Rubicon Documentation

Release 0.4.9.dev30+gc51937c

Russell Keith-Magee

Apr 25, 2024

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:

- @beeware@fosstodon.org on Mastodon
- Discord
- The Rubicon-ObjC Github Discussions forum

2.1 Tutorials

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

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("https://beeware.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_("https://beeware.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("https://beeware.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
'https://beeware.org/contributing/how/first-time/'

>>> longer.debugDescription
'https://beeware.org/contributing/how/first-time/'
```

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

```
>>> str(absolute)
'https://beeware.org/contributing/'

>>> repr(absolute)
'<ObjCInstance: NSURL at 0x1114a3cf8: https://beeware.org/contributing/>'

>>> print(absolute)
https://beeware.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, objc_property()

class PureHandler(NSObject):
    value = objc_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 won't 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: An Objective-C class named b'Handler' already exists
```

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

To allow a class name to be re-used, you can set the class variable *auto_rename* to `True`. This option enables automatic renaming of the Objective C class if a naming collision is detected:

```
>>> ObjCClass.auto_rename = True
```

This option can also be enabled on a per-class basis by using the `auto_rename` argument in the class declaration:

```
>>> class Handler(NSObject, auto_rename=True):
...     pass
```

If this option is used, the Objective C class name will have a numeric suffix (e.g., *Handler_2*). The Python class name will be unchanged. 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 life cycle - 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 Getting Started with Rubicon

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

```
$ python3 -m venv venv
$ source venv/bin/activate
(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 strongly and statically-typed language. Every variable has a specific type, and that type cannot change over time. Function parameters also have fixed types, and a function will only accept arguments of the correct types.

Python, on the other hand, is a strongly, but *dynamically*-typed language. Every object has a specific type, but all variables can hold objects of any type. When a function accepts an argument, Python will allow you to pass *any* object, 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 Rubicon can work out how to convert a Python object of arbitrary type into a specific type matching Objective-C's expectations.

Type annotations

If you're calling an Objective-C method defined in a library, its types are already known to the Objective-C runtime and Rubicon. However, if you're defining a new method (or a method override) in Python, you need to manually provide its types. This is done using Python 3's type annotation syntax.

Passing and returning Objective-C objects doesn't require any extra work — if you don't annotate a parameter or the return type, Rubicon assumes that it is an Objective-C object. (To define a method that doesn't return anything, you need to add an explicit `-> None` annotation.)

All other parameter and return types (primitives, pointers, structs) need to be annotated to tell Rubicon and Objective-C which type to expect. These annotations can use any of the types defined by Rubicon, such as *NSInteger* or *NSRange*, as well as standard C types from the `ctypes` module, such as `c_byte` or `c_double`.

For example, a method that takes a C `double` and returns a *NSInteger* would be defined and annotated like this:

```
@objc_method
def roundToZero_(self, value: c_double) -> NSInteger:
    return int(value)
```

Rubicon also allows certain Python types to be used in method signatures, and converts them to matching primitive `ctypes` types. For example, Python `int` is treated as `c_int`, and `float` is treated as `c_double`.

See also:

The *rubicon.objc.types* reference documentation lists all C type definitions provided by Rubicon, and provides additional information about how Rubicon converts types.

Type conversions

When you call existing Objective-C methods, Rubicon already knows which type each argument needs to have and what it returns. Based on this type information, Rubicon will automatically convert the passed arguments to the proper Objective-C types, and the return value to an appropriate Python type. This makes explicit type conversions between Python and Objective-C types unnecessary in many cases.

Argument conversion

If an Objective-C method expects a C primitive argument, you can pass an equivalent Python value instead. For example, a Python `int` value can be passed into any integer argument (`int`, `NSInteger`, `uint8_t`, ...), and a Python `float` value can be passed into any floating-point argument (`double`, `CGFloat`, ...).

To pass a C structure as an argument, you would normally need to construct a structure instance by name. This can get somewhat lengthy, especially with nested structures (e. g. `NSRect(NSPoint(1.2, 3.4), NSSize(5.6, 7.8))`). As a shorthand, Rubicon allows passing tuples instead of structure objects (e. g. `((1.2, 3.4), (5.6, 7.8))`) and automatically converts them to the required structure type.

If a parameter expects an Objective-C object, you can also pass certain Python objects, which are automatically converted to their Objective-C counterparts. For example, a Python `str` is converted to an `NSString`, `bytes` to `NSData`, etc. Collections are also supported: `list` and `dict` are converted to `NSArray` and `NSDictionary`, and their elements are converted recursively.

Note: All of these conversions can also be performed manually - see *Manual conversions* for details.

Return value conversion and wrapping

Primitive values returned from methods are converted using the usual `ctypes` conversions, e. g. C integers are converted to Python `int` and floating-point values to Python `float`.

Objective-C objects are automatically returned as *`ObjCInstance`* objects, so you can call methods on them and access their properties. In some cases, Rubicon also provides additional Python methods on Objective-C objects - see *Python-style APIs and methods for Objective-C objects* for details.

Python-style APIs and methods for Objective-C objects

For some standard Foundation classes, such as lists and dictionaries, Rubicon provides additional Python methods to make them behave more like their Python counterparts. This allows using Foundation objects in place of regular Python objects, so that you do not need to convert them manually.

Strings

`NSString` objects behave almost exactly like Python `str` objects - they can be sliced, concatenated, compared, etc. with other Objective-C and Python strings.

```
# Call an Objective-C method that returns a string.
# We're using NSBundle to give us a string version of a path
>>> NSBundle.mainBundle.bundlePath
<ObjCStrInstance: __NSCFString at 0x114a94d68: /Users/brutus/path/to/somewhere>

# Slice the Objective-C string
>>> NSBundle.mainBundle.bundlePath[:14]
<ObjCStrInstance: __NSCFString at 0x114aa80f0: /Users/brutus/>
```

Note: *`ObjCInstance`* objects wrapping a *`NSString`* internally have the class `ObjCStrInstance`, and you will see this name in the `repr()` of *`NSString`* objects. This is an implementation detail - you should not refer to the

`ObjCStrInstance` class explicitly in your code.

If you have an *NSString*, and you need to pass it to a method that does a specific type check for `str`, you can use `str(nsstring)` to convert the *NSString* to `str`:

```
# Convert the Objective-C string to a Python string.
>>> str(NSBundle.mainBundle.bundlePath)
'/Users/rkm/projects/beeware/venv3.6/bin'
```

Conversely, if you have a `str`, and you specifically require a *NSString*, you can use the *at()* function to convert the Python instance to an *NSString*.

```
>>> from rubicon.objc import at
# Create a Python string
>>> py_str = 'hello world'
# Convert to an Objective-C string
>>> at(py_str)
<ObjCStrInstance: __NSCFString at 0x114a94e48: hello world>
```

NSString also supports all the utility methods that are available on `str`, such as `replace` and `split`. When these methods return a string, the implementation may return Python `str` or Objective-C *NSString* instances. If you need to use the return value from these methods, you should always use `str` or *at()* to ensure that you have the right kind of string for your needs.

```
# Is the path comprised of all lowercase letters? (Hint: it isn't)
>>> NSBundle.mainBundle.bundlePath.islower()
False

# Convert string to lower case; use str() to ensure we get a Python string.
>>> str(NSBundle.mainBundle.bundlePath.lower())
'/users/rkm/projects/beeware/venv3.6/bin'
```

Note: *NSString* objects behave slightly differently than Python `str` objects in some cases. For technical reasons, *NSString*s are not hashable in Python, which means they cannot be used as `dict` keys (but they *can* be used as *NSDictionary* keys). *NSString* also handles Unicode code points above `U+FFFF` differently than Python `str`, because the former is based on UTF-16.

Lists

NSArray objects behave like any other Python sequence - they can be indexed, sliced, etc. and standard operations like `len()` and `in` are supported:

```
>>> from rubicon.objc import NSArray
>>> array = NSArray.arrayWithArray(list(range(4)))
>>> array[0]
0
>>> array[1:3]
<ObjCListInstance: _NSArrayI at 0x10b855208: <__NSArrayI 0x7f86f8e61950>(
1,
2
)
```

(continues on next page)

(continued from previous page)

```
>
>>> len(array)
4
>>> 2 in array
True
>>> 5 in array
False
```

Note: *ObjCInstance* objects wrapping a *NSArray* internally have the class `ObjCListInstance` or `ObjCMutableListInstance`, and you will see these names in the `repr()` of *NSArray* objects. This is an implementation detail - you should not refer to the `ObjCListInstance` and `ObjCMutableListInstance` classes explicitly in your code.

NSMutableArray objects additionally support mutating operations, like item and slice assignment:

```
>>> from rubicon.objc import NSMutableArray
>>> mutarray = NSMutableArray.arrayWithArray(list(range(4)))
>>> mutarray[0] = 42
>>> mutarray
<ObjCMutableListInstance: __NSArrayM at 0x10b8558d0: <__NSArrayM 0x7f86fb04d9f0>(
42,
1,
2,
3
)
>
>>> mutarray[1:3] = [9, 8, 7]
>>> mutarray
<ObjCMutableListInstance: __NSArrayM at 0x10b8558d0: <__NSArrayM 0x7f86fb04d9f0>(
42,
9,
8,
7,
3
)
>
```

Sequence methods like `index` and `pop` are also supported:

```
>>> mutarray.index(7)
3
>>> mutarray.pop(3)
7
```

Note: Python objects stored in an *NSArray* are converted to Objective-C objects using the rules described in *Argument conversion*.

Dictionaries

NSDictionary objects behave like any other Python mapping - their items can be accessed and standard operations like `len()` and `in` are supported:

```
>>> from rubicon.objc import NSDictionary
>>> d = objc.NSDictionary.dictionaryWithDictionary({"one": 1, "two": 2})
>>> d["one"]
1
>>> len(d)
>>> 2
>>> "two" in d
True
>>> "five" in d
False
```

Note: *ObjCInstance* objects wrapping a *NSDictionary* internally have the class `ObjCDictInstance` or `ObjCMutableDictInstance`, and you will see these names in the `repr()` of *NSDictionary* objects. This is an implementation detail - you should not refer to the `ObjCDictInstance` and `ObjCMutableDictInstance` classes explicitly in your code.

NSMutableDictionary objects additionally support mutating operations, like item assignment:

```
>>> md = objc.NSMutableDictionary.dictionaryWithDictionary({"one": 1, "two": 2})
>>> md["three"] = 3
>>> md
<ObjCMutableDictInstance: __NSDictionaryM at 0x10b8a7860: {
    one = 1;
    three = 3;
    two = 2;
}>
```

Mapping methods like `keys` and `values` are also supported:

```
>>> d.keys()
<ObjCListInstance: __NSArrayI at 0x10b898a90: <__NSArrayI 0x7f86f8db6b70>(
one,
two
)
>
>>> d.values()
<ObjCListInstance: __NSArrayI at 0x10b8a7b38: <__NSArrayI 0x7f86f8c00370>(
1,
2
)
>
```

Note: Python objects stored in an *NSDictionary* are converted to Objective-C objects using the rules described in *Argument conversion*.

Manual conversions

If necessary, you can also manually call Rubicon's type conversion functions, to convert objects between Python and Objective-C when Rubicon doesn't do so automatically.

Converting from Python to Objective-C

The function *ns_from_py()* (also available as *at()* for short) can convert most standard Python objects to Foundation equivalents. For a full list of possible conversions, see the reference documentation for *ns_from_py()*.

These conversions are performed automatically when a Python object is passed into an Objective-C method parameter that expects an object - in that case you do not need to call *ns_from_py()* manually (see *Argument conversion*).

Converting from Objective-C to Python

The function *py_from_ns()* can convert many common Foundation objects to Python equivalents. For a full list of possible conversions, see the reference documentation for *py_from_ns()*.

These conversions are not performed automatically by Rubicon. For example, if an Objective-C method returns an `NSString`, Rubicon will return it as an *ObjCInstance* (with some additional Python methods - see *Python-style APIs and methods for Objective-C objects*). Using *py_from_ns()*, you can convert the `NSString` to a real Python `str`.

When converting collections, such as `NSArray` or `NSDictionary`, *py_from_ns()* will convert them recursively to a pure Python object. For example, if `nsarray` is an `NSArray` containing `NSString`s, `py_from_ns(nsarray)` will return a `list` of `str`s. In most cases, that is the desired behavior, but you can also avoid this recursive conversion by passing the Foundation collection into a Python collection constructor: for example `list(nsarray)` will return a `list` of `NSString`s.

2.2.3 Memory management for Objective-C instances

Reference counting works differently in Objective-C compared to Python. Python will automatically track where variables are referenced and free memory when the reference count drops to zero whereas Objective-C uses explicit reference counting to manage memory. The methods `retain`, `release` and `autorelease` are used to increase and decrease the reference counts as described in the Apple developer documentation. When enabling automatic reference counting (ARC), the appropriate calls for memory management will be inserted for you at compile-time. However, since Rubicon Objective-C operates at runtime, it cannot make use of ARC.

Reference counting in Rubicon Objective-C

You won't have to manage reference counts in Python, Rubicon Objective-C will do that work for you. It does so by tracking when you gain ownership of an object. This is the case when you create an Objective-C instance using a method whose name begins with `alloc`, `new`, `copy`, or `mutableCopy`. Rubicon Objective-C will then insert a `release` call when the Python variable that corresponds to the Objective-C instance is deallocated.

An exception to this is when you manually `retain` an object. Rubicon Objective-C will not keep track of such retain calls and you will be responsible to insert appropriate `release` calls yourself.

You will also need to pay attention to reference counting in case of **weak references**. In Objective-C, creating a **weak reference** means that the reference count of the object is not incremented and the object will still be deallocated when no strong references remain. Any weak references to the object are then set to `nil`.

Some objects will store references to other objects as a weak reference. Such properties will be declared in the Apple developer documentation as "@property(weak)" or "@property(assign)". This is commonly the case for delegates.

