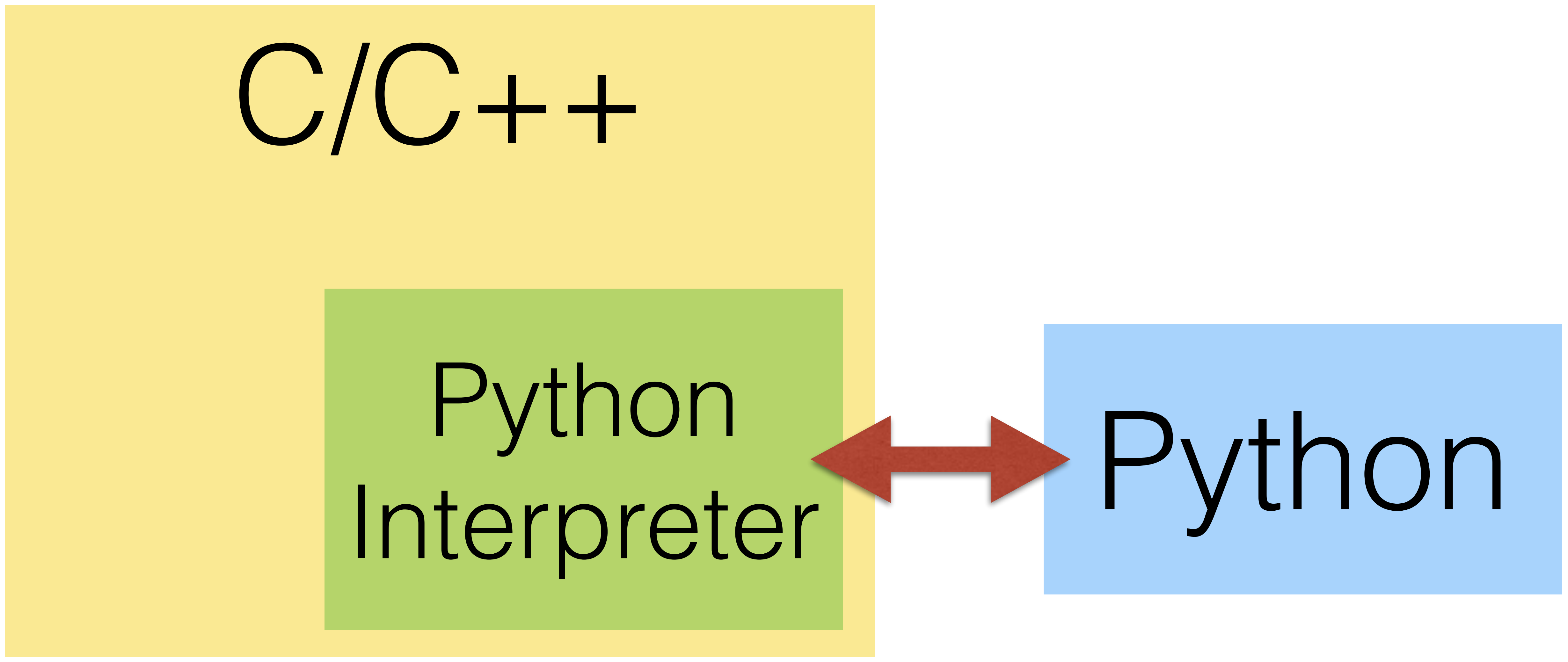
Threads and Callbacks for Embedded Python

Yi-Lung Tsai (Bruce)

