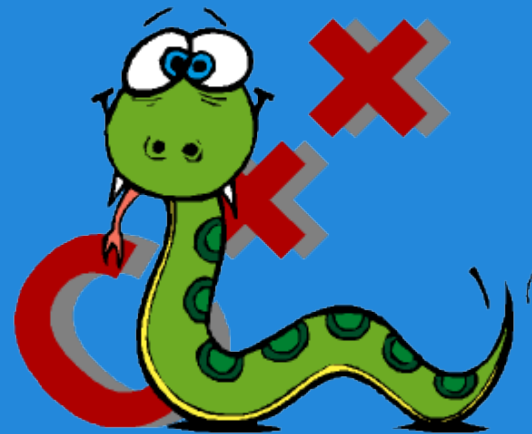
ctypes

boost.python

C/C++ application

Python interpreter

C-API



boost.python

C/C++ application

C/C++ wrapper library

Python module

Goal

Natural, idiomatic C++ APIs built on Python implementations.

```python
import logging
import sys


logging.basicConfig()
log = logging.getLogger()

log.addHandler(
    logging.StreamHandler())

log.addHandler(
    logging.StreamHandler(
        sys.stdout))
```

```cpp
#include <logging.hpp>

void initLogging() {
  basicConfig();
  Logger log = getLogger();

  // This handler will log to stderr
  log.addHandler(handlers::StreamHandler());

  // This will log to stdout
  log.addHandler(
    handlers::StreamHandler(
      boost::python::import("sys")
      .attr("stdout")));
}
```

Motivation

Expressiveness and productivity

```python
print('Hello, world.')
```

vs.

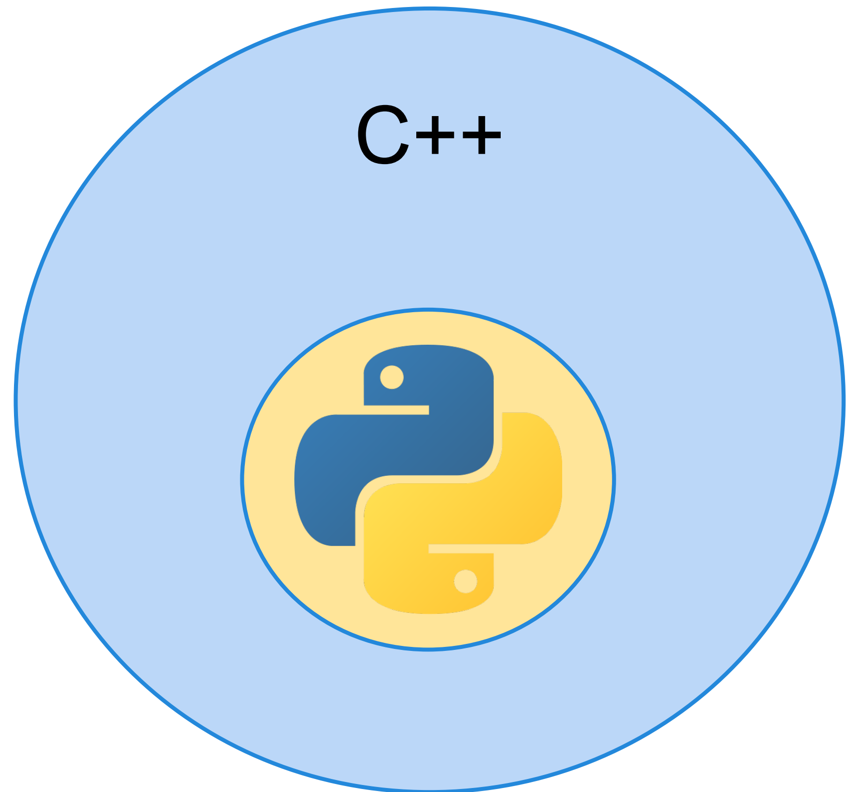
```cpp
#include <iostream>

int main(int, char**) {
  std::cout << "Hello, world."
            << std::endl;
  return 0;
}
```

Edit, Compile, Run, Debug

Access Python modules

- logging
- uuid
- re
- difflib
- shutil
- template engines

...and so on



Existing C/C++ code

Probably the encompassing reason that you would want to use the techniques in this presentation.

Progression

C/C++ application

C/C++ library

C/C++ application

C/C++ wrapper library

Python module

Python application

Extension

Python module

`main()`

C/C++ library

Python application

Python module

Python module

Techniques

Code structure

mypackage

```
namespace mypackage { . . . }
```

Module initialization

```
namespace mypackage {
  void initialize() {
    bp::object mod =
      bp::import("mypackage");

    // register type-conversion

    // anything else: create
    // loggers, etc.
  }
}
```

Type conversion

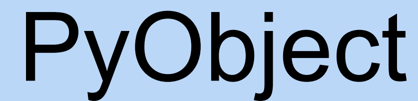
How do we get objects across the Python-C++ boundary?