For example, in the code below, the `NSOutlineView` only stores a weak reference to the object which is assigned to its delegate property:

```
from rubicon.objc import NSObject, ObjCClass
from rubicon.objc.runtime import load_library

app_kit = load_library("AppKit")
NSOutlineView = ObjCClass("NSOutlineView")

outline_view = NSOutlineView.alloc().init()
delegate = NSObject.alloc().init()

outline_view.delegate = delegate
```

You will need to keep a reference to the Python variable `delegate` so that the corresponding Objective-C instance does not get deallocated.

Reference cycles in Objective-C

Python has a garbage collector which detects references cycles and frees objects in such cycles if no other references remain. Cyclical references can be useful in a number of cases, for instance to refer to a "parent" of an instance, and Python makes life easier by properly freeing such references. For example:

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.parent = None
        self.children = []


root = TreeNode("/home")

child = TreeNode("/Documents")
child.parent = root

root.children.append(child)

# This will free both root and child on
# the next garbage collection cycle:
del root
del child
```

Similar code in Objective-C will lead to memory leaks. This also holds for Objective-C instances created through Rubicon Objective-C since Python's garbage collector is unable to detect reference cycles on the Objective-C side. If you are writing code which would lead to reference cycles, consider storing objects as weak references instead. The above code would be written as follows when using Objective-C classes:

```
from rubicon.objc import NSObject, NSMutableArray
from rubicon.objc.api import objc_property, objc_method


class TreeNode(NSObject):
    val = objc_property()
```

(continues on next page)

(continued from previous page)

```
    children = objc_property()
    parent = objc_property(weak=True)

    @objc_method
    def initWithValue_(self, val):
        self.val = val
        self.children = NSMutableArray.new()
        return self


root = TreeNode.alloc().initWithValue("/home")

child = TreeNode.alloc().initWithValue("/Documents")
child.parent = root

root.children.addObject(child)

# This will free both root and child:
del root
del child
```

2.2.4 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
<ObjCProtocol: NSCopying>
```

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
<ObjCInstance: UserAccount at 0x106543210: <UserAccount: 0x106543220>>
>>> ua.copy()
<ObjCInstance: UserAccount at 0x106543210: <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.)

2.2.5 Asynchronous Programming with Rubicon

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

For an introduction to the use of asynchronous programming, see the documentation for the asyncio module.

Integrating asyncio with CoreFoundation

The `asyncio` module provides an event loop to coordinate asynchronous features. However, if you're running an Objective C GUI application, 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.eventloop import EventLoopPolicy

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

# Create an event loop, and run it!
loop = asyncio.new_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` life cycle. 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.eventloop 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')
NSApplication.declare_class_property('sharedApplication')
app = NSApplication.sharedApplication

# Create an event loop, and run it, using the NSApplication!
loop = asyncio.new_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.

Integrating asyncio with iOS and UIApplication

If you're using UIKit and UIApplication on iOS, you need to use the iOS life cycle. To do this, you pass an `iOSLifecycle` object into the call to `loop.run_forever()`:

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

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

# Create an event loop, and run it, using the UIApplication!
loop = asyncio.new_event_loop()
loop.run_forever(lifecycle=iOSLifecycle())
```

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

2.2.6 Calling plain C functions from Python

Most Objective-C APIs are exposed through Objective-C classes and methods, but some parts are implemented as plain C functions. You might also want to want to use a pure C library that provides no Objective-C interface at all. Calling C functions is quite different from calling Objective-C methods and requires some additional work, which will be explained in this how-to.

See also:

The ctypes tutorial in the Python documentation, which explains how to call C functions in general (without a specific focus on Apple platforms and Objective-C).

A simple example: `puts`

We'll start with a simple example: calling the `puts` function from the C standard library. `puts` takes a C string and outputs it to standard output — it's the C equivalent of a simple `print` call.

Before we can call the function, we need to look it up first. To do this, we need to find and load the library in which the function is defined. In the case of standard C functions, this is the standard C library, `libc`. Because this library is commonly used, Rubicon already loads it by default and exposes it in Python as *rubicon.objc.runtime.libc*.

```
>>> from rubicon.objc.runtime import libc
>>> libc
<CDLL '/usr/lib/libc.dylib', handle 7fff60d0cb90 at 0x105850b38>
```

Note: For a list of all C libraries that Rubicon loads and exposes by default, see the *C libraries* section of the *rubicon.objc.runtime* reference documentation.

To access a library that is not predefined by Rubicon, you can use the *load_library()* function:

```
>>> from rubicon.objc.runtime import load_library
>>> libm = load_library("m")
>>> libm
<CDLL '/usr/lib/libm.dylib', handle 7fff60d0cb90 at 0x10596be10>
```

C functions are accessed as attributes on their library:

```
>>> libc.puts
<_FuncPtr object at 0x110178f20>
```

However, unlike Objective-C methods, we cannot call C functions right away — we must first declare the function's argument and return types. (Rubicon cannot do this automatically like with Objective-C methods, because plain C doesn't provide the runtime type information necessary for this.) This type information is found in C header files, in this case `stdio.h` (which defines standard C input/output functions, including `puts`).

The exact location of the macOS C headers varies depending on your version of macOS and the developer tools — it is not a fixed path. To open the header directory in the Finder, run the following command in the terminal:

```
$ open "$(xcrun --show-sdk-path)/usr/include"
```

Note: This command requires a version of the macOS developer tools to be installed. If you do not have Xcode or the command-line developer tools installed yet, run this command in the terminal to install the command-line developer tools:

```
$ xcode-select --install
```

Once you have opened the relevant header file in a text editor, you need to search for the declaration of the function you're looking for. In the case of `puts`, it looks like this:

```
int puts(const char *);
```

This means that `puts` returns an `int` and takes a single argument of type `const char *` (a pointer to one or more characters, i.e. a C string). This translates to the following Python `ctypes` code:

```
>>> from ctypes import c_char_p, c_int
>>> libc.puts.restype = c_int
>>> libc.puts.argtypes = [c_char_p]
```

Now that we have provided all of the necessary type information, we can call `libc.puts`.

For the `c_char_p` argument, we pass a byte string with the message we want to print out. `ctypes` automatically converts the byte string object to a `c_char_p` (`char *`) as the C function expects it. The string specifically needs to be a byte string (`bytes`), because C's `char *` strings are byte-based, unlike normal Python strings (`str`), which are Unicode-based.

```
>>> res = libc.puts(b"Hello!")
Hello!
```

Note: If you're running this code from an editor or IDE and don't see `Hello!` printed out, try running the code from a Python REPL in a terminal window instead. Some editors/IDEs, such as Python's IDLE, can only capture and display output produced by high-level Python functions (such as `print`), but not output from low-level C functions (such as `puts`).

The return value of `puts` is ignored in this example. It indicates whether or not the call was successful. If `puts` succeeds, it returns a non-negative integer (the exact value is not significant and has no defined meaning). If `puts` encounters an error, it returns the `EOF` constant (on Apple OSes, this is `-1`).

The `puts` function generally doesn't fail, except for edge cases that are unlikely to happen in practice. With most other C functions, you need to be more careful about checking the return value, to make sure that errors from the function call are detected and handled. Unlike in Python, if you forget to check whether a C function call failed, any errors from that call are silently ignored, which often leads to bad behavior or crashes.

Most real examples of C functions are more complicated than `puts`, but the basic procedure for calling them is the same: import or load the function's C library, set the function's return type and argument types based on the relevant header, and then call the function as needed.

Each C library only needs to be imported/loaded once, and the `restype` and `argtypes` only need to be set once per function. This is usually done at module level near the beginning of the module, similar to Python imports.

Inline functions (e.g. `NSLocationInRange`)

Regular C functions can be called as explained above, but there is also a second kind of C function that needs to be handled differently: inline functions. Unlike regular C functions, inline functions cannot be called through a library object at runtime. Instead, their implementation is only provided as source code in a header file.

When an inline function is called from regular C code, the C compiler copies (inlines) the inline function's implementation into the calling code. To call an inline C function from Python, we need to do the same thing — copy the code from the header into our own code — but in addition we need to translate the C code from the header into equivalent Python/`ctypes` code.

As an example we will use the function `NSLocationInRange` from the Foundation framework. This function checks whether an index lies inside a `NSRange` value. The definition of this function, from the Foundation header `NSRange.h`, looks like this:

```
NS_INLINE BOOL NSLocationInRange(NSUInteger loc, NSRange range) {
    return (!(loc < range.location) && (loc - range.location) < range.length) ? YES : NO;
}
```

In this case, the translation to Python consists (roughly) of the following steps:

1. The outer part of the function definition needs to be translated to Python's `def` syntax. The return type and argument types can be omitted in the Python code — because Python is dynamically typed, these explicit types are not needed.
2. The `YES` and `NO` constants in the `return` expressions need to be replaced with their Python equivalents, `True` and `False`.
3. Some operators in the `return` expression need to be changed: C `!cond` translates to Python `not cond`, C `left && right` becomes `left and right`, and C `cond ? true_val : false_val` becomes `true_val if cond else false_val`.

The translated Python code looks like this:

```
def NSLocationInRange(loc, range):
    return True if (not (loc < range.location) and (loc - range.location) < range.
↪length) else False
```

You can then put this translated function into your Python code and call it in place of the corresponding C inline function.

Note: Python code translated from C like this is sometimes more complicated than necessary and can be simplified. In this case for example, `True if cond else False` can be simplified to just `cond`, `not (x < y)` can be simplified to `x >= y`, and a few redundant parentheses can be removed. A cleaner version of the translated code might look like this:

```
def NSLocationInRange(loc, range):
    return loc >= range.location and loc - range.location < range.length
```

Global variables and constants (e.g. `NSFoundationVersionNumber`)

Some C libraries expose not just functions, but also global variables. An example of this is the Foundation framework, which defines the global variable `NSFoundationVersionNumber` in `<Foundation/NSObjCRuntime.h>`:

```
FOUNDATION_EXPORT double NSFoundationVersionNumber;
```

Like functions, global variables are accessed via the library that they are defined by. The syntax is somewhat different for global variables though - instead of reading them directly as attributes of the library object, you use the `in_dll` method of the variable's *type*. (Every `ctypes` type has an `in_dll` method.)

```
>>> from ctypes import c_double
>>> from rubicon.objc.runtime import Foundation
>>> NSFoundationVersionNumber = c_double.in_dll(Foundation, "NSFoundationVersionNumber")
>>> NSFoundationVersionNumber
c_double(1575.23)
```

Note that `in_dll` doesn't return the variable's value directly - instead it returns a `ctypes` data object that has the variable's type, in this case `c_double`. To access the variable's actual value, you can use the data object's `value` attribute:

```
>>> NSFoundationVersionNumber.value
1575.23
```

Objective-C object constants

A special case of global variables is often found in Objective-C libraries: object constants. These are global Objective-C object variables with a `const` modifier, meaning that they cannot be modified. Constants of type `NSString *` are especially common and can be found in many places, such as Foundation's `<Foundation/NSMetadataAttribute.h>`:

```
FOUNDATION_EXPORT NSString * const NSMetadataItemFSNameKey;
```

Because they are so common, Rubicon provides the convenience function `objc_const` specifically for accessing Objective-C object constants:

```
>>> from rubicon.objc import objc_const
>>> from rubicon.objc.runtime import Foundation
>>> NSMetadataItemFSNameKey = objc_const(Foundation, "NSMetadataItemFSNameKey")
>>> NSMetadataItemFSNameKey
<ObjCStrInstance: __NSCFConstantString at 0x10eecf350: kMDItemFSName>
```

Note: Sometimes it's not obvious that a constant is an Objective-C object, because its actual type is hidden behind a `typedef`. This is common with the "extensible string enum" pattern, where a set of related string constants are defined together. An example can be found in `<Foundation/NSCalendar.h>`:

```
typedef NSString * NSCalendarIdentifier NS_EXTENSIBLE_STRING_ENUM;

FOUNDATION_EXPORT NSCalendarIdentifier const NSCalendarIdentifierGregorian;
FOUNDATION_EXPORT NSCalendarIdentifier const NSCalendarIdentifierBuddhist;
FOUNDATION_EXPORT NSCalendarIdentifier const NSCalendarIdentifierChinese;
// ... many more ...
```

Even though the constants use the type name `NSCalendarIdentifier`, their actual type is still `NSString *`, based on the `typedef` before.

In some cases, constants use a `typedef` from a different header (or even a different library) than the one defining the constants, which can make it even harder to tell that they are actually Objective-C objects.

A complex example: `dispatch_get_main_queue`

As a final example, we'll look at the function `dispatch_get_main_queue` from the `libdispatch` library. This is a very complex function definition, which involves many of the concepts introduced above, as well as heavy use of C pre-processor macros. If you don't have a lot of experience with the C pre-processor, you may want to skip this section.

First, we need to look at the function's definition, which is found in the header `<dispatch/queue.h>`:

```
DISPATCH_INLINE DISPATCH_ALWAYS_INLINE DISPATCH_CONST DISPATCH_NOTHROW
dispatch_queue_main_t
dispatch_get_main_queue(void)
{
    return DISPATCH_GLOBAL_OBJECT(dispatch_queue_main_t, _dispatch_main_q);
}
```

This is an inline function, which you can see based on the fact that it has a function body and the `DISPATCH_INLINE/DISPATCH_ALWAYS_INLINE` attributes. This means that we cannot look it up directly using `ctypes` - instead we have to translate the function body from C to Python.

We can ignore the first line of the function definition - they contain function attributes intended for the compiler, which we don't need. The second and third line indicate the function's signature - it takes no arguments and returns a value of type `dispatch_queue_main_t`.

The body is a little more complex: it uses `DISPATCH_GLOBAL_OBJECT`, which is actually a C macro. Its definition can be found in `<dispatch/object.h>`:

```
#define DISPATCH_GLOBAL_OBJECT(type, object) ((OS_OBJECT_BRIDGE type)&(object))
```

If we substitute the macro's parameters (`type` and `object`) for their real values in our case (`dispatch_queue_main_t` and `_dispatch_main_q`), we get the expression `((OS_OBJECT_BRIDGE dispatch_queue_main_t)&(_dispatch_main_q))`. `OS_OBJECT_BRIDGE` is also a macro, this time from `<os/object.h>`:

```
#define OS_OBJECT_BRIDGE __bridge
```

It expands to `__bridge`, which is an attribute related to Objective-C's automatic reference counting (ARC) feature. In the context of Rubicon, ARC is not relevant (Rubicon performs its own reference management for Objective-C objects), so we can ignore this attribute. This leaves us with the expression `((dispatch_queue_main_t)&(_dispatch_main_q))`, which we can substitute for the macro call in our original function:

```
dispatch_queue_main_t
dispatch_get_main_queue(void)
{
    return (dispatch_queue_main_t)&(_dispatch_main_q));
}
```

With the macro expansion done, we can now see what the function does: it takes a pointer to the global variable `_dispatch_main_q` and casts it to the type `dispatch_queue_main_t`.

First, let's look at the definition of the `_dispatch_main_q` variable, from `<dispatch/queue.h>`:

```
DISPATCH_EXPORT
struct dispatch_queue_s _dispatch_main_q;
```

The variable's type, `struct dispatch_queue_s`, is an *opaque* structure type - it is not defined in any public header. This means that we don't know what fields the structure has, or even how large it is. As a result, we cannot perform any operations on the structure itself, but we can work with *pointers* to the structure - which is exactly what `dispatch_get_main_queue` does.

Even though `struct dispatch_queue_s` is opaque, we still need to define it in Python so that we can look up the `_dispatch_main_q` variable:

```
from ctypes import Structure
from rubicon.objc.runtime import load_library

# On Mac, libdispatch is part of libSystem.
libSystem = load_library("System")
libdispatch = libSystem

class struct_dispatch_queue_s(Structure):
    pass # No _fields_, because this is an opaque structure.

