Digital plumbing with Python/C

Kyle Lanclos
W. M. Keck Observatory
klanclos@keck.hawaii.edu

*Image credit:**Tomasz Sienicki** tomasz.sienicki at gmail.com (via wikimedia)*

Why should we care about C?

Compatibility.

Complex components are far more likely to include a shared library than a Python module. This is especially true for older systems, or systems designed with portability in mind.

Why should we care about C?

Speed.

Your very best Python code will never be faster than the same algorithm implemented in C. You’re also free to use any available threading (multiprocessing) approach.

Why should we care about C?

Control.

When you call a C routine you know what you’re getting: exactly what you see and nothing more. This has particular value when interacting with complex vendor-supplied API’s. You can also carry around non-Python data with Python/C class instances.

What's the downside of using C?

Complexity.

The Python/C API is very well done but your code will always be more verbose in C than it is in Python. You also have the added worry of maintaining a build environment to turn your Python/C code into a shared library.

What’s the downside of using C?

Decreased agility.

The goal is to limit yourself to using as little C as possible so that you can continue working in Python. Making sweeping changes in a body of C code is always more time-consuming than doing the same in Python.

Learning more about Python/C

Start (and end) with the official Python documentation. It's really good, but finding the specific topic you're looking for can be challenging. Google searches can fill in the gaps.

https://docs.python.org/2/c-api/intro.html

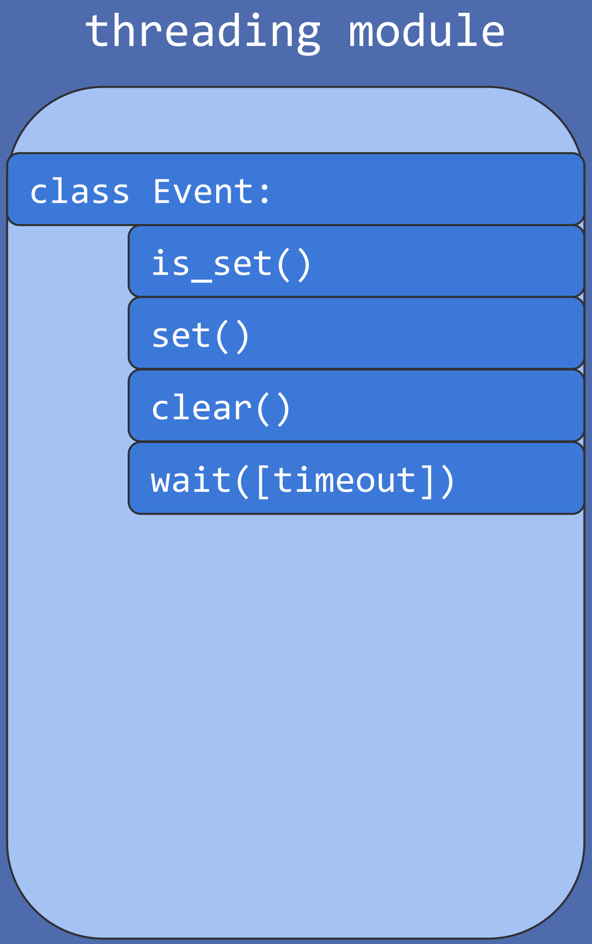
https://docs.python.org/3/c-api/intro.html

Actual usage

Image credit: Tomas Castelazo https://commons.wikimedia.org/wiki/User:Tomascastelazo

Fast threading.Event primitive in Python/C

The threading.Event class is used as a signaling mechanism between threads. Basic usage typically involves one thread clearing an event instance and waiting for it to be set; some other thread sets the event when a condition has been reached; the original thread then proceeds executing. The stock Event class is implemented in pure Python and is regrettably slow. A re-implementation using pthread primitives is an order of magnitude faster.



C structure definition

```python
class Event:
```

```c
typedef struct Event {
  PyObject_HEAD
  bool flag;
  int blockers;
  pthread_cond_t primary_condition;
  pthread_cond_t signal_condition;
  pthread_mutex_t primary_mutex;
  pthread_mutex_t signal_mutex;
} Event;
```

Actual code

`is_set()`

```
static PyObject *
Event_isSet (Event *self, PyObject *args) {

  Py_BEGIN_ALLOW_THREADS
  pthread_mutex_lock (&self->primary_mutex);
  Py_END_ALLOW_THREADS

  current = self->flag;
  pthread_mutex_unlock (&self->primary_mutex);

  if (current == FALSE) {
       result = Py_False;
  } else {
       result = Py_True;
  }

  Py_INCREF (result);
  return result;
}
```

Actual code

set()

```
static PyObject *
Event_set (Event *self, PyObject *args) {

  Py_BEGIN_ALLOW_THREADS
  pthread_mutex_lock (&self->primary_mutex);
  Py_END_ALLOW_THREADS

  self->flag = TRUE;
  pthread_mutex_unlock (&self->primary_mutex);
  pthread_cond_broadcast (&self->primary_condition);

  Py_RETURN_NONE;
}
```