Python/C++/C/Java

Coffee/Jazz/Photography

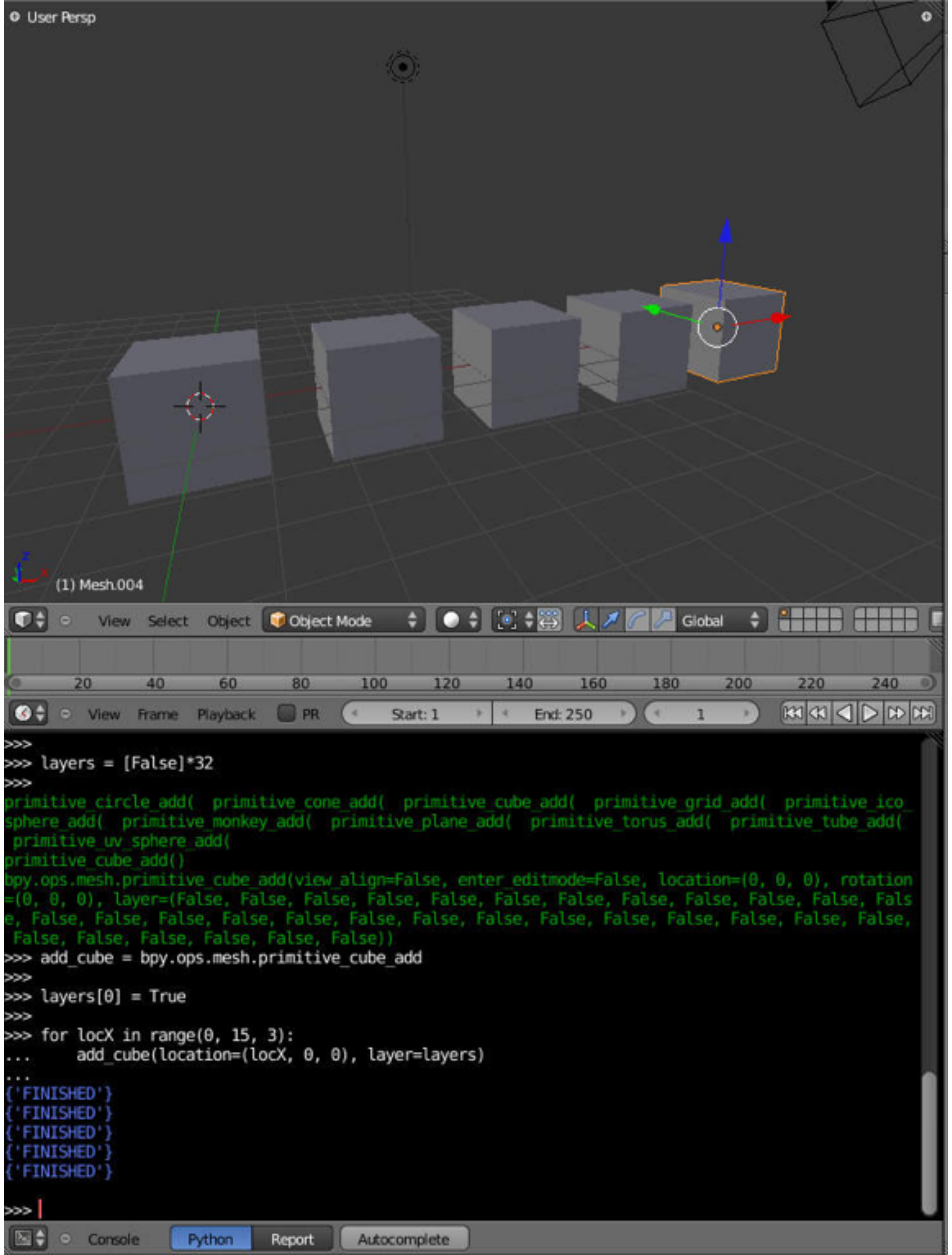
Embedding Python



Embed Python interpreter in C/C++, run a script and use the result

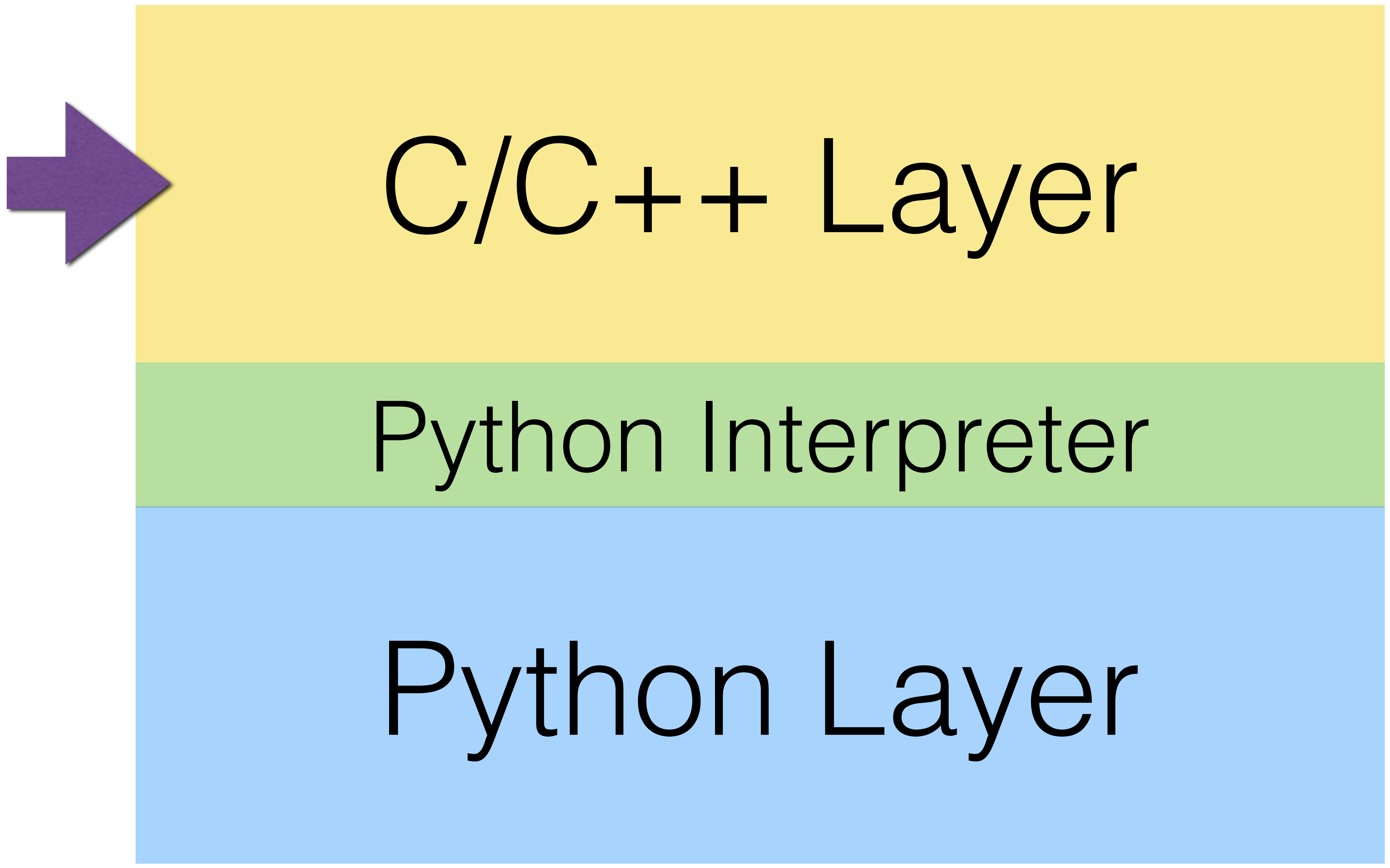
Benefit

- Running custom code without recompiling main program
- Python is more flexible or suitable for solving specific problems
- Exposing scripting API driving main program



