Rubicon Documentation

Release 0.2.10

Russell Keith-Magee

Jan 23, 2023

CONTENTS

Rubicon Objective-C is a bridge between Objective-C and Python. It enables you to:

- Use Python to instantiate objects defined in Objective-C,
- Use Python to invoke methods on objects defined in Objective-C, and
- Subclass and extend Objective-C classes in Python.

It also includes wrappers of the some key data types from the Foundation framework (e.g., `NSString`).

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Tutorial

Get started with a hands-on introduction for beginners

1.2 How-to guides

Guides and recipes for common problems and tasks, including how to contribute

1.3 Background

Explanation and discussion of key topics and concepts

1.4 Reference

Technical reference - commands, modules, classes, methods

CHAPTER

TWO

COMMUNITY

Rubicon is part of the BeeWare suite. You can talk to the community through:

- @pybeeware on Twitter
- pybee/general on Gitter

2.1 Tutorials

These tutorials are step-by step guides for using Briefcase.

2.1.1 Your first bridge

In this example, we're going to use Rubicon to access the Objective-C Foundation library, and the *NSURL* class in that library. *NSURL* is the class used to represent and manipulate URLs.

This tutorial assumes you've set up your environment as described in the *Getting started guide*.

Accessing NSURL

Start Python, and get a reference to an Objective-C class. In this example, we're going to use the *NSURL* class, Objective-C's representation of URLs:

```
>>> from rubicon.objc import ObjCClass
>>> NSURL = ObjCClass("NSURL")
```

This gives us an *NSURL* class in Python which is transparently bridged to the *NSURL* class in the Objective-C runtime. Any method or property described in Apple's documentation on NSURL can be accessed over this bridge.

Let's create an instance of an *NSURL* object. The *NSURL* documentation describes a static constructor *+URLWithString:*; we can invoke this constructor as:

```
>>> base = NSURL.URLWithString("http://pybee.org/")
```

That is, the name of the method in Python is identical to the method in Objective-C. The first argument is declared as being an *NSString* *; Rubicon converts the Python *str* into an *NSString* instance as part of invoking the method.

NSURL has another static constructor: *+URLWithString:relativeToURL:*. We can also invoke this constructor:

```
>>> full = NSURL.URLWithString("contributing/", relativeToURL=base)
```

The second argument (*relativeToURL*) is accessed as a keyword argument. This argument is declared as being of type *NSURL **; since *base* is an instance of NSURL, Rubicon can pass through this instance.

Sometimes, an Objective-C method definition will use the same keyword argument name twice. This is legal in Objective-C, but not in Python, as you can't repeat a keyword argument in a method call. In this case, you can use a "long form" of the method to explicitly invoke a descriptor by replacing colons with underscores:

```
>>> base = NSURL.URLWithString_("http://pybee.org/")
>>> full = NSURL.URLWithString_relativeToURL_("contributing", base)
```

Instance methods

So far, we've been using the *+URLWithString:* static constructor. However, *NSURL* also provides an initializer method *-initWithString:*. To use this method, you first have to instruct the Objective-C runtime to allocate memory for the instance, then invoke the initializer:

```
>>> base = NSURL.alloc().initWithString("http://pybee.org/")
```

Now that you have an instance of *NSURL*, you'll want to manipulate it. *NSURL* describes an *absoluteURL* property; this property can be accessed as a Python attribute:

```
>>> absolute = full.absoluteURL
```

You can also invoke methods on the instance:

```
>>> longer = absolute.URLByAppendingPathComponent('how/first-time/')
```

If you want to output an object at the console, you can use the Objective-C property *description*, or for debugging output, *debugDescription*:

```
>>> longer.description
'http://pybee.org/contributing/how/first-time/'

>>> longer.debugDescription
'http://pybee.org/contributing/how/first-time/>'
```

Internally, *description* and *debugDescription* are hooked up to their Python equivalents, *__str__()* and *__repr__()*, respectively:

```
>>> str(absolute)
'http://pybee.org/contributing/how/first-time/'

>>> repr(absolute)
'<rubicon.objc.runtime.ObjCInstance 0x1114a3cf8: NSURL at 0x7fb2abdd0b20: http://pybee.
↪org/contributing/>'

>>> print(absolute)
<rubicon.objc.runtime.ObjCInstance 0x1114a3cf8: NSURL at 0x7fb2abdd0b20: http://pybee.
↪org/contributing/>
```

Time to take over the world!