Actual code

I can make it fit, I know I can...

```
clear()
```

```
static PyObject *
Event_clear (Event *self, PyObject *args) {

  Py_BEGIN_ALLOW_THREADS
  pthread_mutex_lock (&self->primary_mutex);
  Py_END_ALLOW_THREADS
  if (self->flag == TRUE) {
       pthread_mutex_unlock (&self->primary_mutex);
       Py_BEGIN_ALLOW_THREADS
       pthread_mutex_lock (&self->signal_mutex);
       if (self->blockers > 0) {
              pthread_cond_wait (&self->signal_condition, &self->signal_mutex);
       }
       pthread_mutex_unlock (&self->signal_mutex);
       pthread_mutex_lock   (&self->primary_mutex);
       Py_END_ALLOW_THREADS
  }
  self->flag = FALSE;
  pthread_mutex_unlock (&self->primary_mutex);
  Py_RETURN_NONE;
}
```

Actual code

OK, fine, it doesn't all fit.

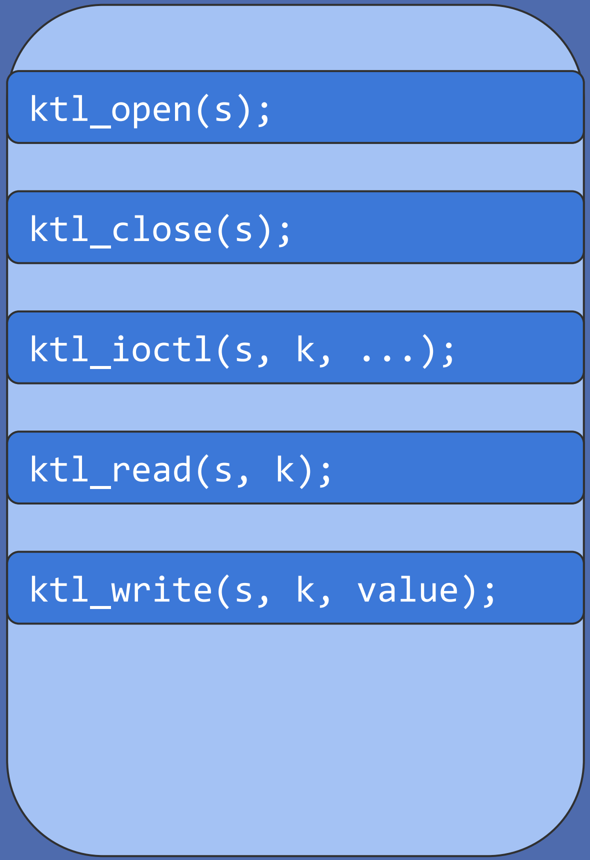
```
static PyObject *
Event_wait (Event *self, PyObject *args, PyObject *kwargs) {

  Py_BEGIN_ALLOW_THREADS
  pthread_mutex_lock (&self->primary_mutex);
  Py_END_ALLOW_THREADS

  current = self->flag;
  pthread_mutex_unlock (&self->primary_mutex);

  if (current == TRUE) {
      Py_INCREF (Py_True);
      return Py_True;
  }

  /* That was the simple case, which could be handled prior to dealing with
   * function arguments. Actual waiting involves timeout calculations,
   * acquiring mutexes, and waiting on pthread conditions. Return True if
   * signaled during the wait and False if a timeout occurred.
   */
}
```