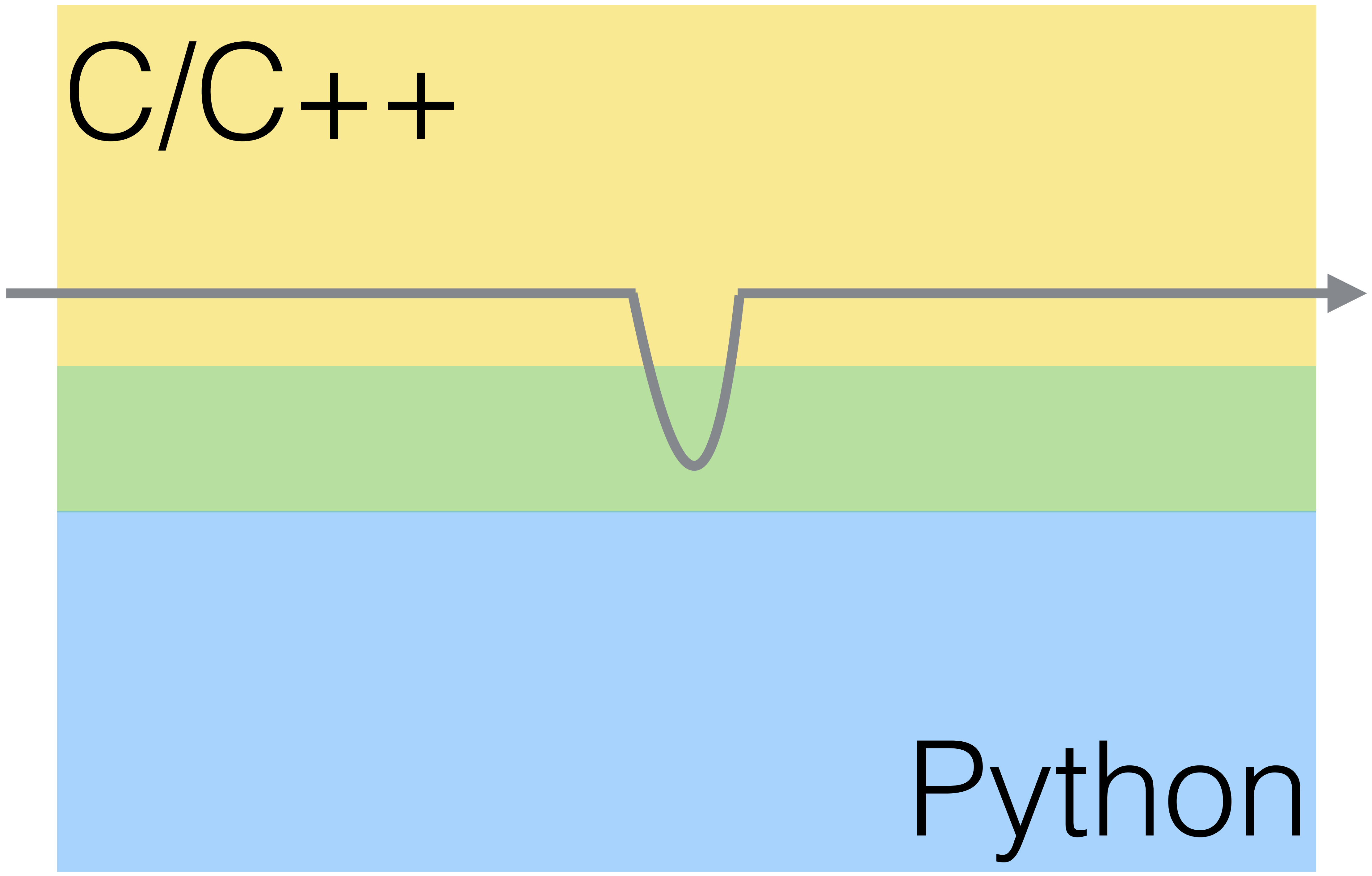
```
>>>
>>> layers = [False]*32
>>>
primitive_circle_add(  primitive_cone_add(  primitive_cube_add(  primitive_grid_add(  primitive_ico_
sphere_add(  primitive_monkey_add(  primitive_plane_add(  primitive_torus_add(  primitive_tube_add(
 primitive_uv_sphere_add(
primitive_cube_add()
bpy.ops.mesh.primitive_cube_add(view_align=False, enter_editmode=False, location=(0, 0, 0), rotation
=(0, 0, 0), layer=(False, False, False, False, False, False, False, False, False, False, False, Fals
e, False, False, False, False, False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False))
>>> add_cube = bpy.ops.mesh.primitive_cube_add
>>>
>>> layers[0] = True
>>>
>>> for locX in range(0, 15, 3):
...     add_cube(location=(locX, 0, 0), layer=layers)
...
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}
{'FINISHED'}

>>>
```

https://www.blender.org/manual/_images/Extensions-Python-Console-Example-bpy-ops.jpg


C/C++ Layer
Python Interpreter
Python Layer

First Glance

```
#include <Python.h>

int main(int argc, char* argv[])
{
    Py_Initialize();

    PyRun_SimpleString("from time import time, ctime\n"
                       "print 'Today is', ctime(time())\n");

    Py_Finalize();
    return 0;
}
```


C/C++
Python

http://fannit.com/wp-content/uploads/2014/03/0960a7436008f887aa6bbf0afd34eb722.jpg

Python/C API

- Everything is PyObject*
- Symbols with Py prefix

```c
...

PyObject* sysPath = PySys_GetObject((char*) "path");
PyList_Append(sysPath, PyString_FromString("."));

printf("Input two integers separated by space:\n");
scanf("%d %d", &a, &b);

do
{
    pModule = PyImport_ImportModule(PY_MODULE_NAME);
    if (pModule == NULL) break;

    pFunc = PyObject_GetAttrString(pModule, PY_FUNCTION_NAME);
    if (pFunc == NULL) break;

    pArgs = Py_BuildValue("ii", a, b);
    if (pArgs == NULL) break;

    pValue = PyObject_Call(pFunc, pArgs, NULL);
    if (pValue == NULL) break;
    printf("Result of call: %ld\n", PyInt_AsLong(pValue));
} while (0);

Py_XDECREF(pValue);
Py_XDECREF(pArgs);
Py_XDECREF(pFunc);
Py_XDECREF(pModule);

...
```

```python
def multiply(a, b):
    print "Will compute", a, "times", b
    c = 0
    for i in range(0, a):
        c = c + b
    return c
```

Python Program Path

```
...

PyObject* sysPath = PySys_GetObject((char*) "path");
PyList_Append(sysPath, PyString_FromString(“."));

...
```

PyObject Manipulation

- Create PyObject
- Manipulate PyObject
- Dereference PyObject

```
...

do
{
    pModule = PyImport_ImportModule(PY_MODULE_NAME);
    if (pModule == NULL) break;

    pFunc = PyObject_GetAttrString(pModule, PY_FUNCTION_NAME);
    if (pFunc == NULL) break;

    pArgs = Py_BuildValue("ii", a, b);
    if (pArgs == NULL) break;

    pValue = PyObject_Call(pFunc, pArgs, NULL);
    if (pValue == NULL) break;
    printf("Result of call: %ld\n", PyInt_AsLong(pValue));
} while (0);

Py_XDECREF(pValue);
Py_XDECREF(pArgs);
Py_XDECREF(pFunc);
Py_XDECREF(pModule);

...
```

Data Conversion

1. Convert data values from C to Python
2. Perform a function call to a Python interface routine using the converted values
3. Convert the data values from the call from Python to C

```
pArgs = Py_BuildValue("ii", a, b);
```