You now have access to *any* method, on *any* class, in any library, in the entire macOS or iOS ecosystem! If you can invoke something in Objective-C, you can invoke it in Python - all you need to do is:

- load the library with ctypes;
- register the classes you want to use; and
- Use those classes as if they were written in Python.

Next steps

The next step is to write your own classes, and expose them into the Objective-C runtime. That's the subject of the *next tutorial*.

2.1.2 Tutorial 2 - Writing your own class

Eventually, you'll come across an Objective-C API that requires you to provide a class instance as an argument. For example, when using macOS and iOS GUI classes, you often need to define "delegate" classes to describe how a GUI element will respond to mouse clicks and key presses.

Let's define a *Handler* class, with two methods:

- an *-initWithValue:* constructor that accepts an integer; and
- a *-pokeWithValue:andName:* method that accepts an integer and a string, prints the string, and returns a float that is one half of the value.

The declaration for this class would be:

```
from rubicon.objc import NSObject, objc_method


class Handler(NSObject):
    @objc_method
    def initWithValue_(self, v: int):
        self.value = v
        return self

    @objc_method
    def pokeWithValue_andName_(self, v: int, name) -> float:
        print("My name is", name)
        return v / 2.0
```

This code has several interesting implementation details:

- The *Handler* class extends *NSObject*. This instructs Rubicon to construct the class in a way that it can be registered with the Objective-C runtime.
- Each method that we want to expose to Objective-C is decorated with *@objc_method*.The method names match the Objective-C descriptor that you want to expose, but with colons replaced by underscores. This matches the "long form" way of invoking methods discussed in *Your first bridge*.
- The *v* argument on *initWithValue_()* uses a Python 3 type annotation to declare it's type. Objective-C is a language with static typing, so any methods defined in Python must provide this typing information. Any argument that isn't annotated is assumed to be of type *id* - that is, a pointer to an Objective-C object.

- The *pokeWithValue_andName_()* method has it's integer argument annotated, and has it's return type annotated as float. Again, this is to support Objective-C typing operations. Any function that has no return type annotation is assumed to return *id*. A return type annotation of *None* will be interpreted as a *void* method in Objective-C. The *name* argument doesn't need to be annotated because it will be passed in as a string, and strings are *NSObject* subclasses in Objective-C.
- *initWithValue_()* is a constructor, so it returns *self*.

Having declared the class, you can then instantiate and use it:

```
>>> my_handler = Handler.alloc().initWithValue(42)
>>> print(my_handler.value)
42
>>> print(my_handler.pokeWithValue(37, andName="Alice"))
My name is Alice
18.5
```

Objective-C properties

When we defined the initializer for *Handler*, we stored the provided value as the *value* attribute of the class. However, as this attribute wasn't declared to Objective-C, it won't be visible to the Objective-C runtime. You can access *value* from within Python - but Objective-C code won't be able to access it.

To expose value to the Objective-C runtime, we need to make one small change, and explicitly declare value as an Objective-C property:

```
from rubicon.objc import NSObject, objc_method

class PureHandler(NSObject):
    value = obj_property()

    @objc_method
    def initWithValue_(self, v: int):
        self.value = v
        return self
```

This doesn't change anything about how you access or modify the attribute - it just means that Objective-C code will be able to see the attribute as well.

Class naming

In this revised example, you'll note that we also used a different class name - *PureHandler*. This was deliberate, because Objective-C doesn't have any concept of namespaces. As a result, you can only define one class of any given name in a process - so, you wont be able to define a second *Handler* class in the same Python shell. If you try, you'll get an error:

```
>>> class Handler(NSObject):
...     pass
Traceback (most recent call last)
...
RuntimeError: ObjC runtime already contains a registered class named 'Handler'.
```

You'll need to be careful (and sometimes, painfully verbose) when choosing class names.

What, no *__init__()*?