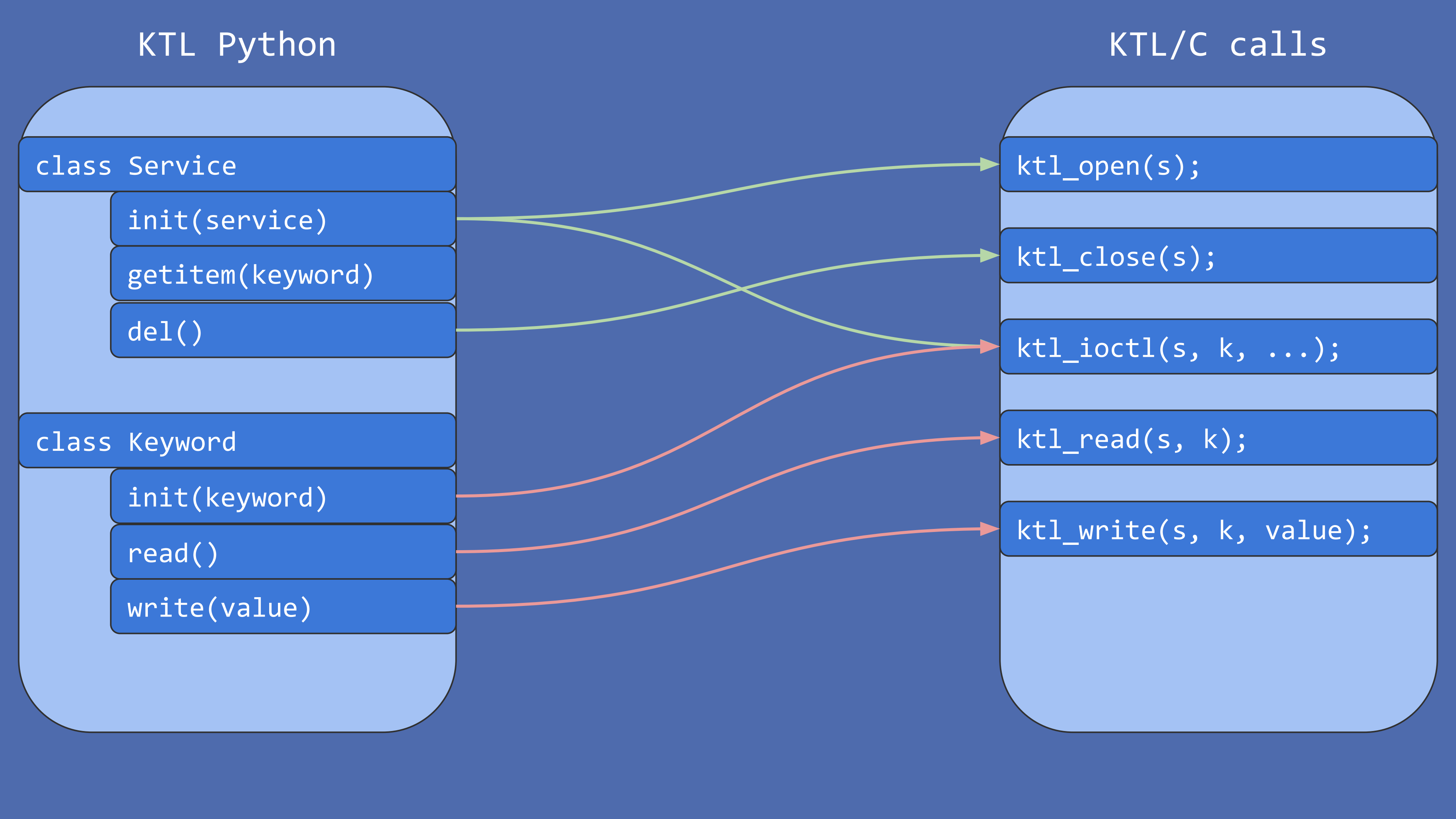
`wait([timeout])`

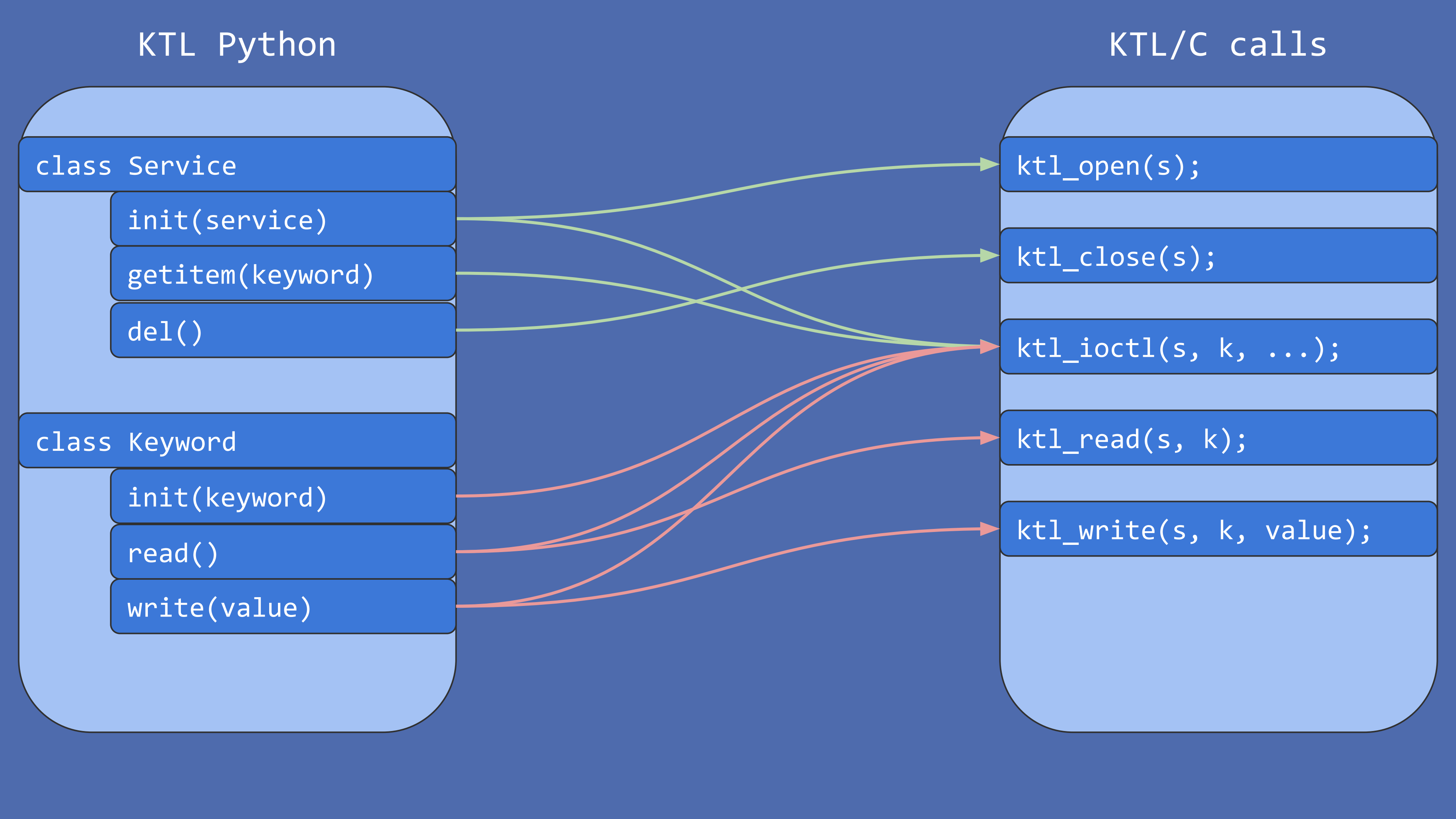
KTL Python architecture overview

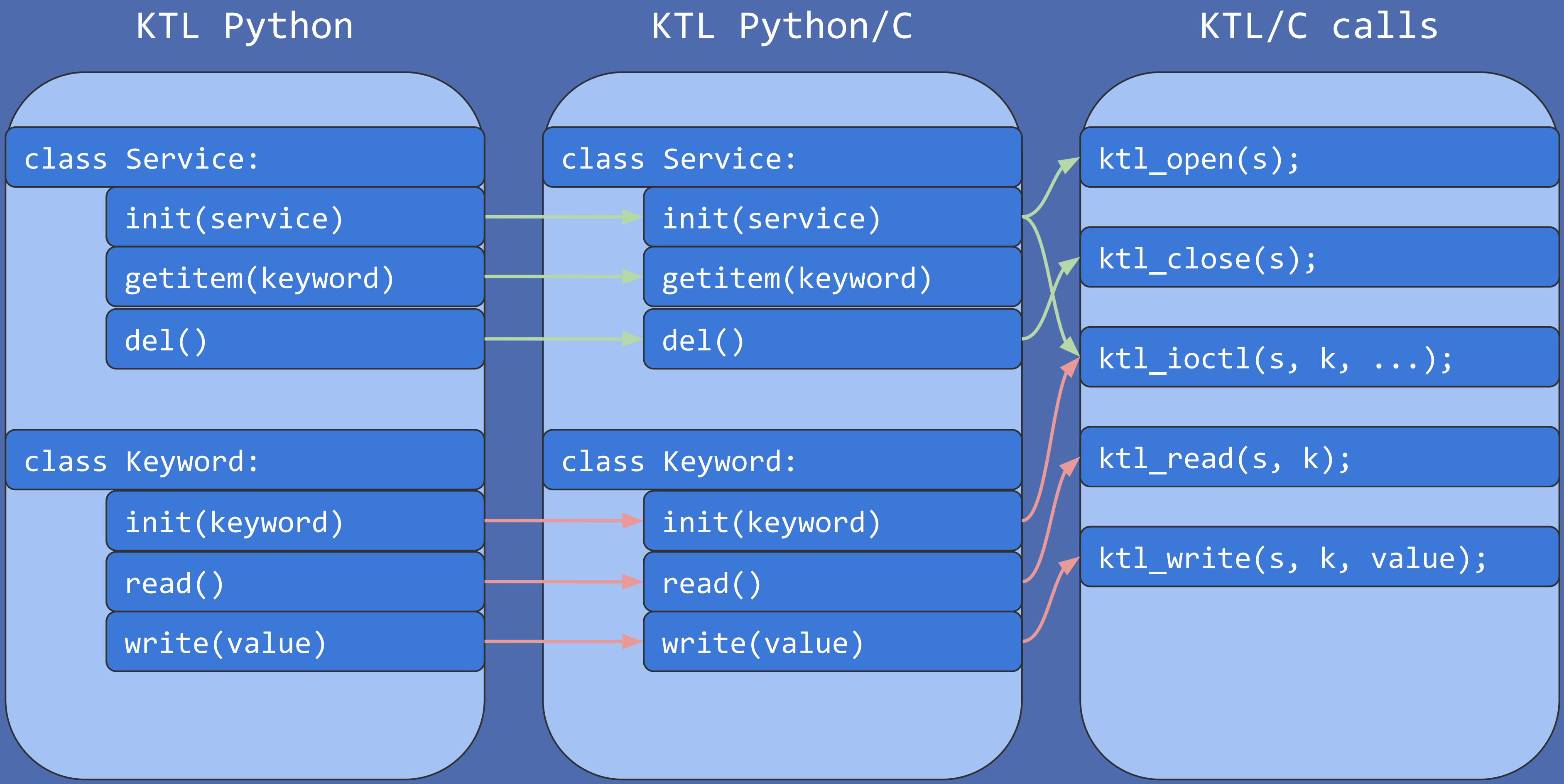
The KTL/C API presents a standard set of functions to interact with keyword/value pairs. A service is comprised of one or more (potentially thousands) of keywords; a service must be opened before it can receive queries; both services and keywords have metadata associated with them that reflect their capabilities and overall function. We want to handle high level complexity in pure Python while cleanly interfacing with the KTL/C API.