Format	Python type	C type
s	string	char*
i	integer	int
b	integer	char
l	integer	long int
d	float	double
o	object	PyObject*
()	tuple	
[]	list	
{}	dictionary	

Free Resource

PyObject* **`PyDict_New`**()

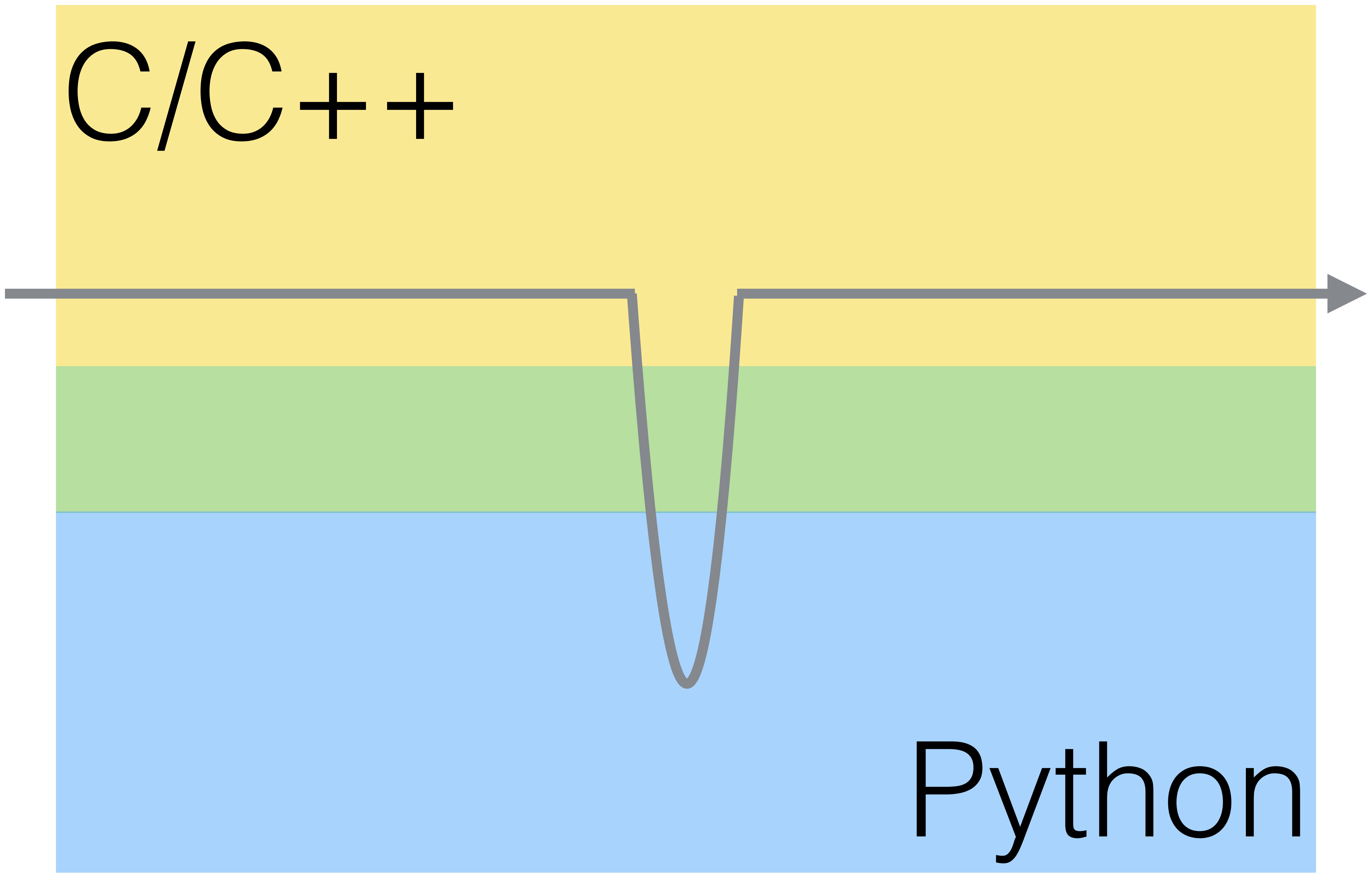
Return value: New reference.

Need to decrease reference counting

PyObject* **`PyDict_GetItem`**(PyObject **p*, PyObject **key*)

Return value: Borrowed reference.

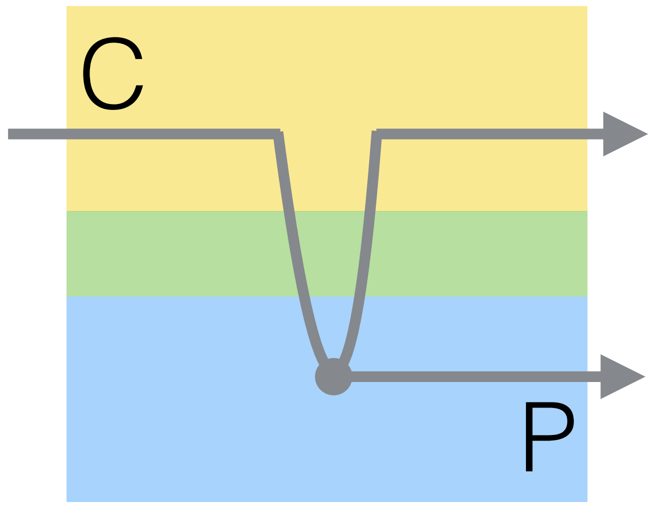
NO need to decrease reference counting


C/C++
Python

Multithreading

- Python interpreter is not fully thread-safe
- Need to deal with global interpreter lock (GIL)
- GIL ensures only one Python instruction runs at one time

```python
import threading
import time


class WorkerThread(threading.Thread):
    def __init__(self, name):
        super(WorkerThread, self).__init__(name=name)
        self.stop_event = threading.Event()

    def run(self):
        while not self.stop_event.is_set():
            print self.name, "is working"
            time.sleep(1)

    def stop(self):
        print self.name, "stop"
        self.stop_event.set()


class ThreadManager(object):
    def __init__(self):
        self.worker = WorkerThread("worker")

    def start_thread(self):
        self.worker.start()

    def stop_thread(self):
        self.worker.stop()
        self.worker.join()
```



```
...

PyEval_InitThreads();

...

do
{
    state = PyGILState_Ensure();
    {
        pModule = PyImport_ImportModule(PY_MODULE_NAME);
        if (pModule == NULL) break;

        pClass = PyObject_GetAttrString(pModule, PY_THREAD_MANAGER_CLASS);
        if (pClass == NULL) break;

        pInst = PyObject_CallObject(pClass, NULL);
        if (pInst == NULL) break;

        PyObject_CallMethod(pInst, PY_START_THREAD_FUNCTION, NULL);
    }
    PyGILState_Release(state);

    for (int i = 0; i < 5; i++)
    {
        printf("main thread is running\n");
        sleep(1);
    }

    state = PyGILState_Ensure();
    {
        PyObject_CallMethod(pInst, PY_STOP_THREAD_FUNCTION, NULL);
    }
    PyGILState_Release(state);

    printf("finish\n");
} while (0);

...
```