_dispatch_main_q = struct_dispatch_queue_s.in_dll(libdispatch, "_dispatch_main_q")
```

Now we need to look at the definition of the `dispatch_queue_main_t` type. This definition is not very obvious to find - it's actually this line in <dispatch/queue.h>:

```
DISPATCH_DECL_SUBCLASS(dispatch_queue_main, dispatch_queue_serial);
```

`DISPATCH_DECL_SUBCLASS` is a macro from <dispatch/object.h>, defined like this:

```
#define DISPATCH_DECL_SUBCLASS(name, base) OS_OBJECT_DECL_SUBCLASS(name, base)
```

It directly calls another macro, `OS_OBJECT_DECL_SUBCLASS`, defined in <os/object.h>:

```
#define OS_OBJECT_DECL_SUBCLASS(name, super) \
        OS_OBJECT_DECL_IMPL(name, <OS_OBJECT_CLASS(super)>)
```

Let's substitute this macro into our original code:

```
OS_OBJECT_DECL_IMPL(dispatch_queue_main, <OS_OBJECT_CLASS(dispatch_queue_serial)>);
```

Next is the `OS_OBJECT_DECL_IMPL` macro, also defined in <os/object.h>:

```
#define OS_OBJECT_DECL_IMPL(name, ...) \
        OS_OBJECT_DECL_PROTOCOL(name, __VA_ARGS__) \
        typedef NSObject<OS_OBJECT_CLASS(name)> \
                * OS_OBJC_INDEPENDENT_CLASS name##_t
```

After we substitute this macro into our code, it looks like this:

```
OS_OBJECT_DECL_PROTOCOL(dispatch_queue_main, <OS_OBJECT_CLASS(dispatch_queue_serial)>) \
typedef NSObject<OS_OBJECT_CLASS(dispatch_queue_main)> \
    * OS_OBJC_INDEPENDENT_CLASS dispatch_queue_main_t;
```

And another macro, `OS_OBJECT_DECL_PROTOCOL`, also from <os/object.h>:

```
#define OS_OBJECT_DECL_PROTOCOL(name, ...) \
        @protocol OS_OBJECT_CLASS(name) __VA_ARGS__ \
        @end
```

Which we can substitute into our code:

```
@protocol OS_OBJECT_CLASS(dispatch_queue_main) <OS_OBJECT_CLASS(dispatch_queue_serial)> \
@end \
typedef NSObject<OS_OBJECT_CLASS(dispatch_queue_main)> \
    * OS_OBJC_INDEPENDENT_CLASS dispatch_queue_main_t;
```

Now let's take care of the `OS_OBJECT_CLASS` macro, defined like this in <os/object.h>:

```
#define OS_OBJECT_CLASS(name) OS_##name
```

And substituted into our code:

```
@protocol OS_dispatch_queue_main <OS_dispatch_queue_serial> \
@end \
typedef NSObject<OS_dispatch_queue_main> \
    * OS_OBJC_INDEPENDENT_CLASS dispatch_queue_main_t;
```

Finally we're left with the `OS_OBJECT_INDEPENDENT_CLASS` macro, which is a compiler attribute that we can ignore.

```
@protocol OS_dispatch_queue_main <OS_dispatch_queue_serial>
@end
typedef NSObject<OS_dispatch_queue_main> * dispatch_queue_main_t;
```

Now we're done with macro expansion and can see what the code actually does - it defines an Objective-C protocol called `OS_dispatch_queue_main` and defines `dispatch_queue_main_t` as a pointer type to an object conforming to that protocol. For our purposes, most of these details don't matter - the important part is that `dispatch_queue_main_t` is actually an Objective-C object pointer type. Because Rubicon doesn't differentiate between object pointer types, we can replace `dispatch_queue_main_t` in our original function with the generic `id` type:

```
id
dispatch_get_main_queue(void)
{
    return (id)&(_dispatch_main_q));
}
```

This code can finally be translated to Python:

```
from ctypes import byref, cast
from rubicon.objc import ObjCInstance
from rubicon.objc.runtime import objc_id

# This requires the _dispatch_main_q Python code from before.

def dispatch_get_main_queue():
    return ObjCInstance(cast(byref(_dispatch_main_q), objc_id))
```

Further information

- cdecl.org: An online service to translate C type syntax into more understandable English.
- cppreference.com: A reference site about the standard C and C++ languages and libraries.
- Apple's reference documentation: Official API documentation for Apple platforms. Make sure to change the language to Objective-C in the top-right corner, otherwise you'll get Swift documentation, which can differ significantly from Objective-C.
- macOS man pages, sections 2 and 3: Documentation for the C functions provided by macOS. View these using the `man` command, or by typing a function name into the search box of the macOS Terminal's Help menu.

2.2.7 How to contribute code 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 clone the repository, create a virtual environment, and install the required dependencies:

```
$ git clone https://github.com/beeware/rubicon-objc.git
$ cd rubicon-objc
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ python3 -m pip install -Ue ".[dev]"
```

Rubicon uses a tool called Pre-Commit to identify simple issues and standardize code formatting. It does this by installing a git hook that automatically runs a series of code linters prior to finalizing any git commit. To enable pre-commit, run:

```
(venv) $ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

When you commit any change, pre-commit will run automatically. If there are any issues found with the commit, this will cause your commit to fail. Where possible, pre-commit will make the changes needed to correct the problems it has found:

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Failed
- hook id: black
- files were modified by this hook

reformatted some/interesting_file.py

All done!
1 file reformatted.

flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
fix end of files.........................................................Passed
trim trailing whitespace.................................................Passed
isort....................................................................Passed
pyupgrade................................................................Passed
docformatter.............................................................Passed
```

You can then re-add any files that were modified as a result of the pre-commit checks, and re-commit the change.

```
(venv) $ git add some/interesting_file.py
(venv) $ git commit -m "Minor change"
black....................................................................Passed
flake8...................................................................Passed
check toml.....................................(no files to check)Skipped
check yaml.....................................(no files to check)Skipped
check for case conflicts.................................................Passed
check docstring is first.................................................Passed
```

(continues on next page)

(continued from previous page)

```
fix end of files.....................................................Passed
trim trailing whitespace.............................................Passed
isort................................................................Passed
pyupgrade............................................................Passed
docformatter.........................................................Passed
[bugfix e3e0f73] Minor change
1 file changed, 4 insertions(+), 2 deletions(-)
```

Rubicon uses tox to manage the testing process. To set up a testing environment and run the full test suite, run:

```
(venv) $ tox
```

By default this will run the test suite multiple times, once on each Python version supported by Rubicon, as well as running some pre-commit checks of code style and validity. This can take a while, so if you want to speed up the process while developing, you can run the tests on one Python version only:

```
(venv) $ tox -e py
```

Or, to run using a specific version of Python:

```
(venv) $ tox -e py310
```

substituting the version number that you want to target. You can also specify one of the pre-commit checks *flake8*, *docs* or *package* to check code formatting, documentation syntax and packaging metadata, respectively.

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

2.2.8 Contributing to the documentation

Here are some tips for working on this documentation. You're welcome to add more and help us out!

First of all, you should check the reStructuredText (reST) Primer to learn how to write your `.rst` file.

Create a `.rst` file

Look at the structure and choose the best category to put your `.rst` file. Make sure that it is referenced in the index of the corresponding category, so it will show on in the documentation. If you have no idea how to do this, study the other index files for clues.

Build documentation locally

To build the documentation locally, *set up a development environment*.

You'll also need to install the Enchant spell checking library. Enchant can be installed using Homebrew:

```
(venv) $ brew install enchant
```

If you're on an M1 machine, you'll also need to manually set the location of the Enchant library:

```
(venv) $ export PYENCHANT_LIBRARY_PATH=/opt/homebrew/lib/libenchant-2.2.dylib
```

Once your development environment is set up, run:

```
(venv) $ tox -e docs
```

The output of the file should be in the `docs/_build/html` folder. If there are any markup problems, they'll raise an error.

Documentation linting

Before committing and pushing documentation updates, run linting for the documentation:

macOS

Linux

Windows

```
(venv) $ tox -e docs-lint
```

```
(venv) $ tox -e docs-lint
```

```
C:\...>tox -e docs-lint
```

This will validate the documentation does not contain:

- invalid syntax and markup
- dead hyperlinks
- misspelled words

If a valid spelling of a word is identified as misspelled, then add the word to the list in `docs/spelling_wordlist`. This will add the word to the spellchecker's

Rebuilding all documentation

To force a rebuild for all of the documentation:

macOS

Linux

Windows

```
(venv) $ tox -e docs-all
```

```
(venv) $ tox -e docs-all
```

```
C:\...>tox -e docs-all
```

The documentation should be fully rebuilt in the `docs/_build/html` folder. If there are any markup problems, they'll raise an error.

Live documentation preview

To support rapid editing of documentation, Rubicon also has a "live preview" mode:

macOS

Linux

Windows

```
(venv) $ tox -e docs-live
```

```
(venv) $ tox -e docs-live
```

```
(venv) C:\...>tox -e docs-live
```

This will build the documentation, start a web server to serve the build documentation, and watch the file system for any changes to the documentation source. If a change is detected, the documentation will be rebuilt, and any browser viewing the modified page will be automatically refreshed.

Live preview mode will only monitor the `docs` directory for changes. If you're updating the inline documentation associated with Toga source code, you'll need to use the `docs-live-src` target to build docs:

macOS

Linux

Windows

```
(venv) $ tox -e docs-live-src
```

```
(venv) $ tox -e docs-live-src
```

```
(venv) C:\...>tox -e docs-live-src
```

This behaves the same as `docs-live`, but will also monitor any changes to the `src/rubicon/objc` folder, reflecting any changes to inline documentation. However, the rebuild process takes much longer, so you may not want to use this target unless you're actively editing inline documentation.

2.2.9 Internal How-to guides

These guides are for the maintainers of the Rubicon-ObjC project, documenting internal project procedures.

How to cut a Rubicon-ObjC release

The release infrastructure for Rubicon is semi-automated, using GitHub Actions to formally publish releases.

This guide assumes that you have an `upstream` remote configured on your local clone of the Rubicon repository, pointing at the official repository. If all you have is a checkout of a personal fork of the Rubicon-ObjC repository, you can configure that checkout by running:

```
$ git remote add upstream https://github.com/beeware/rubicon-objc.git
```

The procedure for cutting a new release is as follows:

1. Check the contents of the upstream repository's main branch:

```
$ git fetch upstream
$ git checkout --detach upstream/main
```

Check that the HEAD of release now matches upstream/main.

2. Ensure that the release notes are up to date. Run:

```
$ tox -e towncrier -- --draft
```

to review the release notes that will be included, and then:

```
$ tox -e towncrier
```

to generate the updated release notes.

3. Tag the release, and push the tag upstream:

```
$ git tag v1.2.3
$ git push upstream HEAD:main
$ git push upstream v1.2.3
```

4. Pushing the tag will start a workflow to create a draft release on GitHub. You can follow the progress of the workflow on GitHub; once the workflow completes, there should be a new draft release, and an entry on the Test PyPI server.

Confirm that this action successfully completes. If it fails, there's a couple of possible causes:

a. The final upload to Test PyPI failed. Test PyPI is not have the same service monitoring as PyPI-proper, so it sometimes has problems. However, it's also not critical to the release process; if this step fails, you can perform Step 6 by manually downloading the "packages" artifact from the GitHub workflow instead.

b. Something else fails in the build process. If the problem can be fixed without a code change to the Rubicon-ObjC repository (e.g., a transient problem with build machines not being available), you can re-run the action that failed through the GitHub Actions GUI. If the fix requires a code change, delete the old tag, make the code change, and re-tag the release.

5. Create a clean virtual environment, install the new release from Test PyPI, and perform any pre-release testing that may be appropriate:

```
$ python3 -m venv testvenv
$ . ./testvenv/bin/activate
(testvenv) $ pip install --extra-index-url https://test.pypi.org/simple/ rubicon-
↪objc==1.2.3
(testvenv) $ python -c "from rubicon.objc import __version__; print(__version__)"
1.2.3
(testvenv) $ #... any other manual checks you want to perform ...
```

6. Log into ReadTheDocs, visit the Versions tab, and activate the new version. Ensure that the build completes; if there's a problem, you may need to correct the build configuration, roll back and re-tag the release.

7. Edit the GitHub release to add release notes. You can use the text generated by Towncrier, but you'll need to update the format to Markdown, rather than ReST. If necessary, check the pre-release checkbox.

8. Double check everything, then click Publish. This will trigger a publication workflow on GitHub.

9. Wait for the package to appear on PyPI.

Congratulations, you've just published a release!

If anything went wrong during steps 3 or 5, you will need to delete the draft release from GitHub, and push an updated tag. Once the release has successfully appeared on PyPI, it cannot be changed; if you spot a problem in a published package, you'll need to tag a completely new release.

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:

- @beeware@fosstodon.org
- Discord

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 Success Stories

Want to see examples of Rubicon in use? Here's some:

- Travel Tips is an app in the iOS App Store that uses Rubicon to access the iOS UIKit libraries.

2.3.4 Release History

2.3.5 0.4.8 (2024-04-03)

Features

- Name clashes caused by re-registering Objective C classes and protocols can now be automatically avoided by marking the class with `auto_rename`. (#181)
- Apple Silicon is now formally tested by Rubicon's continuous integration configuration. (#374)
- Support for Python 3.13 was added. (#374)
- The `__repr__` output for `ObjCBoundMethod`, `ObjCClass`, `ObjCInstance`, `ObjCMethod`, `ObjCPartialMethod`, and `ObjCProtocol` were simplified. (#432)

Bugfixes

- The `__all__` definition for `rubicon.objc` was corrected to use strings, rather than symbols. (#401)

Documentation

- The documentation contribution guide was updated to use a more authoritative reStructuredText reference. (#427)

Misc

- #381, #382, #383, #384, #385, #386, #387, #388, #389, #390, #391, #392, #393, #395, #396, #397, #398, #399, #400, #402, #403, #404, #405, #407, #408, #409, #410, #411, #412, #413, #414, #415, #416, #417, #418, #420, #421, #422, #423, #424, #425, #426, #429, #430, #431, #433, #434, #435, #437, #438

2.3.6 0.4.7 (2023-10-19)

Features

- The `__repr__` and `__str__` implementations for `NSPoint`, `CGPoint`, `NSRect`, `CGRect`, `NSSize`, `CGSize`, `NSRange`, `CFRange`, `NSEdgeInsets` and `UIEdgeInsets` have been improved. (#222)
- `objc_id` and `objc_block` are now exposed as part of the `rubicon.objc` namespace, rather than requiring an import from `rubicon.objc.runtime`. (#357)

Bugfixes

- References to blocks obtained from an Objective C API can now be invoked on M1 hardware. (#225)
- Rubicon is now compatible with PEP563 deferred annotations (`from __future__ import annotations`). (#308)
- iOS now uses a full `NSRunLoop`, rather than a `CFRunLoop`. (#317)

Backward Incompatible Changes

- Support for Python 3.7 was dropped. (#334)

Documentation

- All code blocks were updated to add a button to copy the relevant contents on to the user's clipboard. (#300)

Misc

- #295, #296, #297, #298, #299, #301, #302, #303, #305, #306, #307, #310, #311, #312, #314, #315, #319, #320, #321, #326, #327, #328, #329, #330, #331, #332, #335, #336, #337, #338, #341, #342, #343, #344, #345, #346, #348, #349, #350, #351, #353, #354, #355, #356, #358, #359, #360, #361, #362, #363, #364, #365, #366, #367, #368, #369, #370, #371, #372, #373, #375, #376, #377, #378, #379, #380

2.3.7 0.4.6 (2023-04-14)

Bugfixes

- The error message returned when a selector has the wrong type has been improved. (#271)
- Rubicon now uses an implicit namespace package, instead of relying on the deprecated `pkg_resources` API. (#292)

Misc

- #267, #268, #269, #270, #273, #274, #275, #276, #277, #278, #279, #280, #281, #282, #283, #284, #285, #286, #287, #288, #289, #290, #291, #294

2.3.8 0.4.5 (2023-02-03)

Bugfixes

- Classes that undergo a class name change between `alloc()` and `init()` (e.g., `NSWindow` becomes `NSKVONotifying_Window`) no longer trigger instance cache eviction logic. (#258)

Misc

- #259, #260, #262, #263, #264, #265, #266

2.3.9 0.4.5rc1 (2023-01-25)

Features

- Support for Python 3.6 was dropped. (#255)

Misc

- #254

2.3.10 0.4.4 (2023-01-23)

This version was yanked from PyPI because of an incompatibility with Toga-iOS 0.3.0dev39, which was the published Toga release at the time.

Bugfixes

- Background threads will no longer lock up on iOS when an asyncio event loop is in use. (#228)
- The `ObjCInstance` cache no longer returns a stale wrapper objects if a memory address is re-used by the Objective C runtime. (#249)
- It is now safe to open an asyncio event loop on a secondary thread. Previously this would work, but would intermittently fail with a segfault when then loop was closed. (#250)
- A potential race condition that would lead to duplicated creation on `ObjCInstance` wrapper objects has been resolved. (#251)
- A race condition associated with populating the `ObjCClass` method/property cache has been resolved. (#252)

Misc

- #225, #237, #240, #241, #242, #243, #244, #245, #247, #248, #253

2.3.11 0.4.3 (2022-12-05)

Features

- Support for Python 3.11 has been added. (#224)
- Support for Python 3.12 has been added. (#231)

Bugfixes

- Enforce usage of *argtypes* when calling *send_super*. (#220)
- The check identifying the architecture on which Rubicon is running has been corrected for x86_64 simulators using a recent Python-Apple-support releases. (#235)

Misc

- #227, #228, #229, #232, #233, #234

0.4.2 (2021-11-14)

Features

- Added `autoreleasepool` context manager to mimic Objective-C `@autoreleasepool` blocks. (#213)
- Allow storing Python objects in Objective-C properties declared with `@objc_property`. (#214)
- Added support for Python 3.10. (#218)

Bugfixes

- Raise `TypeError` when trying to declare a weak property of a non-object type. (#215)
- Corrected handling of methods when a class overrides a method defined in a grandparent. (#216)

0.4.1 (2021-07-25)

Features

- Added official support for Python 3.9. (#193)
- Added official support for macOS 11 (Big Sur). (#195)
- Autorelease Objective-C instances when the corresponding Python instance is destroyed. (#200)
- Improved memory management when a Python instance is assigned to a new `ObjCInstance` attribute. (#209)
- Added support to declare weak properties on custom Objective-C classes. (#210)

Bugfixes

- Fixed incorrect behavior of *`Block`* when trying to create a block with no arguments and using explicit types. This previously caused an incorrect exception about missing argument types; now a `no-arg` block is created as expected. (#153)
- Fixed handling of type annotations when passing a bound Python method into *`Block`*. (#153)
- A cooperative entry point for starting event loop has been added. This corrects a problem seen when using Python 3.8 on iOS. (#182)
- Improved performance of Objective-C method calls and *`ObjCInstance`* creation in many cases. (#183)
- Fix calling of signal handlers added to the asyncio loop with `CFRunLoop` integration. (#202)

- Allow restarting a stopped event loop. (#205)

Deprecations and Removals

- Removed automatic conversion of Objective-C numbers (`NSNumber` and `NSDecimalNumber`) to Python numbers when received from Objective-C (i.e. returned from an Objective-C method or property or passed into an Objective-C method implemented in Python). This automatic conversion significantly slowed down every Objective-C method call that returns an object, even though the conversion doesn't apply to most method calls. If you have code that receives an Objective-C number and needs to use it as a Python number, please convert it explicitly using *py_from_ns()* or an appropriate Objective-C method.

  As a side effect, `NSNumber` and `NSDecimalNumber` values stored in Objective-C collections (`NSArray`, `NSDictionary`) are also no longer automatically unwrapped when retrieved from the collection, even when using Python syntax to access the collection. For example, if `arr` is a `NSArray` of integer `NSNumber`, `arr[0]` now returns an Objective-C `NSNumber` and not a Python `int` as before. If you need the contents of an Objective-C collection as Python values, you can use *py_from_ns()* to convert either single values (e. g. `py_from_ns(arr[0])`) or the entire collection (e. g. `py_from_ns(arr)`). (#183)

- Removed macOS 10.12 through 10.14 from our automatic test matrix, due to pricing changes in one of our CI services (Travis CI). OS X 10.11 is still included in the test matrix for now, but will probably be removed relatively soon. Automatic tests on macOS 10.15 and 11.0 are unaffected as they run on a different CI service (GitHub Actions).

  Rubicon will continue to support macOS 10.14 and earlier on a best-effort basis, even though compatibility is no longer tested automatically. If you encounter any bugs or other problems with Rubicon on these older macOS versions, please report them! (#197)

Misc

- #185, #189, #194, #196, #208

0.4.0 (2020-07-04)

Features

- Added macOS 10.15 (Catalina) to the test matrix. (#145)
- Added **PEP 517** and **PEP 518** build system metadata to `pyproject.toml`. (#156)
- Added official support for Python 3.8. (#162)
- Added a `varargs` keyword argument to *send_message()* to allow calling variadic methods more safely. (#174)
- Changed `ObjCMethod` to call methods using *send_message()* instead of calling *IMP*s directly. This is mainly an internal change and should not affect most existing code, although it may improve compatibility with Objective-C code that makes heavy use of runtime reflection and method manipulation (such as swizzling). (#177)

Bugfixes

- Fixed Objective-C method calls in "flat" syntax accepting more arguments than the method has. The extra arguments were previously silently ignored. An exception is now raised if too many arguments are passed. (#123)
- Fixed *ObjCInstance.__str__* throwing an exception if the object's Objective-C `description` is `nil`. (#125)
- Corrected a slow memory leak caused every time an asyncio timed event handler triggered. (#146)
- Fixed various minor issues in the build and packaging metadata. (#156)
- Removed unit test that attempted to pass a struct with bit fields into a C function by value. Although this has worked in the past on x86 and x86_64, `ctypes` never officially supported this, and started generating an error in Python 3.7.6 and 3.8.1 (see bpo-39295). (#157)
- Corrected the invocation of `NSApplication.terminate()` when the *CocoaLifecycle* is ended. (#170)
- Fixed *send_message()* not accepting *SEL* objects for the `selector` parameter. The documentation stated that this is allowed, but actually doing so caused a type error. (#177)

Improved Documentation

- Added detailed *reference documentation* for all public APIs of *rubicon.objc*. (#118)
- Added a *how-to guide for calling regular C functions* using `ctypes` and *rubicon.objc*. (#147)

Deprecations and Removals

- Removed the i386 architecture from the test matrix. It is still supported on a best-effort basis, but compatibility is not tested automatically. (#139)
- Tightened the API of *send_message()*, removing some previously allowed shortcuts and features that were rarely used, or likely to be used by accident in an unsafe way.

  Note: In most cases, Rubicon's high-level method call syntax provided by *ObjCInstance* can be used instead of *send_message()*. This syntax is almost always more convenient to use, more readable and less error-prone. *send_message()* should only be used in cases not supported by the high-level syntax.

- Disallowed passing class names as `str`/`bytes` as the `receiver` argument of *send_message()*. If you need to send a message to a class object (i. e. call a class method), use *ObjCClass* or *get_class()* to look up the class, and pass the resulting *ObjCClass* or *Class* object as the receiver.
- Disallowed passing `c_void_p` objects as the `receiver` argument of *send_message()*. The `receiver` argument now has to be of type *objc_id*, or one of its subclasses (such as *Class*), or one of its high-level equivalents (such as *ObjCInstance*). All Objective-C objects returned by Rubicon's high-level and low-level APIs have one of these types. If you need to send a message to an object pointer stored as `c_void_p`, `cast()` it to *objc_id* first.
- Removed default values for *send_message()*'s `restype` and `argtypes` keyword arguments. Every *send_message()* call now needs to have its return and argument types set explicitly. This ensures that all arguments and the return value are converted correctly between (Objective-)C and Python.
- Disallowed passing more argument values than there are argument types in `argtypes`. This was previously allowed to support calling variadic methods - any arguments beyond the types set in `argtypes` would be passed as `varargs`. However, this feature was easy to misuse by accident, as it allowed passing extra arguments to *any*

method, even though most Objective-C methods are not variadic. Extra arguments passed this way were silently ignored without causing an error or a crash.

To prevent accidentally passing too many arguments like this, the number of arguments now has to exactly match the number of `argtypes`. Variadic methods can still be called, but the `varargs` now need to be passed as a list into the separate `varargs` keyword argument. (#174)

- Removed the `rubicon.objc.core_foundation` module. This was an internal module with few remaining contents and should not have any external uses. If you need to call Core Foundation functions in your code, please load the framework yourself using `load_library('CoreFoundation')` and define the types and functions that you need. (#175)
- Removed the `ObjCMethod` class from the public API, as there was no good way to use it from external code. (#177)

Misc

- #143, #145, #155, #158, #159, #164, #173, #178, #179

0.3.1

- Added a workaround for bpo-36880, which caused a "deallocating None" crash when returning structs from methods very often.
- Added macOS High Sierra (10.13) and macOS Mojave (10.14) to the test matrix.
- Renamed the `rubicon.objc.async` module to `rubicon.objc.eventloop` to avoid conflicts with the Python 3.6 `async` keyword.
- Removed support for Python 3.4.
- Removed OS X Yosemite (10.10) from the test matrix. This version is (and older ones are) still supported on a best-effort basis, but compatibility is not tested automatically.

0.3.0

- Added Pythonic operators and methods on `NSString` objects, similar to those for `NSArray` and `NSDictionary`.
- Removed automatic conversion of `NSString` objects to `str` when returned from Objective-C methods. This feature made it difficult to call Objective-C methods on `NSString` objects, because there was no easy way to prevent the automatic conversion.

  In most cases, this change will not affect existing code, because `NSString` objects now support operations similar to `str`. If an actual `str` object is required, the `NSString` object can be wrapped in a `str` call to convert it.
- Added support for `objc_property`s with non-object types.
- Added public `get_ivar` and `set_ivar` functions for manipulating `ivars`.
- Changed the implementation of `objc_property` to use `ivars` instead of Python attributes for storage. This fixes name conflicts in some situations.
- Added the *load_library()* function for loading CDLLs by their name instead of their full path.
- Split the high-level Rubicon API (*ObjCInstance*, *ObjCClass*, etc.) out of *rubicon.objc.runtime* into a separate *rubicon.objc.api* module. The *runtime* module now only contains low-level runtime interfaces like *libobjc*.

This is mostly an internal change, existing code will not be affected unless it imports names directly from *rubicon.objc.runtime*.

- Moved *c_ptrdiff_t* from *rubicon.objc.runtime* to *rubicon.objc.types*.
- Removed some rarely used names (*IMP*, *Class*, *Ivar*, *Method*, *get_ivar()*, *objc_id*, *objc_property_t*, *set_ivar()*) from the main *rubicon.objc* namespace.

  If needed, these names can be imported explicitly from the *rubicon.objc.runtime* module.
- Fixed `objc_property` setters on non-macOS platforms. (cculianu)
- Fixed various bugs in the collection `ObjCInstance` subclasses:
- Fixed getting/setting/deleting items or slices with indices lower than `-len(obj)`. Previously this crashed Python, now an `IndexError` is raised.
- Fixed slices with step size 0. Previously they were ignored and 1 was incorrectly used as the step size, now an `IndexError` is raised.
- Fixed equality checks between Objective-C arrays/dictionaries and non-sequence/mapping objects. Previously this incorrectly raised a `TypeError`, now it returns `False`.
- Fixed equality checks between Objective-C arrays and sequences of different lengths. Previously this incorrectly returned `True` if the shorter sequence was a prefix of the longer one, now `False` is returned.
- Fixed calling `popitem` on an empty Objective-C dictionary. Previously this crashed Python, now a `KeyError` is raised.
- Fixed calling `update` with both a mapping and keyword arguments on an Objective-C dictionary. Previously the kwargs were incorrectly ignored if a mapping was given, now both are respected.
- Fixed calling methods using `kwarg` syntax if a superclass and subclass define methods with the same prefix, but different names. For example, if a superclass had a method `initWithFoo:bar:` and the subclass `initWithFoo:spam:`, the former could not be called on instances of the subclass.
- Fixed the internal `ctypes_patch` module so it no longer depends on a non-public CPython function.

0.2.10

- Rewrote almost all Core Foundation-based functions to use Foundation instead.
  - The functions `from_value` and `NSDecimalNumber.from_decimal` have been removed and replaced by `ns_from_py`.
  - The function `at` is now an alias for `ns_from_py`.
  - The function `is_str` has been removed. `is_str(obj)` calls should be replaced with `isinstance(obj, NSString)`.
  - The functions `to_list`, `to_number`, `to_set`, `to_str`, and `to_value` have been removed and replaced by `py_from_ns`.
- Fixed `declare_property` not applying to subclasses of the class it was called on.
- Fixed `repr` of `ObjCBoundMethod` when the wrapped method is not an `ObjCMethod`.
- Fixed the encodings of `NSPoint`, `NSSize`, and `NSRect` on 32-bit systems.
- Renamed the `async` support package to `eventloop` to avoid a Python 3.5+ keyword clash.

0.2.9

- 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 dictionaries 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 returning compound values (structures and unions) from Objective-C methods defined in Python.
- 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.
- Added support for integrating `NSApplication` and `UIApplication` event loops with Python's asyncio event loop.

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 read-only 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`.

2.3.12 Road map

2.4 Reference

2.4.1 rubicon.objc — The main Rubicon module

This is the main namespace of Rubicon-ObjC. Rubicon is structured into multiple submodules of *rubicon.objc*, and the most commonly used attributes from these submodules are exported via the *rubicon.objc* module. This means that most users only need to import and use the main *rubicon.objc* module; the individual submodules only need to be used for attributes that are not also available on *rubicon.objc*.

Exported Attributes

This is a full list of all attributes exported on the *rubicon.objc* module. For detailed documentation on these attributes, click the links below to visit the relevant sections of the submodules' documentation.

From rubicon.objc.api

- *Block*
- *NSArray*
- *NSDictionary*
- *NSMutableArray*
- *NSMutableDictionary*
- *NSObject*
- *NSObjectProtocol*
- *ObjCBlock*
- *ObjCClass*
- *ObjCInstance*
- *ObjCMetaClass*
- *ObjCProtocol*
- *at()*
- *ns_from_py()*
- *objc_classmethod()*
- *objc_const()*
- *objc_ivar()*
- *objc_method()*
- *objc_property()*
- *objc_rawmethod()*
- *py_from_ns()*

From rubicon.objc.runtime

- *SEL*
- *send_message()*
- *send_super()*

From rubicon.objc.types

- *CFIndex*
- *CFRange*
- *CGFloat*
- *CGGlyph*
- *CGPoint*
- CGPointMake()
- *CGRect*
- CGRectMake()
- *CGSize*
- CGSizeMake()
- *NSEdgeInsets*
- NSEdgeInsetsMake()
- *NSInteger*
- NSMakePoint()
- NSMakeRect()
- NSMakeSize()
- *NSPoint*
- *NSRange*
- *NSRect*
- *NSSize*

- *NSTimeInterval*
- *NSUInteger*
- *NSZeroPoint*
- *UIEdgeInsets*
- UIEdgeInsetsMake()
- *UIEdgeInsetsZero*
- *UniChar*
- *unichar*

2.4.2 rubicon.objc.api — The high-level Rubicon API

This module contains Rubicon's main high-level APIs, which allow easy interaction with Objective-C classes and objects using Pythonic syntax.