You'll also notice that our example code *doesn't* have an *__init__()* method like you'd normally expect of Python code. As we're defining an Objective-C class, we need to follow the Objective-C object lifecycle - which means defining initializer methods that are visible to the Objective-C runtime, and invoking them over that bridge.

Next steps

???

2.1.3 Tutorial 1 - Your first bridge

In *Your first bridge*, you will use Rubicon to invoke an existing Objective-C library on your computer.

2.1.4 Tutorial 2 - Writing your own class

In *Tutorial 2 - Writing your own class*, you will write a Python class, and expose it to the Objective-C runtime.

2.2 How-to Guides

How-to guides are recipes that take the user through steps in key subjects. They are more advanced than tutorials and assume a lot more about what the user already knows than tutorials do, and unlike documents in the tutorial they can stand alone.

2.2.1 How to get started

To use Rubicon, create a new virtual environment, and install it:

```
$ python3 -m venv venv
$ source venv/bin/activate.sh
(venv) $ pip install rubicon-objc
```

You're now ready to use Rubicon! Your next step is to work through the *Tutorials*, which will take you step-by-step through your first steps and introduce you to the important concepts you need to become familiar with.

2.2.2 You're just not my type: Using Objective-C types in Python

Objective-C is a strong, static typed language. Every variable has a specific type, and that type cannot change over time. If a function declares that it accepts an integer, then it must receive a variable that is declared as an integer, or an expression that results in an integer.

Python, on the other hand, is strong, but *dynamically* typed language. Every variable has a specific type, but that type can be modified or interpreted in other ways. When a function accepts an argument, Python will allow you to pass *any* variable, of *any* type.

So, if you want to bridge between Objective-C and Python, you need to be able to provide static typing information so that Python can work out how to convert a variable of arbitrary type into a specific type matching Objective-C's expectations.

If you're calling an Objective C method defined in a library, this conversion is done automatically - the Objective-C runtime contains enough information for Rubicon to infer the required types. However, if you're defining a new method (or a method override) in Python, we need to provide that typing information. To do this, we use Python 3's type annotation. Here's how.

Primitives

If a Python value needs to be passed in as a primitive, Rubicon will wrap the primitive:

bool 8 bit integer (although it can only hold 2 values - 0 and 1) int 32 bit integer float double precision floating point
===== ==

If a Python value needs to be passed in as an object, Rubicon will wrap the primitive in an object:

Value	Objective C type
bool	NSNumber (bool)
int	NSNumber (long)
float	NSNumber (double)

If you're declaring a method and need to annotate the type of an argument, the Python type name can be used as the annotation type. You can also use any of the *ctypes* primitive types. Rubicon also provides type definitions for common Objective-C typedefs, like *NSInteger*, *CGFloat*, and so on.

Lists

If a method calls for an *NSArray* or *NSMutableArray* argument, you can provide a Python *list* for that argument. Rubicon will construct an *NSMutableArray* instance from the data in the *list* provided, and pass that value for the argument.

If a method returns an *NSArray* or *NSMutableArray*, the return value will be a wrapped *ObjCListInstance* type. This type implements a *list*-like interface, wrapped around the underlying *NSArray* data. This means you can treat the return value as if it were a list - iterating over values, retrieving objects by index, and so on.

Dictionaries

If a method calls for an *NSDictionary* or *NSMutableDictionary* argument, you can provide a Python *dict*. Rubicon will construct an *NSMutableDictionary* instance from the data in the *list* provided, and pass that value for the argument.

If a method returns an *NSDictionary* or *NSMutableDictionary*, the return value will be a wrapped *ObjCDictInstance* type. This type implements a *dict*-like interface, wrapped around the underlying *NSDictionary* data. This means you can treat the return value as if it were a dict - iterating over keys, values or items, retrieving objects by key, and so on.

NSPoint, *NSSize*, and *NSRect*

2.2.3 Using and creating Objective-C protocols

Protocols are used in Objective-C to declare a set of methods and properties for a class to implement. They have a similar purpose to ABCs (abstract base classes) in Python.