D’oh!

http://www.ralphhoweministries.com/sancfiles/homer.jpg

no GIL

1. PyEval_InitThreads();

   main thread acquires GIL

2. state = PyGILState_Ensure();

3. <start thread in Python> **main thread acquires GIL**

4. PyGILState_Release(state)

5. <running in C> $\boxed{\textbf{main thread acquires GIL}}$

no GIL

1. PyEval_InitThreads();

main thread acquires GIL

2. **save = PyEval_SaveThread();**

main thread releases GIL

3. state = PyGILState_Ensure();

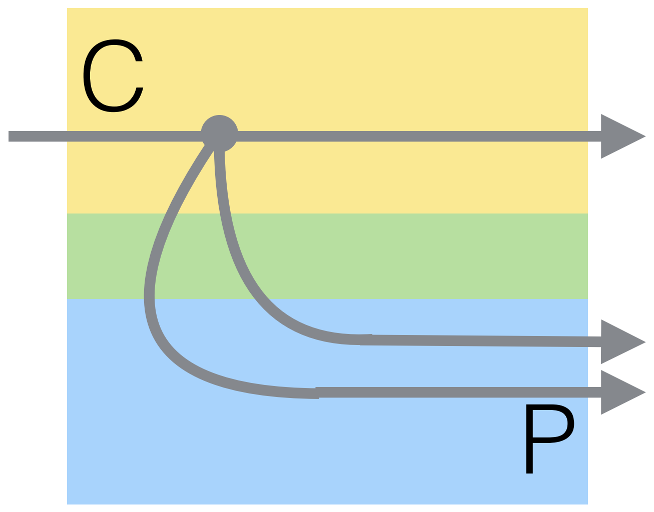
4. <start thread in Python> main thread acquires GIL

5. PyGILState_Release(state)

6. <running in C> $\boxed{\text{main thread releases GIL}}$

```
...

PyEval_InitThreads();

...

PyThreadState* save = PyEval_SaveThread();

do
{
    state = PyGILState_Ensure();
    {
        pModule = PyImport_ImportModule(PY_MODULE_NAME);
        if (pModule == NULL) break;

        pClass = PyObject_GetAttrString(pModule, PY_THREAD_MANAGER_CLASS);
        if (pClass == NULL) break;

        pInst = PyObject_CallObject(pClass, NULL);
        if (pInst == NULL) break;

        PyObject_CallMethod(pInst, PY_START_THREAD_FUNCTION, NULL);
    }
    PyGILState_Release(state);

    for (int i = 0; i < 5; i++)
    {
        printf("main thread is running\n");
        sleep(1);
    }

    state = PyGILState_Ensure();
    {
        PyObject_CallMethod(pInst, PY_STOP_THREAD_FUNCTION, NULL);
    }
    PyGILState_Release(state);

    printf("finish\n");
} while (0);

PyEval_RestoreThread(save);

...
```