Nearly all attributes of this module are also available on the main *rubicon.objc* module, and if possible that module should be used instead of importing *rubicon.objc.api* directly.

Objective-C objects

class rubicon.objc.api.**ObjCInstance**(*ptr*)

Python wrapper for an Objective-C instance.

The constructor accepts an *objc_id* or anything that can be cast to one, such as a c_void_p, or an existing *ObjCInstance*.

ObjCInstance objects are cached — this means that for every Objective-C object there can be at most one *ObjCInstance* object at any time. Rubicon will automatically create new *ObjCInstance*s or return existing ones as needed.

The returned object's Python class is not always exactly *ObjCInstance*. For example, if the passed pointer refers to a class or a metaclass, an instance of *ObjCClass* or *ObjCMetaClass* is returned as appropriate. Additional custom *ObjCInstance* subclasses may be defined and registered using *register_type_for_objcclass()*. Creating an *ObjCInstance* from a nil pointer returns None.

Rubicon currently does not perform any automatic memory management on the Objective-C object wrapped in an *ObjCInstance*. It is the user's responsibility to retain and release wrapped objects as needed, like in Objective-C code without automatic reference counting.

ptr

_as_parameter_

The wrapped object pointer as an *objc_id*. This attribute is also available as _as_parameter_ to allow *ObjCInstance*s to be passed into ctypes functions.

objc_class

The Objective-C object's class, as an *ObjCClass*.

__str__()

Get a human-readable representation of self.

By default, self.description converted to a Python string is returned. If self.description is nil, self.debugDescription converted to a Python is returned instead. If that is also nil, repr(self) is returned as a fallback.

__repr__()

Get a debugging representation of self, which includes the Objective-C object's class and debugDescription.

__getattr__(*name*)

Allows accessing Objective-C properties and methods using Python attribute syntax.

If `self` has a Python attribute with the given name, its value is returned.

If there is an Objective-C property with the given name, its value is returned using its getter method. An attribute is considered a property if any of the following are true:

- A property with the name is present on the class (i.e. declared using `@property` in the source code)
- There is both a getter and setter method for the name
- The name has been declared as a property using *ObjCClass.declare_property()*

Otherwise, a method matching the given name is looked up. *ObjCInstance* understands two syntaxes for calling Objective-C methods:

- "Flat" syntax: the Objective-C method name is spelled out in the attribute name, with all colons replaced with underscores, and all arguments are passed as positional arguments. For example, the Objective-C method call `[self initWithWidth:w height:h]` translates to `self.initWithWidth_height_(w, h)`.
- "Interleaved" syntax: the Objective-C method name is split up between the attribute name and the keyword arguments passed to the returned method. For example, the Objective-C method call `[self initWithRed:r green:g blue:b]` translates to `self.initWithRed(r, green=g, blue=b)`.

The "interleaved" syntax is usually preferred, since it looks more similar to normal Objective-C syntax. However, the "flat" syntax is also fully supported. Certain method names require the "flat" syntax, for example if two arguments have the same label (e.g. `performSelector:withObject:withObject:`), which is not supported by Python's keyword argument syntax.

> **Warning:** The "interleaved" syntax currently ignores the ordering of its keyword arguments. However, in the interest of readability, the keyword arguments should always be passed in the same order as they appear in the method name.
>
> This also means that two methods whose names which differ only in the ordering of their keywords will conflict with each other, and can only be called reliably using "flat" syntax.
>
> As of Python 3.6, the order of keyword arguments passed to functions is preserved (**PEP 468**). In the future, once Rubicon requires Python 3.6 or newer, "interleaved" method calls will respect keyword argument order. This will fix the kind of conflict described above, but will also disallow specifying the keyword arguments out of order.

__setattr__(*name*, *value*)

Allows modifying Objective-C properties using Python syntax.

If `self` has a Python attribute with the given name, it is set. Otherwise, the name should refer to an Objective-C property, whose setter method is called with `value`.

rubicon.objc.api.**objc_const**(*dll*, *name*)

Create an *ObjCInstance* from a global pointer variable in a CDLL.

This function is most commonly used to access constant object pointers defined by a library/framework, such as NSCocoaErrorDomain.

Objective-C classes

class rubicon.objc.api.**ObjCClass**(*name_or_ptr*[, *bases*, *attrs*[, *protocols=()*, *auto_rename=None*]])

Python wrapper for an Objective-C class.

ObjCClass is a subclass of *ObjCInstance* and supports the same syntaxes for calling methods and accessing properties.

The constructor accepts either the name of an Objective-C class to look up (as `str` or `bytes`), or a pointer to an existing class object (in any form accepted by *ObjCInstance*).

If given a pointer, it must refer to an Objective-C class; pointers to other objects are not accepted. (Use *ObjCInstance* to wrap a pointer that might also refer to other kinds of objects.) If the pointer refers to a metaclass, an instance of *ObjCMetaClass* is returned instead. Creating an *ObjCClass* from a `Nil` pointer returns `None`.

ObjCClass can also be called like `type`, with three arguments (name, bases list, namespace mapping). This form is called implicitly by Python's `class` syntax, and is used to create a new Objective-C class from Python (see *Creating custom Objective-C classes and protocols*). The bases list must contain exactly one *ObjCClass* to be extended by the new class. An optional `protocols` keyword argument is also accepted, which must be a sequence of *ObjCProtocol*s for the new class to adopt.

If the name of the class has already registered with the Objective C runtime, the `auto_rename` option can be used to ensure that the Objective C name for the new class will be unique. A numeric suffix will be appended to the Objective C name to ensure uniqueness (for example, `MyClass` will be renamed to `MyClass_2`, `MyClass_3` etc until a unique name is found). By default, classes will *not* be renamed, unless *ObjCClass.auto_rename* is set at the class level.

name

The name of this class, as a `str`.

superclass

The superclass of this class, or `None` if this is a root class (such as *NSObject*).

protocols

The protocols adopted by this class.

auto_rename = False

A `bool` value describing whether a defined class should be renamed automatically if a class with the same name already exists in the Objective C runtime.

declare_property(*name*)

Declare the instance method `name` to be a property getter.

This causes the attribute named `name` on instances of this class to be treated as a property rather than a method — accessing it returns the property's value, without requiring an explicit method call. See *ObjCInstance.__getattr__* for a full description of how attribute access behaves for properties.

Most properties do not need to be declared explicitly using this method, as they are detected automatically by *ObjCInstance.__getattr__*. This method only needs to be used for properties that are read-only and don't have a `@property` declaration in the source code, because Rubicon cannot tell such properties apart from normal zero-argument methods.

Note: In the standard Apple SDKs, some properties are introduced as regular methods in one system version, and then declared as properties in a later system version. For example, the `description` method/property of *NSObject* was declared as a regular method up to OS X 10.9, but changed to a property as of OS X 10.10.

Such properties cause compatibility issues when accessed from Rubicon: `obj.description()` works on 10.9 but is a `TypeError` on 10.10, whereas `obj.description` works on 10.10 but returns a method object on 10.9. To solve this issue, the property can be declared explicitly using `NSObject.declare_property('description')`, so that it can always be accessed using `obj.description`.

declare_class_property(*name*)

Declare the class method `name` to be a property getter.

This is equivalent to `self.objc_class.declare_property(name)`.

__instancecheck__(*instance*)

Check whether the given object is an instance of this class.

If the given object is not an Objective-C object, `False` is returned.

This method allows using *ObjCClass*es as the second argument of `isinstance()`: `isinstance(obj, NSString)` is equivalent to `obj.isKindOfClass(NSString)`.

__subclasscheck__(*subclass*)

Check whether the given class is a subclass of this class.

If the given object is not an Objective-C class, `TypeError` is raised.

This method allows using *ObjCClass*es as the second argument of `issubclass()`: `issubclass(cls, NSValue)` is equivalent to `obj.isSubclassOfClass(NSValue)`.

class rubicon.objc.api.**ObjCMetaClass**(*name_or_ptr*)

Python wrapper for an Objective-C metaclass.

ObjCMetaClass is a subclass of *ObjCClass* and supports almost exactly the same operations and methods. However, there is usually no need to look up a metaclass manually. The main reason why *ObjCMetaClass* is a separate class is to differentiate it from *ObjCClass* in the `repr()`. (Otherwise there would be no way to tell classes and metaclasses apart, since metaclasses are also classes, and have exactly the same name as their corresponding class.)

The constructor accepts either the name of an Objective-C metaclass to look up (as `str` or `bytes`), or a pointer to an existing metaclass object (in any form accepted by *ObjCInstance*).

If given a pointer, it must refer to an Objective-C metaclass; pointers to other objects are not accepted. (Use *ObjCInstance* to wrap a pointer that might also refer to other kinds of objects.) Creating an *ObjCMetaClass* from a `Nil` pointer returns `None`.

Standard Objective-C and Foundation classes

The following classes from the Objective-C runtime and the Foundation framework are provided as *ObjCClass*es for convenience. (Other classes not listed here can be looked up by passing a class name to the *ObjCClass* constructor.)

Note: None of the following classes have a usable Python-style constructor - for example, you *cannot* call `NSString("hello")` to create an Objective-C string from a Python string. To create instances of these classes, you should use *ns_from_py()* (also called *at()*): `ns_from_py("hello")` returns a *NSString* instance with the value `hello`.

class rubicon.objc.api.**NSObject**

The NSObject class from <objc/NSObject.h>.

Note: See the *ObjCInstance* documentation for a list of operations that Rubicon supports on all objects.

debugDescription

description

These Objective-C properties have been declared using *ObjCClass.declare_property()* and can always be accessed using attribute syntax.

class rubicon.objc.api.**Protocol**

The Protocol class from <objc/Protocol.h>.

Note: This class has no (non-deprecated) Objective-C methods; protocol objects can only be manipulated using Objective-C runtime functions. Rubicon automatically wraps all *Protocol* objects using *ObjCProtocol*, which provides an easier interface for working with protocols.

class rubicon.objc.api.**NSNumber**

The NSNumber class from <Foundation/NSValue.h>.

Note: This class can be converted to and from standard Python primitives (`bool`, `int`, `float`) using *py_from_ns()* and *ns_from_py()*.

class rubicon.objc.api.**NSDecimalNumber**

The NSDecimalNumber class from <Foundation/NSDecimalNumber.h>.

Note: This class can be converted to and from Python `decimal.Decimal` using *py_from_ns()* and *ns_from_py()*.

class rubicon.objc.api.**NSString**

The NSString class from <Foundation/NSString.h>.

This class also supports all methods that `str` does.

Note: This class can be converted to and from Python `str` using *py_from_ns()* and *ns_from_py()*. You can also call `str(nsstring)` to convert a `NSString` to `str`.

NSString objects consist of UTF-16 code units, unlike `str`, which consists of Unicode code points. All *NSString* indices and iteration are based on UTF-16, even when using the Python-style operations/methods. If indexing or iteration based on code points is required, convert the *NSString* to `str` first.

__str__()

Return the value of this *NSString* as a `str`.

UTF8String

This Objective-C property has been declared using *ObjCClass.declare_property()* and can always be accessed using attribute syntax.

class rubicon.objc.api.**NSData**

The NSData class from <Foundation/NSData.h>.

Note: This class can be converted to and from Python `bytes` using *py_from_ns()* and *ns_from_py()*.

class rubicon.objc.api.**NSArray**

The NSArray class from <Foundation/NSArray.h>.

Note: This class can be converted to and from Python `list` using *py_from_ns()* and *ns_from_py()*.

`py_from_ns(nsarray)` will recursively convert `nsarray`'s elements to Python objects, where possible. To avoid this recursive conversion, use `list(nsarray)` instead.

`ns_from_py(pylist)` will recursively convert `pylist`'s elements to Objective-C. As there is no way to store Python object references as Objective-C objects yet, this recursive conversion cannot be avoided. If any of `pylist`'s elements cannot be converted to Objective-C, an error is raised.

__getitem__(*index*)

__len__()

__iter__()

__contains__(*value*)

__eq__(*other*)

__ne__(*other*)

index(*value*)

count(*value*)

copy()

Python-style sequence interface.

class rubicon.objc.api.**NSMutableArray**

The NSMutableArray class from <Foundation/NSArray.h>.

Note: This class can be converted to and from Python exactly like its superclass `NSArray`.

__setitem__(*index*, *value*)

__delitem__(*index*)

append(*value*)

clear()

extend(*values*)

insert(*index*, *value*)

pop([*index=-1*])

remove(*value*)

reverse()

Python-style mutable sequence interface.

class rubicon.objc.api.**NSDictionary**

The NSDictionary class from <Foundation/NSDictionary.h>.

Note: This class can be converted to and from Python `dict` using *py_from_ns()* and *ns_from_py()*.

`py_from_ns(nsdict)` will recursively convert `nsdict`'s keys and values to Python objects, where possible. To avoid the recursive conversion of the values, use `{py_from_ns(k): v for k, v in nsdict.items()}`. The conversion of the keys cannot be avoided, because Python `dict` keys need to be hashable, which *`ObjCInstance`* is not. If any of the keys convert to a Python object that is not hashable, an error is raised (regardless of which conversion method you use).

`ns_from_py(pydict)` will recursively convert `pydict`'s keys and values to Objective-C. As there is no way to store Python object references as Objective-C objects yet, this recursive conversion cannot be avoided. If any of `pydict`'s keys or values cannot be converted to Objective-C, an error is raised.

__getitem__(*key*)

__len__()

__iter__()

__contains__(*key*)

__eq__(*other*)

__ne__(*other*)

copy()

get(*key*[, *default=None*])

keys()

items()

values()

Python-style mapping interface.

Note: Unlike most Python mappings, *`NSDictionary`*'s *`keys`*, *`values`*, and *`items`* methods don't return dynamic views of the dictionary's keys, values, and items.

`keys` and *`values`* return lists that are created each time the methods are called, which can have an effect on performance and memory usage for large dictionaries. To avoid this, you can cache the return values of *`keys`* and *`values`*, or convert the *`NSDictionary`* to a Python `dict` beforehand.

`items` is currently implemented as a generator, meaning that it returns a single-use iterator. If you need to iterate over *`items`* more than once or perform other operations on it, you should convert it to a Python `set` or `list` first.

---

class rubicon.objc.api.**NSMutableDictionary**

The NSMutableDictionary class from <Foundation/NSDictionary.h>.

Note: This class can be converted to and from Python exactly like its superclass `NSDictionary`.

__setitem__(*key*, *value*)

__delitem__(*key*)

clear()

pop(*item*[, *default*])

popitem()

setdefault(*key*[, *default=None*])

update([*other*,]***kwargs*)

Python-style mutable mapping interface.

Objective-C protocols

class rubicon.objc.api.**ObjCProtocol**(*name_or_ptr*[, *bases*, *attrs*[, *auto_rename=None*]])

Python wrapper for an Objective-C protocol.

The constructor accepts either the name of an Objective-C protocol to look up (as `str` or `bytes`), or a pointer to an existing protocol object (in any form accepted by *ObjCInstance*).

If given a pointer, it must refer to an Objective-C protocol; pointers to other objects are not accepted. (Use *ObjCInstance* to wrap a pointer that might also refer to other kinds of objects.) Creating an *ObjCProtocol* from a `nil` pointer returns `None`.

ObjCProtocol can also be called like `type`, with three arguments (name, bases list, namespace mapping). This form is called implicitly by Python's `class` syntax, and is used to create a new Objective-C protocol from Python (see *Creating custom Objective-C classes and protocols*). The bases list can contain any number of *ObjCProtocol* objects to be extended by the new protocol.

If the name of the protocol has already registered with the Objective C runtime, the `auto_rename` option can be used to ensure that the Objective C name for the new protocol will be unique. A numeric suffix will be appended to the Objective C name to ensure uniqueness (for example, `MyProtocol` will be renamed to `MyProtocol_2`, `MyProtocol_3` etc until a unique name is found). By default, protocols will *not* be renamed, unless *ObjCProtocol.auto_rename* is set at the class level.

name

The name of this protocol, as a `str`.

protocols

The protocols that this protocol extends.

auto_rename = False

A `bool` value whether a defined protocol should be renamed automatically if a protocol with the same name is already exists.

__instancecheck__(*instance*)

Check whether the given object conforms to this protocol.

If the given object is not an Objective-C object, `False` is returned.

This method allows using *ObjCProtocol*s as the second argument of `isinstance()`: `isinstance(obj, NSCopying)` is equivalent to `obj.conformsToProtocol(NSCopying)`.

__subclasscheck__(*subclass*)

Check whether the given class or protocol conforms to this protocol.

If the given object is not an Objective-C class or protocol, `TypeError` is raised.

This method allows using *ObjCProtocol*s as the second argument of `issubclass()`: `issubclass(cls, NSCopying)` is equivalent to `cls.conformsToProtocol(NSCopying)`, and `issubclass(proto, NSCopying)` is equivalent to `protocol_conformsToProtocol(proto, NSCopying)`.

Standard Objective-C and Foundation protocols

The following protocols from the Objective-C runtime and the Foundation framework are provided as *ObjCProtocol*s for convenience. (Other protocols not listed here can be looked up by passing a protocol name to the *ObjCProtocol* constructor.)

rubicon.objc.api.**NSObjectProtocol**

The NSObject protocol from <objc/NSObject.h>. The protocol is exported as *NSObjectProtocol* in Python because it would otherwise clash with the *NSObject* class.

Converting objects between Objective-C and Python

rubicon.objc.api.**py_from_ns**(*nsobj*)

Convert a Foundation object into an equivalent Python object if possible.

Currently supported types:

- ***objc_id*: Wrapped in an** *ObjCInstance* and converted as below
- *NSString*: Converted to `str`
- *NSData*: Converted to `bytes`
- *NSDecimalNumber*: Converted to `decimal.Decimal`
- ***NSDictionary*: Converted to `dict`, with all keys and** values converted recursively
- ***NSArray*: Converted to `list`, with all elements converted** recursively
- ***NSNumber*: Converted to a `bool`, `int` or** `float` based on the type of its contents

Other objects are returned unmodified as an *ObjCInstance*.

rubicon.objc.api.**ns_from_py**(*pyobj*)

Convert a Python object into an equivalent Foundation object. The returned object is autoreleased.

This function is also available under the name *at()*, because its functionality is very similar to that of the Objective-C `@` operator and literals.

Currently supported types:

- `None`, *ObjCInstance*: Returned as-is
- **`enum.Enum`: Replaced by their `value` and** converted as below
- `str`: Converted to *NSString*
- `bytes`: Converted to *NSData*
- `decimal.Decimal`: Converted to *NSDecimalNumber*
- **`dict`: Converted to *NSDictionary*, with all keys and** values converted recursively
- **`list`: Converted to *NSArray*, with all elements converted** recursively

- **bool, int, float: Converted to** *NSNumber*

Other types cause a TypeError.

rubicon.objc.api.**at**(*pyobj*)

Alias for *ns_from_py()*.

Creating custom Objective-C classes and protocols

Custom Objective-C classes are defined using Python `class` syntax, by subclassing an existing *ObjCClass* object:

```
class MySubclass(NSObject):
    # method, property, etc. definitions go here
```

A custom Objective-C class can only have a single superclass, since Objective-C does not support multiple inheritance. However, the class can conform to any number of protocols, which are specified by adding the `protocols` keyword argument to the base class list:

```
class MySubclass(NSObject, protocols=[NSCopying, NSMutableCopying]):
    # method, property, etc. definitions go here
```

Note: Rubicon requires specifying a superclass when defining a custom Objective-C class. If you don't need to extend any specific class, use *NSObject* as the superclass.

Although Objective-C technically allows defining classes without a base class (so-called *root classes*), this is almost never the desired behavior (attempting to do so causes a compiler error by default). In practice, this feature is only used in the definitions of core Objective-C classes like *NSObject*. Because of this, Rubicon does not support defining Objective-C root classes.

Similar syntax is used to define custom Objective-C protocols. Unlike classes, protocols can extend multiple other protocols:

```
class MyProtocol(NSCopying, NSMutableCopying):
    # method, property, etc. definitions go here
```

A custom protocol might not need to extend any other protocol at all. In this case, we need to explicitly tell Python to define an *ObjCProtocol*. Normally Python detects the metaclass automatically by examining the base classes, but in this case there are none, so we need to specify the metaclass manually.