The key is often to remember that PyObjects are wholly legitimate C-level entities. There is *no magic*.

Type conversion: A simple model

C++ class

bp::object obj_;



Type conversion: to C++

Python to C++ | inspection

```
if (PyObject_IsInstance(obj, class_obj))
  return new T(
    object(
      handle<>(
        borrowed(
          obj))));
```

Type conversion: to Python

C++ to Python | templates

```
template <typename T>
struct to_python_object_
{
  static PyObject* convert(const T& t)
  {
    return boost::python::incref(
      t.obj().ptr());
  }
};
```

Type conversion: registration

```
// Register from-python converter
boost::python::converter::registry::push_back(
  &convertible,
  &construct,
  boost::python::type_id<MyType>());

// Register to-python converter
boost::python::to_python_converter<
  MyType,
  to_python_object_<MyType> >();

// Covert from Python to C++
MyType x = boost::python::extract<MyType>(obj);

// Convert from C++ to Python (implicit in bp::object
constructor)
python_class.attr("doit")(x);
```

Exceptions: with C-API

```
PythonAPI_Foo();
if (PyErr_Occurred())
  respondToPythonException();
PythonAPI_Bar();
if (PyErr_Occurred())
  respondToPythonException();
```

Exceptions: `error_already_set`

Python exception

bp::error_already_set

```
try {
    some_class.attr("method")(3);
}
catch (const bp::error_already_set&) {
  // Some exception has been thrown in Python
  throw SomeException();
}
```

Exception translation

```
PyObject *t, *v, *tb;
PyErr_Fetch(&t, &v, &tb);

// Check if it's a ValueError
if (PyErr_GivenExceptionMatches(
     value_error_class_obj, t))
{
  throw ValueError();
}
```

Exception translation

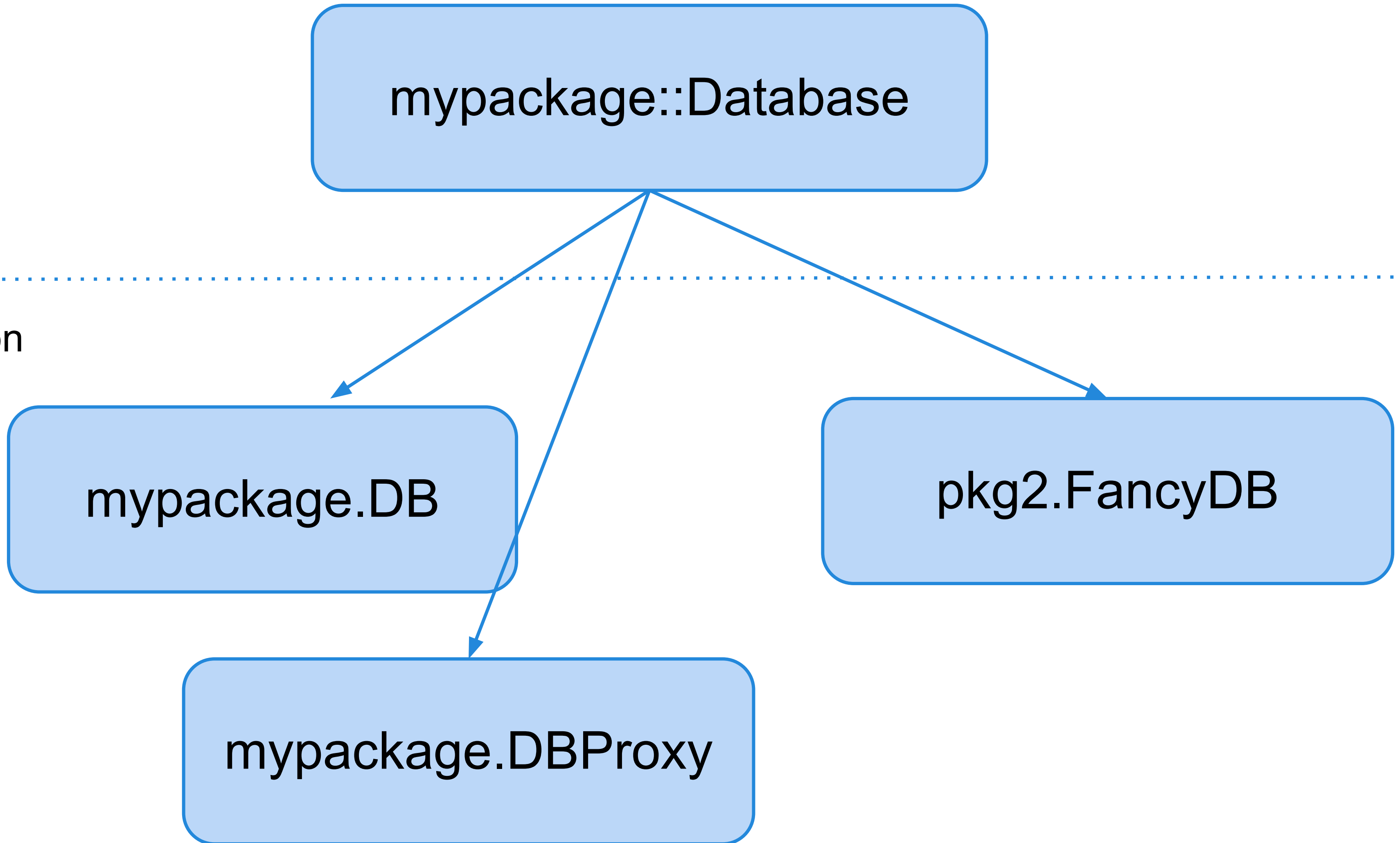
```
// The easy way!
try {
  some_python_function();
}
catch (const bp::error_already_set&) {
  ackward::core::translatePythonException();
}
```