Multithreading - 2



- Create threads in C
- Each C thread calls Python function

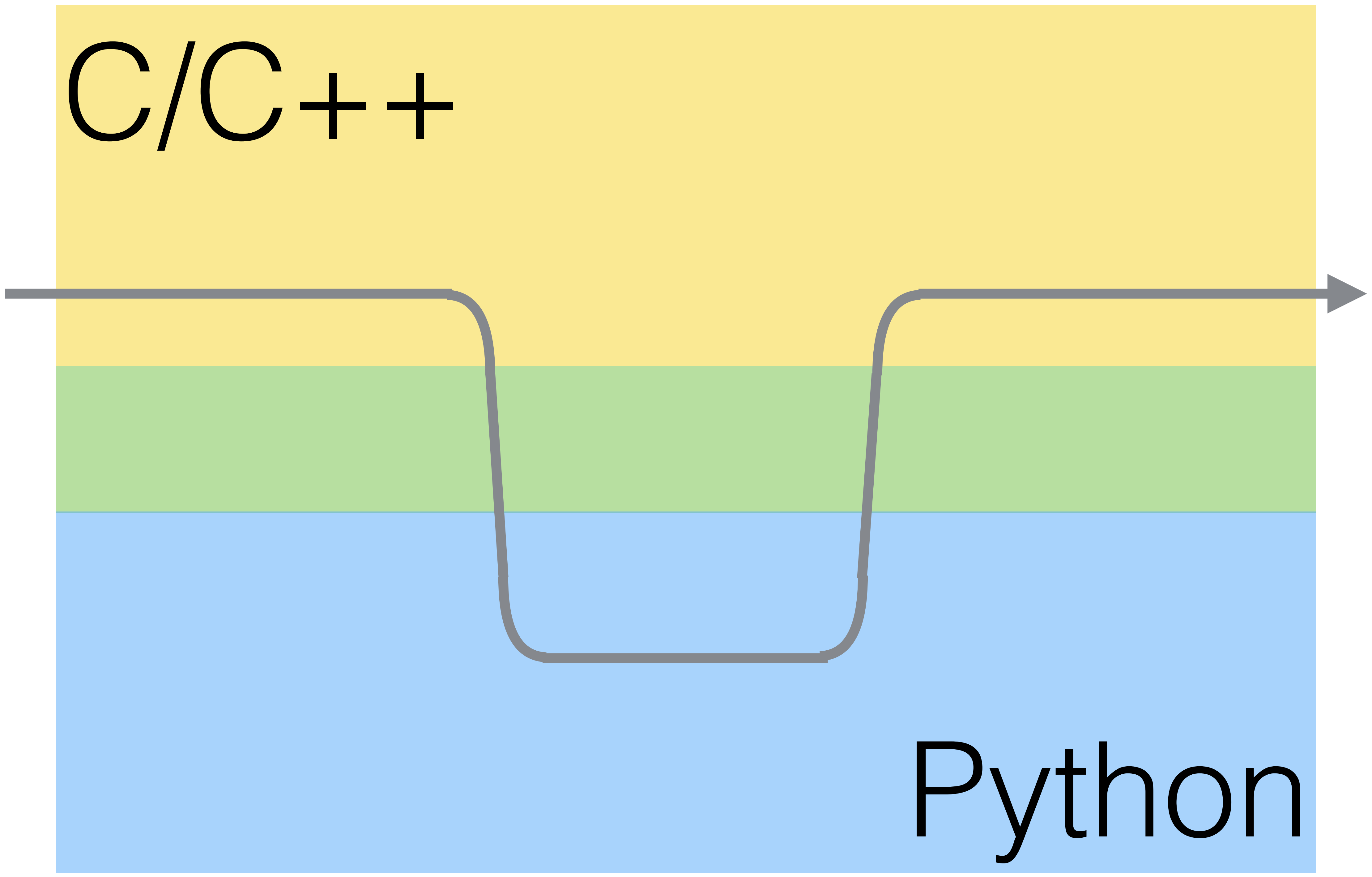
```
int main(int argc, char* argv[])
{
    PyEval_InitThreads();
    Py_Initialize();
    PyObject* sysPath = PySys_GetObject((char*) "path");
    PyList_Append(sysPath, PyString_FromString("."));

    PyThreadState* save = PyEval_SaveThread();

    pthread_t tid1, tid2;
    char* tname1 = "worker1";
    char* tname2 = "worker2";
    pthread_create(&tid1, NULL, &run_python_function, &tname1);
    pthread_create(&tid2, NULL, &run_python_function, &tname2);

    for (int i = 0; i < 5; i++)
    {
        printf("main thread is running\n");
        sleep(1);
    }

    stop_event = 1;
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("finish\n");

    PyEval_RestoreThread(save);
    Py_Finalize();

    pthread_exit(NULL);

    return 0;
}
```

```c
void* run_python_function(void* arg)
{
    PyGILState_STATE state = PyGILState_Ensure();

    char* name = *((char**) arg);
    PyObject *pModule = NULL, *pFunc = NULL;

    do
    {
        pModule = PyImport_ImportModule(PY_MODULE_NAME);
        if (pModule == NULL) break;

        pFunc = PyObject_GetAttrString(pModule, PY_WORKING_FUNCTION);
        if (pFunc == NULL) break;

        while (!stop_event)
        {
            PyObject_CallFunction(pFunc, "s", name);
        }
    } while (0);

    Py_XDECREF(pFunc);
    Py_XDECREF(pModule);

    PyGILState_Release(state);

    pthread_exit(NULL);

    return NULL;
}
```

```python
import time


def working(name):
    print name, "is working"
    time.sleep(1)
```

Callback Function

- Asynchronous method invocation
- Push event back to C
- Pass heavy computation to C


C/C++
Python

```
...

static char* MODULE_NAME = "cmodule";
static char* FUNCTION_NAME = "callback";
static PyObject* callback(PyObject* self, PyObject* args)
{
    long c;
    PyArg_ParseTuple(args, "l", &c);
    printf("Result of call: %ld\n", c);

    Py_RETURN_NONE;
}

int main(int argc, char* argv[])
{
    ...

    do
    {
        PyMethodDef CFunctions[] = {
            {FUNCTION_NAME, callback, METH_VARARGS, ""},
            {NULL, NULL, 0, NULL}
        };
        Py_InitModule(MODULE_NAME, CFunctions);

        pModule = PyImport_ImportModule(PY_MODULE_NAME);
        if (pModule == NULL) break;

        pFunc = PyObject_GetAttrString(pModule, PY_FUNCTION_NAME);
        if (pFunc == NULL) break;

        pArgs = Py_BuildValue("ii", a, b);
        if (pArgs == NULL) break;

        PyObject_Call(pFunc, pArgs, NULL);
    } while (0);

    ...
}
```

Create New Module

```
...

    PyMethodDef CFunctions[] = {
        {FUNCTION_NAME, callback, METH_VARARGS, ""},
        {NULL, NULL, 0, NULL}
    };
    Py_InitModule(MODULE_NAME, CFunctions);

...
```

Import and Use

```python
import cmodule


def multiply(a, b):
    print "Will compute", a, "times", b
    c = 0
    for i in range(0, a):
        c = c + b
    cmodule.callback(c)
```

Callback to C

```
...

static PyObject* callback(PyObject* self, PyObject* args)
{
    long c;
    PyArg_ParseTuple(args, "l", &c);
    printf("Result of call: %ld\n", c);

    Py_RETURN_NONE;
}

...
```

http://pre05.deviantart.net/8b23/th/pre/f/2011/288/e/1/not_bad_by_rober_raik-d4cwbtb.png