Looking up a protocol

Protocol objects can be looked up using the `ObjCProtocol` constructor, similar to how classes can be looked up using `ObjCClass`:

```
>>> NSCopying = ObjCProtocol('NSCopying')
>>> NSCopying
<rubicon.objc.runtime.ObjCProtocol: NSCopying at 0x7fff76543210>
```

The `isinstance` function can be used to check whether an object conforms to a protocol:

```
>>> isinstance(NSObject.new(), NSCopying)
False
>>> isinstance(NSArray.array(), NSCopying)
True
```

Implementing a protocol

When writing a custom Objective-C class, you might want to have it conform to one or multiple protocols. In Rubicon, this is done by using the `protocols` keyword argument in the base class list. For example, if you have a class `UserAccount` and want it to conform to `NSCopyable`, you would write it like this:

```
class UserAccount(NSObject, protocols=[NSCopying]):
    username = objc_property()
    emailAddress = objc_property()

    @objc_method
    def initWithUsername_emailAddress_(self, username, emailAddress):
        self = self.init()
        if self is None:
            return None
        self.username = username
        self.emailAddress = emailAddress
        return self

    # This method is required by NSCopying.
    # The "zone" parameter is obsolete and can be ignored, but must be included for␣
↪backwards compatibility.
    # This method is not normally used directly. Usually you call the copy method␣
↪instead,
    # which calls copyWithZone: internally.
    @objc_method
    def copyWithZone_(self, zone):
        return UserAccount.alloc().initWithUsername(self.username, emailAddress=self.
↪emailAddress)
```

We can now use our class. The `copy` method (which uses our implemented `copyWithZone:` method) can also be used:

```
>>> ua = UserAccount.alloc().initWithUsername_emailAddress_(at('person'), at(
↪'person@example.com'))
>>> ua
<rubicon.objc.runtime.ObjCInstance 0x106543210: UserAccount at 0x106543220:
```

(continues on next page)

(continued from previous page)

```
↪<UserAccount: 0x106543220>>
>>> ua.copy()
<rubicon.objc.runtime.ObjCInstance 0x106543210: UserAccount at 0x106543220:
↪<UserAccount: 0x106543220>>
```

And we can check that the class conforms to the protocol:

```
>>> isinstance(ua, NSCopying)
True
```

Writing custom protocols

You can also create custom protocols. This works similarly to creating custom Objective-C classes:

```
class Named(metaclass=ObjCProtocol):
    name = objc_property()

    @objc_method
    def sayName(self):
        ...
```

There are two notable differences between creating classes and protocols:

1. Protocols do not need to extend exactly one other protocol - they can also extend multiple protocols, or none at all. When not extending other protocols, as is the case here, we need to explicitly add `metaclass=ObjCProtocol` to the base class list, to tell Python that this is a protocol and not a regular Python class. When extending other protocols, Python detects this automatically.
2. Protocol methods do not have a body. Python has no dedicated syntax for functions without a body, so we put `...` in the body instead. (You could technically put code in the body, but this would be misleading and is not recommended.)

2.2.4 Asynchronous Programming with Rubicon

One of the banner features of Python 3 is the introduction of native asychronous programming, implemented in the *asyncio*.

For an introduction to the use of asynchronous programming, see *the documentation for the asyncio module <https://docs.python.org/3/library/asyncio.html>*, or *this introductory tutorial to asynchronous programming with asyncio <http://asyncio.readthedocs.io/en/latest/index.html>*.

Integrating asyncio with CoreFoundation

The asyncio module provides an event loop to coordinate asynchronous features. However, if you're running an Objective C GUI applicaiton, you probably already have an event loop - the one provided by CoreFoundation. This CoreFoundation event loop is then wrapped by *NSApplication* or *UIApplication* in end-user code.

However, you can't have two event loops running at the same time, so you need a way to integrate the two. Luckily, *asyncio* provides a way to customize it's event loop so it can be integrated with other event sources.

It does this using an Event Loop Policy. Rubicon provides an Core Foundation Event Loop Policy that inserts Core Foundation event handling into the asyncio event loop.

To use asyncio in a pure Core Foundation application, do the following:

```
# Import the Event Loop Policy
from rubicon.objc.async import EventLoopPolicy

# Install the event loop policy
asyncio.set_event_loop_policy(EventLoopPolicy())

# Get an event loop, and run it!
loop = asyncio.get_event_loop()
loop.run_forever()
```

The last call (`loop.run_forever()`) will, as the name suggests, run forever - or, at least, until an event handler calls `loop.stop()` to terminate the event loop.