Iteration

There's a very natural mapping between C++ and Python iterators

- C++ iterator holds a Python iterator reference
- Incrementing iterator calls `next(iter)` and stores values
- `StopIteration` is caught in increment
- The "end" iterator simply has a `None` Python iterator

Duck Typing vs. Static Typing

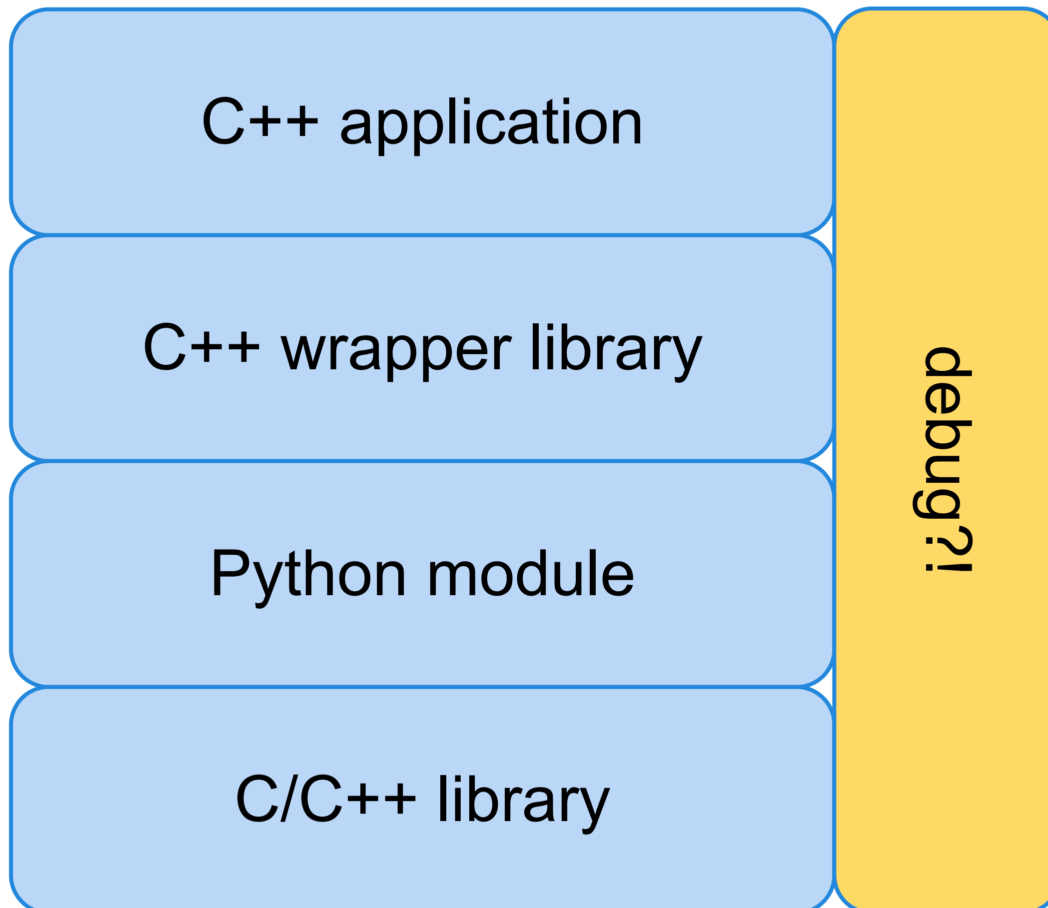


Other stuff

- Enumerations
- Const-ness
- Properties
- Tuples
- Kwargs
- Subclassing
- Lifetime management