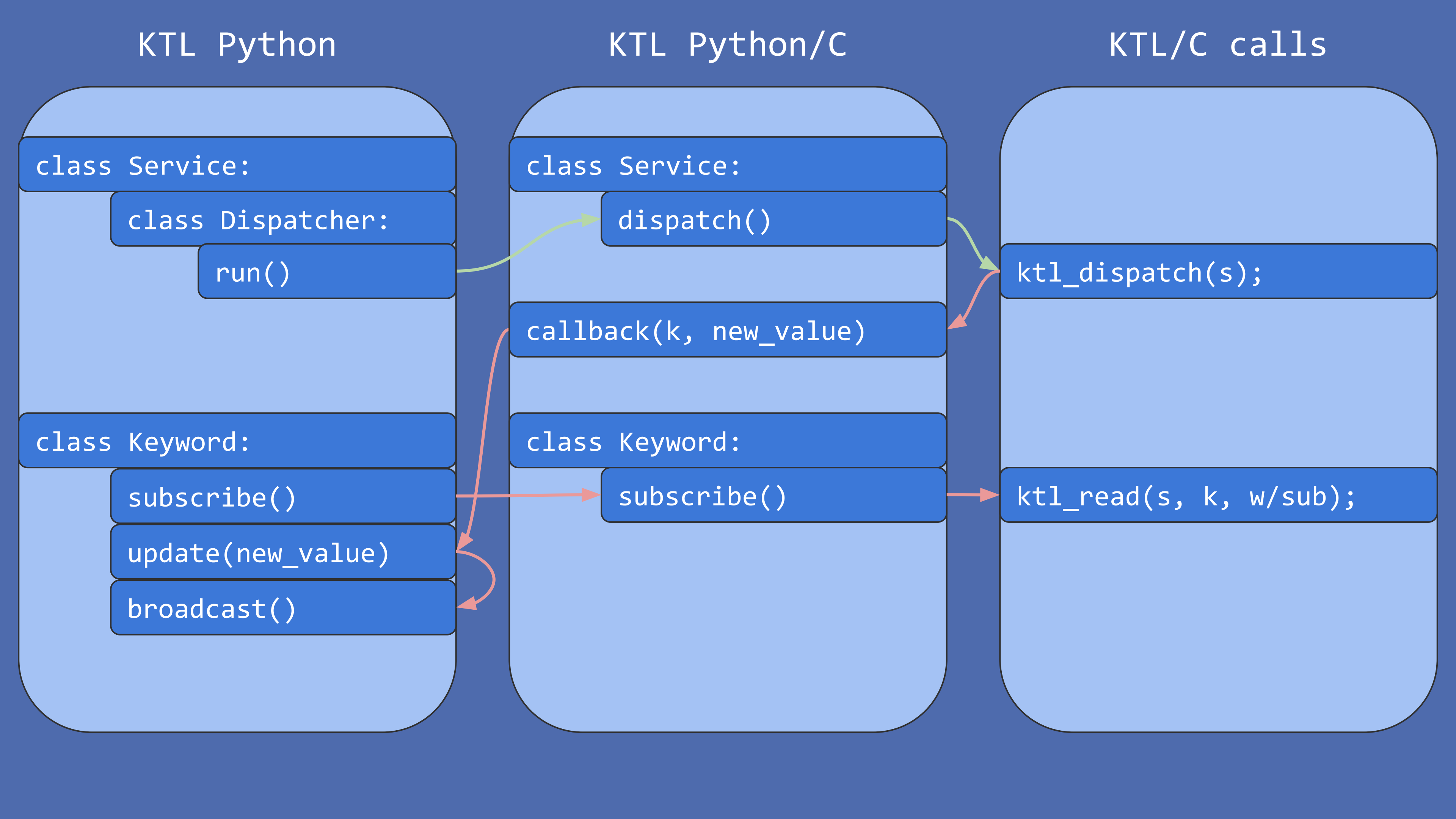
KTL/C calls




KTL Python
KTL/C calls
class Service
init(service)
getitem(keyword)
del()
class Keyword
init(keyword)
read()
write(value)
ktl_open(s);
ktl_close(s);
ktl_ioctl(s, k, ...);
ktl_read(s, k);
ktl_write(s, k, value);


KTL Python
class Service
init(service)
getitem(keyword)
del()
class Keyword
init(keyword)
read()
write(value)
KTL/C calls
ktl_open(s);
ktl_close(s);
ktl_ioctl(s, k, ...);
ktl_read(s, k);
ktl_write(s, k, value);


KTL Python
KTL Python/C
KTL/C calls
class Service:
init(service)
getitem(keyword)
del()
class Keyword:
init(keyword)
read()
write(value)
class Service:
init(service)
getitem(keyword)
del()
class Keyword:
init(keyword)
read()
write(value)
ktl_open(s);
ktl_close(s);
ktl_ioctl(s, k, ...);
ktl_read(s, k);
ktl_write(s, k, value);


KTL Python
KTL Python/C
KTL/C calls
class Service:
class Dispatcher:
run()
class Keyword:
subscribe()
update(new_value)
broadcast()
class Service:
dispatch()
callback(k, new_value)
class Keyword:
subscribe()
ktl_dispatch(s);
ktl_read(s, k, w/sub);

Example code

```
static int
Service_init (Service *self, PyObject *args, PyObject *kwargs) {

        /* Argument processing and boilerplate omitted for brevity’s sake. */

        ktl_open (name, "keyword", 0, &self->handle);
        ktl_ioctl (self->handle, KTL_CONNREG, alertNotifyHandler, self) < 0);
        ktl_ioctl (self->handle, KTL_FDREG, alertNotifyHandler, self) < 0);
        ktl_ioctl (self->handle, KTL_SUPERSUPP, &self->superpoly);
        ktl_ioctl (self->handle, KTL_IMPLCAPS, &capabilities);
        ktl_ioctl (self->handle, KTL_OBJECTS, &quantity, &keywords);

        /* Lots and lots of error checking omitted for the same reason. */
}
```

Example code

```
static PyObject *
Keyword_read (Keyword *self, PyObject *args, PyObject *kwargs) {

	/* Likewise missing all the boilerplate. */

	ktl_context_create (self->service->handle, readNotifyHandler, self, NULL, &context);

	if (self->service->notify == Py_True) {
		ktl_read (self->service->handle,
		          KTL_NOTIFY | self->service->superpoly,
		          self->name, self, NULL, context);
	} else {
		ktl_read (self->service->handle,
		          KTL_WAIT | self->service->superpoly,
		          self->name, NULL, &anypoly, NULL);
	}
}
```