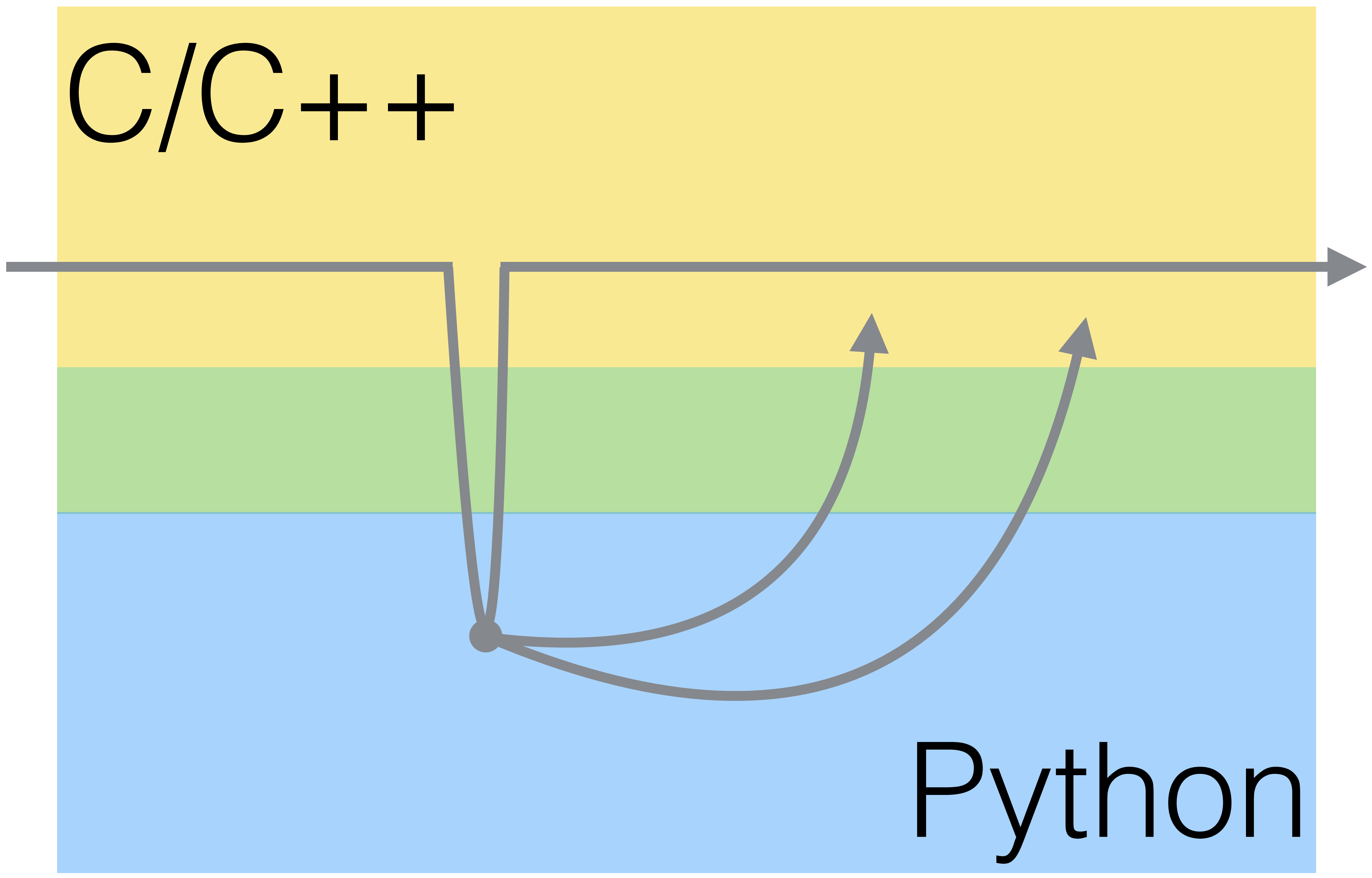
Callback as Argument

```
def multiply(a, b, callback):
    print "Will compute", a, "times", b
    c = 0
    for i in range(0, a):
        c = c + b
    callback(c)
```

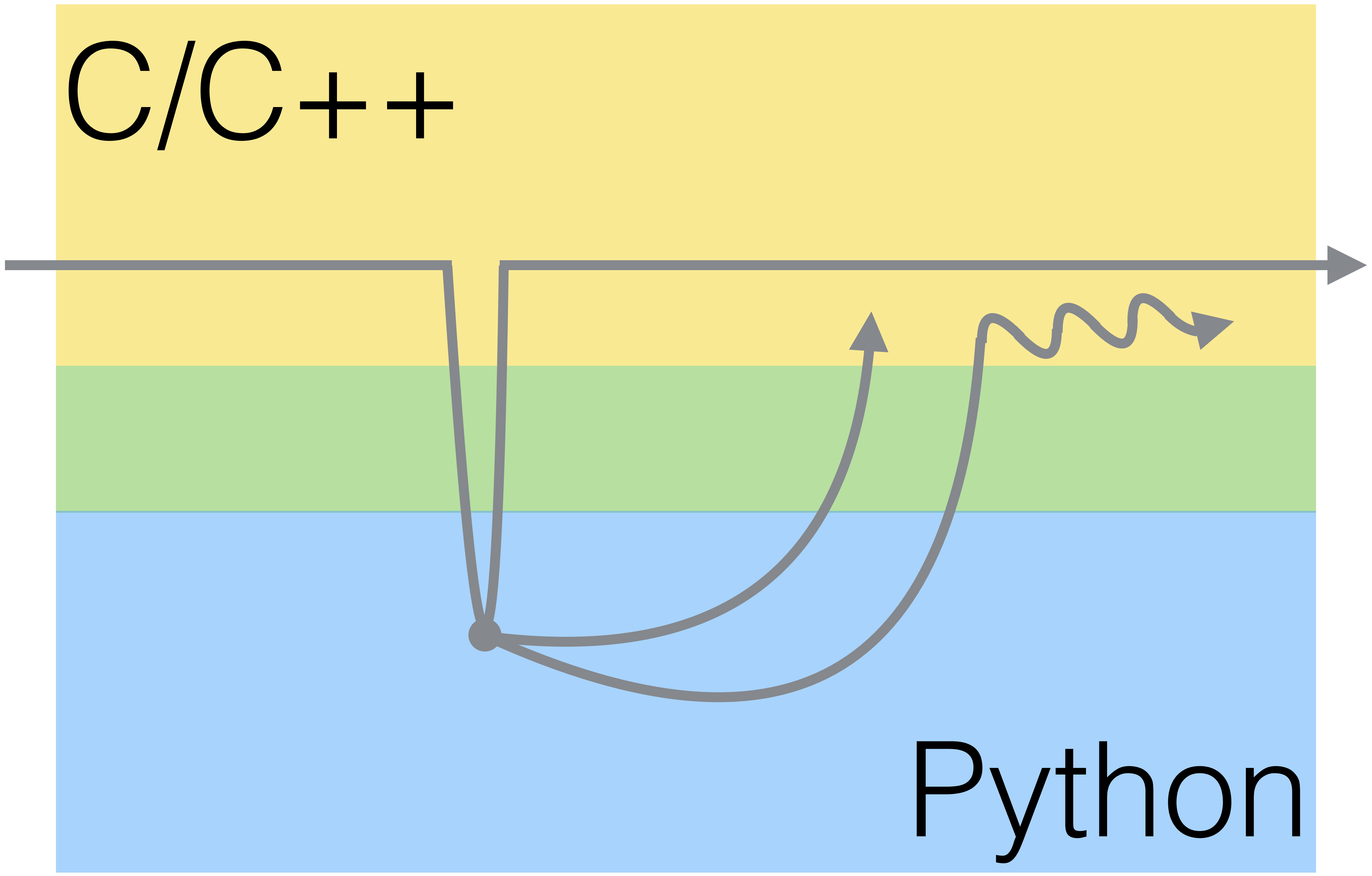
CFunction Object

```
...

do
{
    PyMethodDef CFunc = {FUNCTION_NAME, callback, METH_VARARGS, ""};
    pCallbackFunc = PyCFunction_New(&CFunc, NULL);
    if (pCallbackFunc == NULL) break;

    pModule = PyImport_ImportModule(PY_MODULE_NAME);
    if (pModule == NULL) break;

    pFunc = PyObject_GetAttrString(pModule, PY_FUNCTION_NAME);
    if (pFunc == NULL) break;

    pArgs = Py_BuildValue("iiO", a, b, pCallbackFunc);
    if (pArgs == NULL) break;

    PyObject_Call(pFunc, pArgs, NULL);
} while (0);

...
```