Integrating asyncio with AppKit and NSApplication

If you're using AppKit and NSApplication, you don't just need to start the CoreFoundation event loop - you need to start the full NSApplication lifecycle. To do this, you pass the application instance into the call to `loop.run_forever()`:

```
# Import the Event Loop Policy and lifecycle
from rubicon.objc.async import EventLoopPolicy, CocoaLifecycle

# Install the event loop policy
asyncio.set_event_loop_policy(EventLoopPolicy())

# Get a handle to the shared NSApplication
from ctypes import cdll, util
from rubicon.objc import ObjCClass

appkit = cdll.LoadLibrary(util.find_library('AppKit'))
NSApplication = ObjCClass('NSApplication')
app = NSApplication.sharedApplication()

# Get an event loop, and run it, using the NSApplication!
loop = asyncio.get_event_loop()
loop.run_forever(lifecycle=CocoaLifecycle(app))
```

Again, this will run "forever" – until either `loop.stop()` is called, or `terminate:` is invoked on the NSApplication.

2.2.5 How to contribute to Rubicon

If you experience problems with Rubicon, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

Set up your development environment

The recommended way of setting up your development environment for Rubicon is to install a virtual environment, install the required dependencies and start coding:

```
$ python3 -m venv venv
$ source venv/bin/activate.sh
$ git clone git@github.com:pybee/rubicon-objc.git
$ cd rubicon-objc
$ pip install -e .
```

In order to test the capabilities of Rubicon, the test suite contains an Objective-C library with some known classes. To run the test suite, you'll need to compile this library:

```
$ make
```

This will produce *tests/objc/librubiconharness.dylib*.

In order for Rubicon to find this file, it will need to be on your dynamic library path. You can set this by setting an environment variable:

```
$ export DYLD_LIBRARY_PATH=$(pwd)/tests/objc
```

You can then run the test suite:

```
$ python setup.py test
```

Now you are ready to start hacking on Rubicon. Have fun!

2.3 Background

Want to know more about the Rubicon project, it's history, community, and plans for the future? That's what you'll find here!

2.3.1 Why "Rubicon"?

So... why the name Rubicon?

The Rubicon is a river in Italy. It was of importance in ancient times as the border of Rome. The Roman Army was prohibited from crossing this border, as that would be considered a hostile act against the Roman Senate.

In 54 BC, Julius Caesar marched the Roman Army across the Rubicon, signaling his intention to overthrow the Roman Senate. As he did so, legend says he uttered the words "Alea Iacta Est" - The die is cast. This action led to Julius being crowned as Emperor of Rome, and the start of the Roman Empire.

Of course, in order to cross any river, you need to use a bridge.

This project provides a bridge between the open world of the Python ecosystem, and the walled garden of Apple's Objective-C ecosystem.

2.3.2 The Rubicon Objective-C Developer and User community

Rubicon Objective-C is part of the BeeWare suite. You can talk to the community through:

- @pybeeware on Twitter
- The pybee/general channel on Gitter.

Code of Conduct

The BeeWare community has a strict Code of Conduct. All users and developers are expected to adhere to this code.

If you have any concerns about this code of conduct, or you wish to report a violation of this code, please contact the project founder *Russell Keith- Magee*.

Contributing

If you experience problems with Rubicon, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

2.3.3 Release History

(next version)

- Improved handling of boolean types.
- Added support for using primitives as object values (e.g, as the key/value in an NSDictonary).
- Added support for passing Python lists as Objective-C NSArray arguments, and Python dicts as Objective-C NSDictionary arguments.
- Corrected support to storing strings and other objects as properties on Python-defined Objective-C classes.
- Added support for creating Objective-C blocks from Python callables. (ojii)
- Added support for creating, extending and conforming to Objective-C protocols.
- Added an `objc_const` convenience function to look up global Objective-C object constants in a DLL.
- Added support for registering custom `ObjCInstance` subclasses to be used to represent Objective-C objects of specific classes.