Image credit: W. Carter https://commons.wikimedia.org/wiki/User:W.carter

The basics

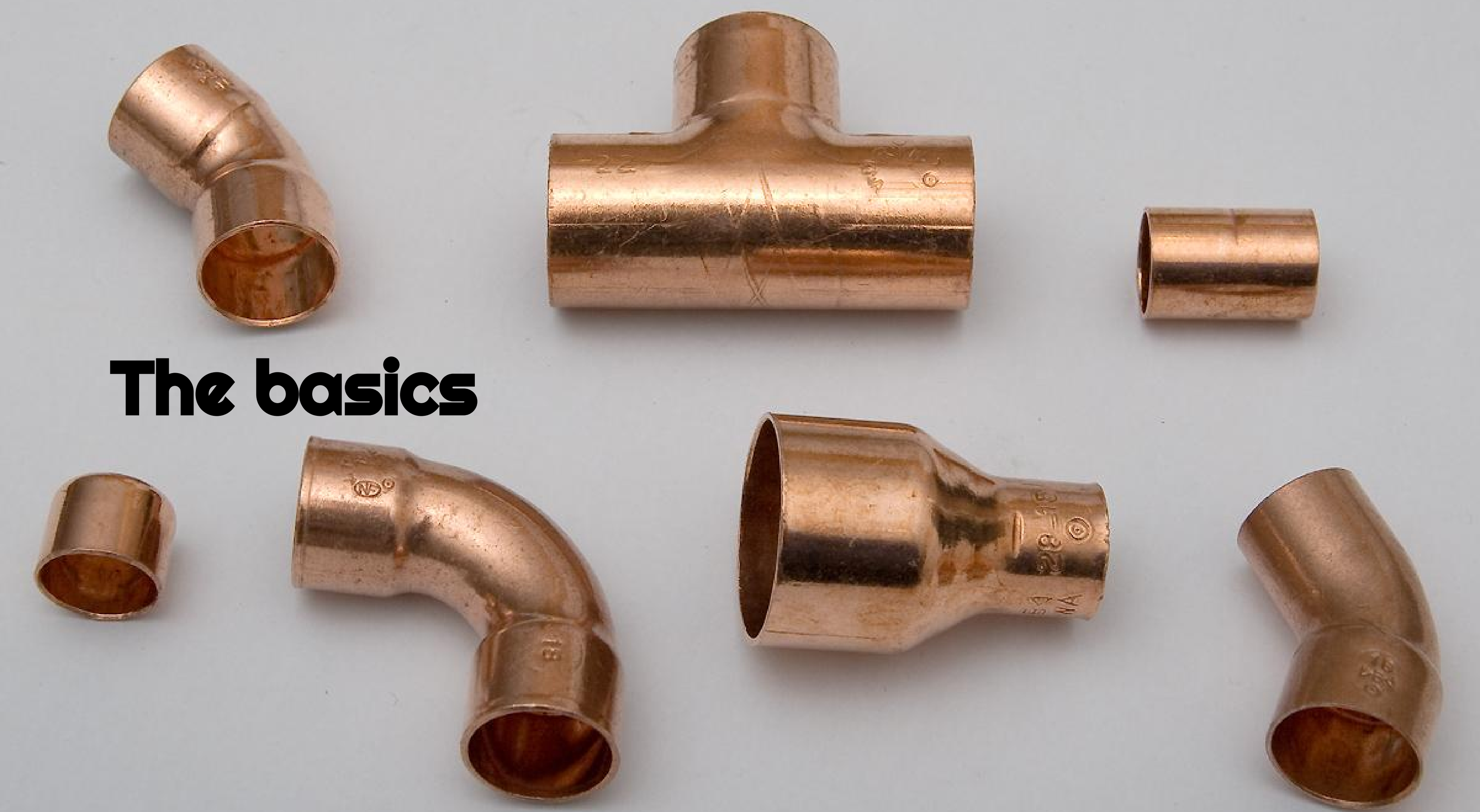


Image credit:Torsten Bätge https://commons.wikimedia.org/wiki/User:Torsten_B%C3%A4tge

All Python functionality is available in C

Everything you create is contained within a single module as seen from Python.

You can define functions, classes, and class methods.

You can initialize module contents (constants and the like).

You can call native Python code directly from your C code, but this isn’t an efficient use of the Python/C layer. Use sparingly.

Calling your code

```
import my_c_module

my_c_module.someFunction(thing1, thing2, foobar='A keyword argument')

class_instance = my_c_module.SomeClass(thing3, thing4, sauce='teriyaki')

class_instance.doThing()
class_instance.doTheOtherThing()
```

In short: it's just Python code. You don't do anything special to call Python/C functions, instantiate classes, etc. That's good because that's how a significant chunk of the Python standard library is implemented!

Backwards compatibility

There's no serious obstacle to backwards compatibility going back to at least Python 2.5. You have to tweak the way you define things in your C header file, and you have to use a couple conditionals when you define your module in the C layer, but it's not outrageous.

If your C code is just one part of a module (this is common) you have to choose between Python 2.4 and 3.x. The way you structure a Python module in 2.4 isn't forwards-compatible.

Backwards compatibility (.h snippet)

```
#ifndef PyMODINIT_FUNC          /* boilerplate macro for DLL import/export */
  #define PyMODINIT_FUNC void
#endif

#ifndef T_BOOL                  /* T_BOOL was established in Python 2.6. */
  #define T_BOOL T_INT
#endif

#if PY_VERSION_HEX < 0x02060000
  #ifndef lenfunc               /* PyMappingMethods field change in Python 2.6. */
        #define lenfunc inquiry
  #endif

  #ifndef PyVarObject_HEAD_INIT /* New way to define classes in Python 2.6. */
        #define PyVarObject_HEAD_INIT(type, size) \
                PyObject_HEAD_INIT(type) size,
  #endif

  #ifndef Py_TYPE               /* Forward-compatible way to access ob_type. */
        #define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
  #endif
#endif

/* Portable workaround for: https://bugs.python.org/issue15657 */
#define COMBO_ARGS METH_VARARGS | METH_KEYWORDS
```