Event Loop

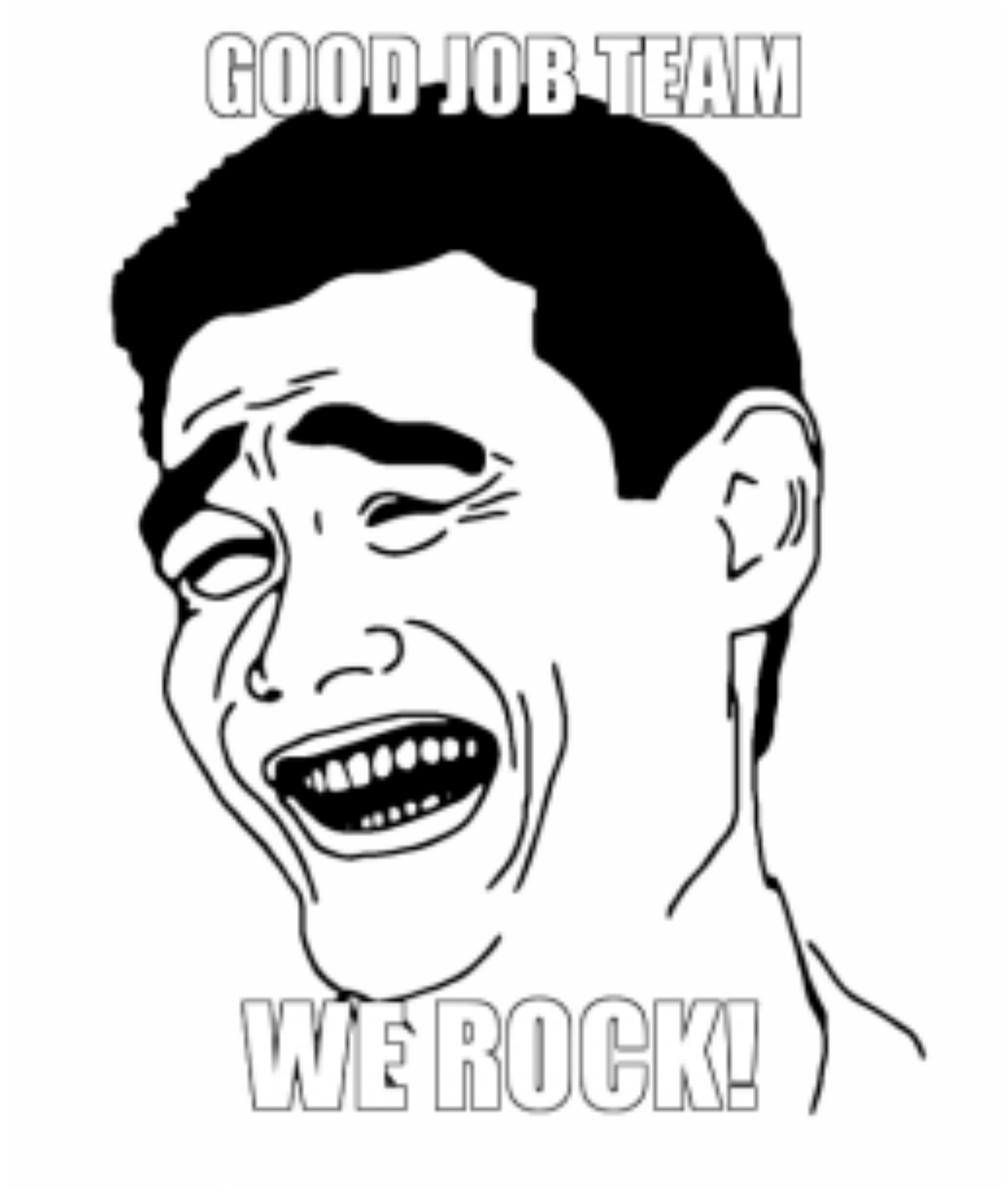


Blocking Code



```c
static PyObject* callback1(PyObject* self, PyObject* args)
{
    char* str;
    PyArg_ParseTuple(args, "s", &str);
    printf("%s push event\n", str);

    Py_RETURN_NONE;
}
static PyObject* callback2(PyObject* self, PyObject* args)
{
    char* str;
    PyArg_ParseTuple(args, "s", &str);
    printf("%s start heavy computing\n", str);
    sleep(5);
    printf("%s end heavy computing\n", str);

    Py_RETURN_NONE;
}
```

```c
static PyObject* callback1(PyObject* self, PyObject* args)
{
    char* str;
    PyArg_ParseTuple(args, "s", &str);
    printf("%s push event\n", str);

    Py_RETURN_NONE;
}
static PyObject* callback2(PyObject* self, PyObject* args)
{
    char* str;
    PyArg_ParseTuple(args, "s", &str);
    printf("%s start heavy computing\n", str);

    PyThreadState* save = PyEval_SaveThread();
    {
        sleep(5);
    }
    PyEval_RestoreThread(save);

    printf("%s end heavy computing\n", str);

    Py_RETURN_NONE;
}
```


GOOD JOB TEAM
WE ROCK!

Summary

- Python as script plug-in to C application
- PyObject manipulation in C
- GIL in multithreading
- CFunction object as callback
- Multithreading with callback