0.2.8

- Added support for using native Python sequence/mapping syntax with `NSArray` and `NSDictionary`. (jeamland)
- Added support for calling Objective-C blocks in Python. (ojii)
- Added functions for declaring custom conversions between Objective-C type encodings and `ctypes` types.
- Added functions for splitting and decoding Objective-C method signature encodings.
- Added automatic conversion of Python sequences to C arrays or structures in method arguments.
- Extended the Objective-C type encoding decoder to support block types, bit fields (in structures), typed object pointers, and arbitrary qualifiers. If unknown pointer, array, struct or union types are encountered, they are created and registered on the fly.
- Changed the `PyObjectEncoding` to match the real definition of `PyObject *`.

- Fixed the declaration of `unichar` (was previously `c_wchar`, is now `c_ushort`).
- Removed the `get_selector` function. Use the `SEL` constructor instead.
- Removed some runtime function declarations that are deprecated or unlikely to be useful.
- Removed the encoding constants. Use `encoding_for_ctype` to get the encoding of a type.

0.2.7

- (#40) Added the ability to explicitly declare no-attribute methods as properties. This is to enable a workaround when Apple introduces readonly properties as a way to access these methods.

0.2.6

- Added a more compact syntax for calling Objective-C methods, using Python keyword arguments. (The old syntax is still fully supported and will *not* be removed; certain method names even require the old syntax.)
- Added a `superclass` property to `ObjCClass`.

0.2.5

- Added official support for Python 3.6.
- Added keyword arguments to disable argument and/or return value conversion when calling an Objective-C method.
- Added support for (`NS`/`UI`) `EdgeInsets` structs. (Longhanks)
- Improved `str` of Objective-C classes and objects to return the `debugDescription`, or for `NSString`s, the string value.
- Changed `ObjCClass` to extend `ObjCInstance` (in addition to `type`), and added an `ObjCMetaClass` class to represent metaclasses.
- Fixed some issues on non-x86_64 architectures (i386, ARM32, ARM64).
- Fixed example code in README. (Dayof)
- Removed the last of the Python 2 compatibility code.

0.2.4

- Added `objc_property` function for adding properties to custom Objective-C subclasses. (Longhanks)

0.2.3

- Removed most Python 2 compatibility code.

0.2.2

- Dropped support for Python 3.3.
- Added conversion of Python `enum.Enum` objects to their underlying values when passed to an Objective-C method.
- Added syntax highlighting to example code in README. (stsievert)
- Fixed the `setup.py` shebang line. (uranusjr)

0.2.1

- Fixed setting of `ObjCClass`/`ObjCInstance` attributes that are not Objective-C properties.

0.2.0

- First beta release.
- Dropped support for Python 2. Python 3 is now required, the minimum tested version is Python 3.3.
- Added error detection when attempting to create an Objective-C class with a name that is already in use.
- Added automatic conversion between Python `decimal.Decimal` and Objective-C `NSDecimal` in method arguments and return values.
- Added PyPy to the list of test platforms.
- When subclassing Objective-C classes, the return and argument types of methods are now specified using Python type annotation syntax and `ctypes` types.
- Improved property support.

0.1.3

- Fixed some issues on ARM64 (iOS 64-bit).

0.1.2

- Fixed `NSString` conversion in a few situations.
- Fixed some issues on iOS and 32-bit platforms.

0.1.1

- Objective-C classes can now be subclassed using Python class syntax, by using an `ObjCClass` as the superclass.
- Removed `ObjCSubclass`, which is made obsolete by the new subclassing syntax.

0.1.0

- Initial alpha release.
- Objective-C classes and instances can be accessed via `ObjCClass` and `ObjCInstance`.
- Methods can be called on classes and instances with Python method call syntax.
- Properties can be read and written with Python attribute syntax.
- Method return and argument types are read automatically from the method type encoding.
- A small number of commonly used structs are supported as return and argument types.
- Python strings are automatically converted to and from `NSString` when passed to or returned from a method.
- Subclasses of Objective-C classes can be created with `ObjCSubclass`.

2.3.4 Rubicon Roadmap

2.4 Reference

This is the technical reference for public APIs provided by Briefcase.