Forwards compatibility (.h snippet)

```
#if PY_MAJOR_VERSION >= 3
  #define PyInt_AsLong PyLong_AsLong
  #define PyInt_Check PyLong_Check
  #define PyInt_FromLong PyLong_FromLong

  #define PyNumber_Int PyNumber_Long

  #define PyString_FromString PyUnicode_FromString
  #define PyString_AsString PyUnicode_AsUTF8

  #ifndef Py_TPFLAGS_HAVE_SEQUENCE_IN
        #define Py_TPFLAGS_HAVE_SEQUENCE_IN 0
  #endif
#endif

#if PY_VERSION_HEX < 0x03020000
  #define SLICE PySliceObject
#else
  #define SLICE PyObject
#endif
```

Example class definition

```
typedef struct Service {
  PyObject_HEAD
  char *name;
  KTL_HANDLE *handle;
  PyObject *keywords;
  PyObject *callback;
  PyObject *registered;
  int notify;
  int check;
  int check_fd;
  int prompt_fd;
} Service;
```

A PyObject * can refer to *any* Python object: numbers, strings, sequence types, or custom classes. Anything. The remaining structure members are all native C constructs.

Example class definition

```
typedef struct Keyword {
  PyObject_HEAD
  Service *service;
  char *name;
  KTL_DATATYPE type;
  PyObject *callback;
} Keyword;
```

Note that we have a stored reference to a Service instance, defined in the previous slide. We could use a PyObject * reference; using the native type means we can access its structure members directly.

Example module method

```
static PyObject *
ktlc_dumpster (PyObject *self, PyObject *args, PyObject *kwargs) {

  static char *kwlist[] = {"append", "close", NULL};
  PyObject *append=NULL;
  PyObject *close=NULL;

  if (!PyArg_ParseTupleAndKeywords (args, kwargs, "OO|", kwlist,
                                    &append, &close)) {
       PyErr_SetString (PyExc_TypeError, "arguments are a function and a close boolean");
       return NULL;
  }
       /* ...and so on. */
```

Note that we have a stored reference to a Service instance, defined in the previous slide. We could use a PyObject * reference; using the native type means we can access its structure members directly.

Example class method

```
static PyObject *
Keyword_convert (Keyword *self, PyObject *args, PyObject *kwargs) {

      /* Keyword.convert() function goes here */
}
```

Note that the self argument is of the native type for the class, not a PyObject *. This allows you to directly access members of the struct without any extra type-checking.

Gotchas

Image credit: US Navy (public domain)

Reference counting

All Python objects have a reference count. When that count drops to zero the object is no longer valid. If you store a PyObject * reference *anywhere* in your Python/C code you have to know whether that reference is **new** or **borrowed**. If it's a borrowed reference you need to manually increment and later decrement the reference count for that object. References can be **stolen** by functions you call. Check the documentation!

Reference counting

```
void
printException (void) {
  PyObject *result = NULL;

  result = PyErr_Occurred ();

  if (result != NULL) {
      /* Print and clear the exception. */
      PyErr_Print ();
  }

  /* No need to invoke Py_DECREF() on 'result', PyErr_Occurred() returns
      a borrowed reference. */
}
```

This code includes a comment noting that PyErr_Occurred() returns a borrowed reference. How do we know that? From the Python/C documentation.

Reference counting

```
message = PyTuple_New (3);

timestamp = PyFloat_FromDouble (current->timestamp);
severity = PyInt_FromLong ((long) current->severity);
text = PyString_FromString (current->message);

PyTuple_SET_ITEM (message, 0, timestamp);
PyTuple_SET_ITEM (message, 1, severity);
PyTuple_SET_ITEM (message, 2, text);

/* Do not invoke Py_DECREF() on the new set elements because PyTuple_SET_ITEM()
 * steals the new reference returned by PyString_FromString() (et al.).
 */
```

This code includes a comment noting that another Python function will steal a reference. How do we know that? From the Python/C documentation.

Reference counting

```
if (self->keywords != NULL) {
    PyDict_Clear (self->keywords);
    Py_DECREF (self->keywords);
    self->keywords = NULL;
}
```

This is part of a destructor, called when an instance gets deallocated. It takes care to properly delete a PyObject * reference, which in this case is a Python dictionary. Note that we don’t call free(), we just decrement the reference count.

Multithreading issues

Python is single-threaded by default. This is enforced through the **Global Interpreter Lock** (GIL). If you need to asynchronously call native Python code from your C code you must explicitly acquire the GIL and release it when you’re done. This should only be a concern if you have external C threads that trigger events.

Multithreading issues

If you know you're going to block for some reason (waiting on network I/O, etc.) you should explicitly release the GIL so that other Python threads can proceed. The Python/C API includes two macros to simplify the process:

```
Py_BEGIN_ALLOW_THREADS
/* Do a blocking thing…*/
Py_END_ALLOW_THREADS
```

Multithreading issues

Signaling a Python thread is best handled with file descriptors. Open a socket pair and let your Python code invoke select() on the outbound descriptor; when your C code receives an asynchronous event that requires attention, stuff something in the inbound descriptor. This avoids the need to acquire the GIL and invoke a Python-native notification function.