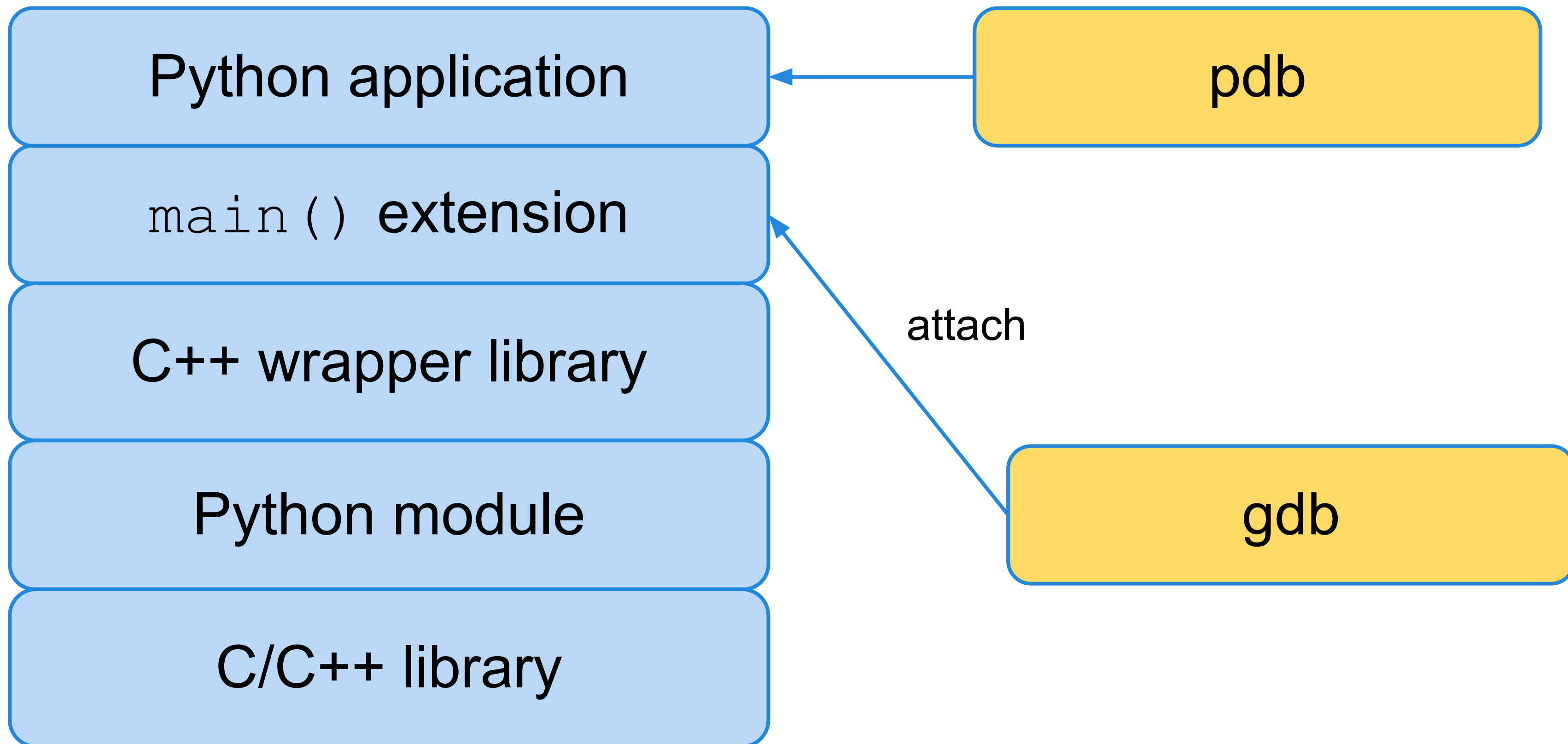
Interesting Bits

Debugging the full stack



- `pdb.set_trace()`
- gdb with python extensions
- VisualStudio python support (?)
- `SIGINT` to break debugger
- Common logging
- Debugging individual layers

Debugging: Pythonizing the App



Performance

- Python function call overhead can cause problems.
- Be aware of unnecessary data copies.
- Learn the buffer and memoryview APIs

In general, the 80/20 principle works in your favor. Performance is not an issue for most code, and when it is you can bring many tools to bear.

References

- Boost.Python

  http://www.boost.org/doc/libs/1_49_0/libs/python/doc/index.html

- Ackward

  code.google.com/p/ackward/

- Type conversion

  misspent.wordpress.com/2009/09/27/how-to-write-boost-python-converters/

- Exception translation misspent.wordpress.com/2009/10/11/boost-python-and-handling-python-exceptions/

Contact

twitter: @austin_bingham
email: austin.bingham@gmail.com