```
class MyProtocol(metaclass=ObjCProtocol):
    # method, property, etc. definitions go here
```

Defining methods

rubicon.objc.api.**objc_method**(*py_method*)

Exposes the decorated method as an Objective-C instance method in a custom class or protocol.

In a custom Objective-C class, decorating a method with *@objc_method* makes it available to Objective-C: a corresponding Objective-C method is created in the new Objective-C class, whose implementation calls the decorated Python method. The Python method receives all arguments (including `self`) from the Objective-C method call, and its return value is passed back to Objective-C.

In a custom Objective-C protocol, the behavior is similar, but the method body is ignored, since Objective-C protocol methods have no implementations. By convention, the method body in this case should be empty (`pass`). (Since the method is never called, you could put any other code there as well, but doing so is misleading and discouraged.)

`rubicon.objc.api.`**`objc_classmethod`**(*py_method*)

Exposes the decorated method as an Objective-C class method in a custom class or protocol.

This decorator behaves exactly like *@objc_method*, except that the decorated method becomes a class method, so it is exposed on the Objective-C class rather than its instances.

Method naming

The name of a Python-defined Objective-C method is same as the Python method's name, but with all underscores (_) replaced with colons (`:`) — for example, `initWithWidth_height_` becomes `initWithWidth:height:`.

Warning: The Objective-C *language* imposes certain requirements on the usage of colons in method names: a method's name must contain exactly as many colons as the method has arguments (excluding the implicit `self` and `_cmd` parameters), and the name of a method with arguments must end with a colon. For example, a method called `init` takes no arguments, `initWithSize:` takes a single argument, `initWithWidth:height:` takes two, etc. `initWithSize:spam` is an invalid method name.

These requirements are not enforced by the Objective-C *runtime*, but methods that do not follow them cannot easily be used from regular Objective-C code.

In addition, although the Objective-C language allows method names with multiple consecutive colons or a colon at the start of the name, such names are considered bad style and never used in practice. For example, `spam::`, `:ham:`, and `:` are unusual, but valid method names.

Future versions of Rubicon may warn about or disallow such nonstandard method names.

Parameter and return types

The argument and return types of a Python-created Objective-C method are determined based on the Python method's type annotations. The annotations may contain any `ctypes` type, as well as any of the Python types accepted by *ctype_for_type()*. If a parameter or the return type is not specified, it defaults to *ObjCInstance*. The `self` parameter is special-cased — its type is always *ObjCInstance*, even if annotated otherwise. To annotate a method as returning `void`, set its return type to `None`.

Before being passed to the Python method, any object parameters (*objc_id*) are automatically converted to *ObjCInstance*. If the method returns an Objective-C object, it is converted using *ns_from_py()* before being returned to Objective-C. These automatic conversions can be disabled by using *objc_rawmethod()* instead of *objc_method()*.

The implicit `_cmd` parameter is not passed to the Python method, as it is normally redundant and not needed. If needed, the `_cmd` parameter can be accessed by using *objc_rawmethod()* instead of *objc_method()*.

`rubicon.objc.api.`**`objc_rawmethod`**(*py_method*)

Exposes the decorated method as an Objective-C instance method in a custom class, with fewer convenience features than *objc_method()*.

This decorator behaves similarly to *@objc_method*. However, unlike with *objc_method()*, no automatic conversions are performed (aside from those by `ctypes`). This means that all parameter and return types must be provided as `ctypes` types (no *ctype_for_type()* conversion is performed), all arguments are passed in their raw form as received from `ctypes`, and the return value must be understood by `ctypes`.

In addition, the implicit `_cmd` parameter is exposed to the Python method, which is not the case when using *objc_method()*. This means that the decorated Python method must always have an additional `_cmd` parameter after `self`; if it is missing, there will be errors at runtime due to mismatched argument counts. Like `self`, `_cmd` never needs to be annotated, and any annotations on it are ignored.

Defining properties and `ivars`

rubicon.objc.api.**objc_property**(*vartype=<class 'rubicon.objc.runtime.objc_id'>*, *weak=False*)

Defines a property in a custom Objective-C class or protocol.

This class should be called in the body of an Objective-C subclass or protocol, for example:

```
class MySubclass(NSObject):
    counter = objc_property(NSInteger)
```

The property type may be any `ctypes` type, as well as any of the Python types accepted by *ctype_for_type()*.

Defining a property automatically defines a corresponding getter and setter. Following standard Objective-C naming conventions, for a property `name` the getter is called `name` and the setter is called `setName:`.

In a custom Objective-C class, implementations for the getter and setter are also generated, which store the property's value in an `ivar` called `_name`. If the property has an object type, the generated setter keeps the stored object retained, and releases it when it is replaced.

In a custom Objective-C protocol, only the metadata for the property is generated.

If `weak` is `True`, the property will be created as a weak property. When assigning an object to it, the reference count of the object will not be increased. When the object is deallocated, the property value is set to None. Weak properties are only supported for Objective-C or Python object types.

rubicon.objc.api.**objc_ivar**(*vartype*)

Defines an `ivar` in a custom Objective-C class.

If you want to store additional data on a custom Objective-C class, it is recommended to use properties (*objc_property()*) instead of `ivars`. Properties are a more modern and high-level Objective-C feature, which automatically deal with reference counting for objects, and creation of getters and setters.

The `ivar` type may be any `ctypes` type.

Unlike properties, the contents of an `ivar` cannot be accessed or modified using Python attribute syntax. Instead, the *get_ivar()* and *set_ivar()* functions need to be used.

rubicon.objc.api.**get_ivar**(*obj*, *varname*, *weak=False*)

Get the value of obj's `ivar` named `varname`.

The returned object is a `ctypes` data object.

For non-object types (everything except *objc_id* and subclasses), the returned data object is backed by the `ivar`'s actual memory. This means that the data object is only usable as long as the "owner" object is alive, and writes to it will directly change the `ivar`'s value.

For object types, the returned data object is independent of the `ivar`'s memory. This is because object `ivars` may be weak, and thus cannot always be accessed directly by their address.

rubicon.objc.api.**set_ivar**(*obj*, *varname*, *value*, *weak=False*)

Set obj's `ivar` `varname` to value. If `weak` is `True`, only a weak reference to the value is stored.

value must be a `ctypes` data object whose type matches that of the `ivar`.

Objective-C blocks

Blocks are the Objective-C equivalent of function objects, so Rubicon provides ways to call Objective-C blocks from Python and to pass Python callables to Objective-C as blocks.

Automatic conversion

If an Objective-C method returns a block (according to its type encoding), Rubicon will convert the return value to a special *ObjCInstance* that can be called in Python:

```
block = an_objc_instance.methodReturningABlock()
res = block(arg, ...)
```

Similarly, if an Objective-C method has a parameter that expects a block, you can pass in a Python callable object, and it will be converted to an Objective-C block. In this case, the callable object needs to have parameter and return type annotations, so that Rubicon can expose this type information to the Objective-C runtime:

```
def result_handler(res: objc_id) -> None:
    print(ObjCInstance(res))

an_objc_instance.doSomethingWithResultHandler(result_handler)
```

If you are writing a custom Objective-C method (see *Creating custom Objective-C classes and protocols*), you can annotate parameter or return types using *objc_block* so that Rubicon converts them appropriately:

```
class AnObjCClass(NSObject):
    @objc_method
    def methodReturningABlock() -> objc_block:
        def the_block(arg: NSInteger) -> NSUInteger:
            return abs(arg)
        return the_block

    @objc_method
    def doSomethingWithResultHandler_(result_handler: objc_block) -> None:
        res = SomeClass.someMethod()
        result_handler(res)
```

Note: These automatic conversions are mostly equivalent to the manual conversions described in the next section. There are internal technical differences between automatic and manual conversions, but they are not noticeable to most users.

The internals of automatic conversion and *objc_block* handling may change in the future, so if you need more control over the block conversion process, you should use the manual conversions described in the next section.

Manual conversion

These classes are used to manually convert blocks to Python callables and vice versa. You may need to use them to perform these conversions outside of Objective-C method calls, or if you need more control over the block's type signature.

class rubicon.objc.api.**ObjCBlock**(*pointer*[, *return_type*, **arg_types*])

Python wrapper for an Objective-C block object.

This class is used to manually wrap an Objective-C block so that it can be called from Python. Usually Rubicon will do this automatically, if the block object was returned from an Objective-C method whose return type is declared to be a block type. If this automatic detection fails, for example if the method's return type is generic `id`, Rubicon has no way to tell that the object in question is a block rather than a regular Objective-C object. In that case, the object needs to be manually wrapped using *ObjCBlock*.

The constructor takes a block object, which can be either an *ObjCInstance*, or a raw *objc_id* pointer.

Note: *objc_block* is also accepted, because it is a subclass of *objc_id*). Normally you do not need to make use of this, because in most cases Rubicon will automatically convert *objc_block*s to a callable object.

In most cases, Rubicon can automatically determine the block's return type and parameter types. If a block object doesn't have return/parameter type information at runtime, Rubicon will raise an error when attempting to convert it. In that case, you need to explicitly pass the correct return type and parameter types to *ObjCBlock* using the `restype` and `argtypes` parameters.

__call__(**args*)

Invoke the block object with the given arguments.

The arguments and return value are converted from/to Python objects according to the default `ctypes` rules, based on the block's return and parameter types.

class rubicon.objc.api.**Block**(*func*[, *restype*, **argtypes*])

A wrapper that exposes a Python callable object to Objective-C as a block.

Note: *Block* instances are currently *not* callable from Python, unlike *ObjCBlock*.

The constructor accepts any Python callable object.

If the callable has parameter and return type annotations, they are used as the block's parameter and return types. This allows using *Block* as a decorator:

```
@Block
def the_block(arg: NSInteger) -> NSUInteger:
    return abs(arg)
```

For callables without type annotations, the parameter and return types need to be passed to the *Block* constructor in the `restype` and `argtypes` arguments:

```
the_block = Block(abs, NSUInteger, NSInteger)
```

Defining custom subclasses of `ObjCInstance`

The following functions can be used to register custom subclasses of *ObjCInstance* to be used when wrapping instances of a certain Objective-C class. This mechanism is for example used by Rubicon to provide Python-style operators and methods on standard Foundation classes, such as *NSString* and *NSDictionary*.

`rubicon.objc.api.`**`register_type_for_objcclass`**(*pytype*, *objcclass*)

Register a conversion from an Objective-C class to an *ObjCInstance* subclass.

After a call of this function, when Rubicon wraps an Objective-C object that is an instance of `objcclass` (or a subclass), the Python object will have the class `pytype` rather than *ObjCInstance*. See *type_for_objcclass()* for a full description of the lookup process.

> **Warning:** This function should only be called if no instances of `objcclass` (or a subclass) have been wrapped by Rubicon yet. If the function is called later, it will not fully take effect: the types of existing instances do not change, and mappings for subclasses of `objcclass` are not updated.

`rubicon.objc.api.`**`for_objcclass`**(*objcclass*)

Decorator for registering a conversion from an Objective-C class to an *ObjCInstance* subclass.

This is equivalent to calling *register_type_for_objcclass()* on the decorated class.

`rubicon.objc.api.`**`type_for_objcclass`**(*objcclass*)

Look up the *ObjCInstance* subclass used to represent instances of the given Objective-C class in Python.

If the exact Objective-C class is not registered, each superclass is also checked, defaulting to *ObjCInstance* if none of the classes in the superclass chain is registered. Afterwards, all searched superclasses are registered for the *ObjCInstance* subclass that was found. (This speeds up future lookups, and ensures that previously computed mappings are not changed by unrelated registrations.)

This method is mainly intended for internal use by Rubicon, but is exposed in the public API for completeness.

`rubicon.objc.api.`**`unregister_type_for_objcclass`**(*objcclass*)

Unregister a conversion from an Objective-C class to an *ObjCInstance* subclass.

> **Warning:** This function should only be called if no instances of `objcclass` (or a subclass) have been wrapped by Rubicon yet. If the function is called later, it will not fully take effect: the types of existing instances do not change, and mappings for subclasses of `objcclass` are not removed.

`rubicon.objc.api.`**`get_type_for_objcclass_map`**()

Get a copy of all currently registered *ObjCInstance* subclasses as a mapping.

Keys are Objective-C class addresses as `int`s.

2.4.3 rubicon.objc.eventloop — Integrating native event loops with asyncio

Note: The documentation for this module is incomplete. You can help by *contributing to the documentation*.

class rubicon.objc.eventloop.**EventLoopPolicy**

Rubicon event loop policy.

In this policy, each thread has its own event loop. However, we only automatically create an event loop by default for the main thread; other threads by default have no event loop.

new_event_loop()

Create a new event loop and return it.

get_default_loop()

Get the default event loop.

get_child_watcher()

Get the watcher for child processes.

If not yet set, a SafeChildWatcher object is automatically created.

set_child_watcher(*watcher*)

Set the watcher for child processes.

class rubicon.objc.eventloop.**CocoaLifecycle**(*application*)

A life cycle manager for Cocoa (NSApplication) apps.

start()

stop()

class rubicon.objc.eventloop.**iOSLifecycle**

A life cycle manager for iOS (UIApplication) apps.

start()

stop()

2.4.4 rubicon.objc.runtime — Low-level Objective-C runtime access

This module contains types, functions, and C libraries used for low-level access to the Objective-C runtime.

In most cases there is no need to use this module directly — the *rubicon.objc.api* module provides the same functionality through a high-level interface.

C libraries

Some commonly used C libraries are provided as CDLLs. Other libraries can be loaded using the *load_library()* function.

rubicon.objc.runtime.**libc = load_library('c')**

The C standard library.

The following functions are accessible by default:

- free

rubicon.objc.runtime.**libobjc = load_library('objc')**

The Objective-C runtime library.

The following functions are accessible by default:

- class_addIvar
- class_addMethod
- class_addProperty
- class_addProtocol
- class_copyIvarList
- class_copyMethodList
- class_copyPropertyList
- class_copyProtocolList
- class_getClassMethod
- class_getClassVariable
- class_getInstanceMethod
- class_getInstanceSize
- class_getInstanceVariable
- class_getIvarLayout
- class_getMethodImplementation
- class_getName
- class_getProperty
- class_getSuperclass
- class_getVersion
- class_getWeakIvarLayout
- class_isMetaClass
- class_replaceMethod
- class_respondsToSelector
- class_setIvarLayout
- class_setVersion
- class_setWeakIvarLayout
- ivar_getName
- ivar_getOffset
- ivar_getTypeEncoding
- method_exchangeImplementations
- method_getImplementation
- method_getName
- method_getTypeEncoding
- method_setImplementation
- objc_allocateClassPair
- objc_copyProtocolList
- objc_getAssociatedObject
- objc_getClass
- objc_getMetaClass
- objc_getProtocol
- objc_registerClassPair
- objc_removeAssociatedObjects
- objc_setAssociatedObject
- object_getClass
- object_getClassName
- object_getIvar
- object_setIvar
- property_getAttributes
- property_getName
- property_copyAttributeList
- protocol_addMethodDescription
- protocol_addProtocol
- protocol_addProperty
- objc_allocateProtocol
- protocol_conformsToProtocol
- protocol_copyMethodDescriptionList
- protocol_copyPropertyList
- protocol_copyProtocolList
- protocol_getMethodDescription
- protocol_getName
- objc_registerProtocol
- sel_getName
- sel_isEqual
- sel_registerName

rubicon.objc.runtime.**Foundation = load_library('Foundation')**

The Foundation framework.

rubicon.objc.runtime.**load_library**(*name*)

Load and return the C library with the given name.

If the library could not be found, a `ValueError` is raised.

Internally, this function uses `ctypes.util.find_library()` to search for the library in the system-standard locations. If the library cannot be found this way, it is attempted to load the library from certain hard-coded locations, as a fallback for systems where `find_library` does not work (such as iOS).

Objective-C runtime types

These are various types used by the Objective-C runtime functions.

class rubicon.objc.runtime.**objc_id**([*value*])

The id type from <objc/objc.h>.

class rubicon.objc.runtime.**objc_block**([*value*])

The low-level type of block pointers.

This type tells Rubicon's internals that the object in question is a block and not just a regular Objective-C object, which affects method argument and return value conversions. For more details, see *Objective-C blocks*.

Note: This type does not correspond to an actual C type or Objective-C class. Although the internal structure of block objects is documented, as well as the fact that they are Objective-C objects, they do not have a documented type or class name and are not fully defined in any header file.

Aside from the special conversion behavior, this type is equivalent to *objc_id*.

---

class rubicon.objc.runtime.**SEL**([*value*])

The SEL type from <objc/objc.h>.

The constructor can be called with a bytes or str object to obtain a selector with that value.

(The normal arguments supported by c_void_p are still accepted.)

name

The selector's name as bytes.

class rubicon.objc.runtime.**Class**([*value*])

The Class type from <objc/objc.h>.

class rubicon.objc.runtime.**IMP**([*value*])

The IMP type from <objc/objc.h>.

An *IMP* cannot be called directly — it must be cast to the correct CFUNCTYPE() first, to provide the necessary information about its signature.

class rubicon.objc.runtime.**Method**([*value*])

The Method type from <objc/runtime.h>.

class rubicon.objc.runtime.**Ivar**([*value*])

The Ivar type from <objc/runtime.h>.

class rubicon.objc.runtime.**objc_property_t**([*value*])

The objc_property_t type from <objc/runtime.h>.

class rubicon.objc.runtime.**objc_property_attribute_t**([*name*, *value*])

The objc_property_attribute_t structure from <objc/runtime.h>.

name

value

The attribute name and value as C strings (bytes).

class rubicon.objc.runtime.**objc_method_description**([*name*, *value*])

The objc_method_description structure from <objc/runtime.h>.

name

The method name as a *SEL*.

types

The method's signature encoding as a C string (bytes).

class rubicon.objc.runtime.**objc_super**([*receiver*, *super_class*])

The objc_super structure from <objc/message.h>.

receiver

The receiver of the call, as an *objc_id*.

super_class

The class in which to start searching for method implementations, as a *Class*.

Objective-C runtime utility functions

These utility functions provide easier access from Python to certain parts of the Objective-C runtime.

rubicon.objc.runtime.**object_isClass**(*obj*)

Return whether the given Objective-C object is a class (or a metaclass).

This is equivalent to the *libobjc* function object_isClass from <objc/runtime.h>, which is only available since OS X 10.10 and iOS 8. This module-level function is provided to support older systems — it uses the *libobjc* function if available, and otherwise emulates it.

rubicon.objc.runtime.**get_class**(*name*)

Get the Objective-C class with the given name as a *Class* object.

If no class with the given name is loaded, None is returned, and the Objective-C runtime will log a warning message.

rubicon.objc.runtime.**should_use_stret**(*restype*)

Return whether a method returning the given type must be called using objc_msgSend_stret on the current system.

rubicon.objc.runtime.**should_use_fpret**(*restype*)

Return whether a method returning the given type must be called using objc_msgSend_fpret on the current system.

rubicon.objc.runtime.**send_message**(*receiver*, *selector*, **args*, *restype*, *argtypes=None*, *varargs=None*)

Call a method on the receiver with the given selector and arguments.

This is the equivalent of an Objective-C method call like [receiver sel:args].

Note: Some Objective-C methods take variadic arguments (varargs), for example +[NSString stringWithFormat:]. When using *send_message()*, variadic arguments are treated differently from regular arguments: they are not passed as normal function arguments in *args, but as a list in a separate varargs keyword argument.

This explicit separation of regular and variadic arguments protects against accidentally passing too many arguments into a method. By default these extra arguments would be considered varargs and passed on to the method, even if the method in question doesn't take varargs. Because of how the Objective-C runtime and most C calling conventions work, this error would otherwise be silently ignored.

The types of varargs are not included in the argtypes list. Instead, the values are automatically converted to C types using the default ctypes argument conversion rules. To ensure that all varargs are converted to the expected C types, it is recommended to manually convert all varargs to ctypes types instead of relying on automatic conversions. For example:

```
send_message(
    NSString,
    "stringWithFormat:",
    at("%i %s %@"),
    restype=objc_id,
    argtypes=[objc_id],
    varargs=[c_int(123), cast(b"C string", c_char_p), at("ObjC string")],
)
```

Parameters

- **receiver** – The object on which to call the method, as an *ObjCInstance* or *objc_id*.
- **selector** – The name of the method as a str, bytes, or *SEL*.
- **args** – The method arguments.
- **restype** – The return type of the method.
- **argtypes** – The argument types of the method, as a list. Defaults to [].
- **varargs** – Variadic arguments for the method, as a list. Defaults to []. These arguments are converted according to the default ctypes conversion rules.

rubicon.objc.runtime.**send_super**(*cls*, *receiver*, *selector*, **args*, *restype=<class 'ctypes.c_void_p'>*, *argtypes=None*, *varargs=None*, *_allow_dealloc=False*)

In the context of the given class, call a superclass method on the receiver with the given selector and arguments.

This is the equivalent of an Objective-C method call like `[super sel:args]` in the class `cls`.

In practice, the first parameter should always be the special variable `__class__`, and the second parameter should be `self`. A typical *send_super()* call would be `send_super(__class__, self, 'init')` for example.

The special variable `__class__` is defined by Python and stands for the class object that is being created by the current `class` block. The exact reasons why `__class__` must be passed manually are somewhat technical, and are not directly relevant to users of *send_super()*. For a full explanation, see issue beeware/rubicon-objc#107 and PR beeware/rubicon-objc#108.

Although it is possible to pass other values than `__class__` and `self` for the first two parameters, this is strongly discouraged. Doing so is not supported by the Objective-C language, and relies on implementation details of the superclasses.

Parameters

- **cls** – The class in whose context the `super` call is happening, as an *ObjCClass* or *Class*.
- **receiver** – The object on which to call the method, as an *ObjCInstance*, *objc_id*, or c_void_p.
- **selector** – The name of the method as a str, bytes, or *SEL*.
- **args** – The method arguments.
- **restype** – The return type of the method.
- **argtypes** – The argument types of the method, as a list. Defaults to [].
- **varargs** – Variadic arguments for the method, as a list. Defaults to []. These arguments are converted according to the default ctypes conversion rules.

rubicon.objc.runtime.**add_method**(*cls*, *selector*, *method*, *encoding*, *replace=False*)

Add a new instance method to the given class.

To add a class method, add an instance method to the metaclass.

Parameters

- **cls** – The Objective-C class to which to add the method, as an *ObjCClass* or *Class*.
- **selector** – The name for the new method, as a str, bytes, or *SEL*.
- **method** – The method implementation, as a Python callable or a C function address.
- **encoding** – The method's signature (return type and argument types) as a list. The types of the implicit self and _cmd parameters must be included in the signature.
- **replace** – If the class already implements a method with the given name, replaces the current implementation if True. Raises a ValueError error otherwise.

Returns

The ctypes C function pointer object that was created for the method's implementation. This return value can be ignored. (In version 0.4.0 and older, callers were required to manually keep a reference to this function pointer object to ensure that it isn't garbage-collected. Rubicon now does this automatically.)

rubicon.objc.runtime.**add_ivar**(*cls*, *name*, *vartype*)

Add a new instance variable of type vartype to cls.

rubicon.objc.runtime.**get_ivar**(*obj*, *varname*, *weak=False*)

Get the value of obj's ivar named varname.

The returned object is a ctypes data object.

For non-object types (everything except *objc_id* and subclasses), the returned data object is backed by the ivar's actual memory. This means that the data object is only usable as long as the "owner" object is alive, and writes to it will directly change the ivar's value.

For object types, the returned data object is independent of the ivar's memory. This is because object ivars may be weak, and thus cannot always be accessed directly by their address.

rubicon.objc.runtime.**set_ivar**(*obj*, *varname*, *value*, *weak=False*)

Set obj's ivar varname to value. If weak is True, only a weak reference to the value is stored.

value must be a ctypes data object whose type matches that of the ivar.

2.4.5 rubicon.objc.types — Non-Objective-C types and utilities

This module contains definitions for common C constants and types, and utilities for working with C types.

Common C type definitions

These are commonly used C types from various frameworks.

class rubicon.objc.types.**c_ptrdiff_t**([*value*])

The ptrdiff_t type from <stddef.h>. Equivalent to c_long on 64-bit systems and c_int on 32-bit systems.

class rubicon.objc.types.**NSInteger**([*value*])

The NSInteger type from <objc/NSObjCRuntime.h>. Equivalent to c_long on 64-bit systems and c_int on 32-bit systems.

class rubicon.objc.types.**NSUInteger**([*value*])

The NSUInteger type from <objc/NSObjCRuntime.h>. Equivalent to c_ulong on 64-bit systems and c_uint on 32-bit systems.

class rubicon.objc.types.**CGFloat**([*value*])

The CGFloat type from <CoreGraphics/CGBase.h>. Equivalent to c_double on 64-bit systems and c_float on 32-bit systems.

class rubicon.objc.types.**NSPoint**([*x*, *y*])

The NSPoint structure from <Foundation/NSGeometry.h>.

Note: On 64-bit systems this is an alias for *CGPoint*.

---

x

y

The X and Y coordinates as *CGFloat*s.

class rubicon.objc.types.**CGPoint**([*x*, *y*])

The CGPoint structure from <CoreGraphics/CGGeometry.h>.

x

y

The X and Y coordinates as *CGFloat*s.

class rubicon.objc.types.**NSSize**([*width*, *height*])

The NSSize structure from <Foundation/NSGeometry.h>.

Note: On 64-bit systems this is an alias for *CGSize*.

---

width

height

The width and height as *CGFloat*s.

class rubicon.objc.types.**CGSize**([*width*, *height*])

The CGSize structure from <CoreGraphics/CGGeometry.h>.

width

height

The width and height as *CGFloat*s.

class rubicon.objc.types.**NSRect**([*origin*, *size*])

The NSRect structure from <Foundation/NSGeometry.h>.

Note: On 64-bit systems this is an alias for *CGRect*.

---

origin

The origin as a *NSPoint*.

size

The size as a *NSSize*.

class rubicon.objc.types.**CGRect**([*origin*, *size*])

The CGRect structure from <CoreGraphics/CGGeometry.h>.

origin

The origin as a *CGPoint*.

size

The size as a *CGSize*.

class rubicon.objc.types.**UIEdgeInsets**([*top*, *left*, *bottom*, *right*])

The UIEdgeInsets structure from <UIKit/UIGeometry.h>.

top

left

bottom

right

The insets as *CGFloat*s.

class rubicon.objc.types.**NSEdgeInsets**([*top*, *left*, *bottom*, *right*])

The NSEdgeInsets structure from <Foundation/NSGeometry.h>.

top

left

bottom

right

The insets as *CGFloat*s.

class rubicon.objc.types.**NSTimeInterval**([*value*])

The NSTimeInterval type from <Foundation/NSDate.h>. Equivalent to c_double.

class rubicon.objc.types.**CFIndex**([*value*])

The CFIndex type from <CoreFoundation/CFBase.h>. Equivalent to c_longlong on 64-bit systems and c_long on 32-bit systems.

class rubicon.objc.types.**UniChar**([*value*])

The UniChar type from <MacTypes.h>. Equivalent to c_ushort.

class rubicon.objc.types.**unichar**([*value*])

The unichar type from <Foundation/NSString.h>. Equivalent to c_ushort.

class rubicon.objc.types.**CGGlyph**([*value*])

The CGGlyph type from <CoreGraphics/CGFont.h>. Equivalent to c_ushort.

class rubicon.objc.types.**CFRange**([*location*, *length*])

The CFRange type from <CoreFoundation/CFBase.h>.

location

length

The location and length as *CFIndex*es.

class rubicon.objc.types.**NSRange**([*location*, *length*])

The NSRange type from <Foundation/NSRange.h>.

location

length

The location and length as *NSUInteger*s.

Common C constants

These are commonly used C constants from various frameworks.

rubicon.objc.types.**UIEdgeInsetsZero**

The constant UIEdgeInsetsZero: a *UIEdgeInsets* instance with all insets set to zero.

rubicon.objc.types.**NSZeroPoint**

The constant NSZeroPoint: a *NSPoint* instance with the X and Y coordinates set to zero.

rubicon.objc.types.**NSIntegerMax**

The macro constant NSIntegerMax from <objc/NSObjCRuntime.h>: the maximum value that a *NSInteger* can hold.

rubicon.objc.types.**NSNotFound**

The constant NSNotFound from <Foundation/NSObjCRuntime.h>: a *NSInteger* sentinel value indicating that an item was not found (usually when searching in a collection).

Architecture detection constants

The following constants provide information about the architecture of the current environment. All of them are equivalent to the C compiler macros of the same names.

rubicon.objc.types.**__LP64__**

Indicates whether the current environment is 64-bit. If true, C `long`s and pointers are 64 bits in size, otherwise 32 bits.

rubicon.objc.types.**__i386__**

rubicon.objc.types.**__x86_64__**

rubicon.objc.types.**__arm__**

rubicon.objc.types.**__arm64__**

Each of these constants is true if the current environment uses the named architecture. At most one of these constants is true at once in a single Python runtime. (If the current architecture cannot be determined, all of these constants are false.)

Objective-C type encoding conversion

These functions are used to convert Objective-C type encoding strings to and from `ctypes` types, and to manage custom conversions in both directions.

All Objective-C encoding strings are represented as `bytes` rather than `str`.

rubicon.objc.types.**ctype_for_encoding**(*encoding*)

Return the C type corresponding to an Objective-C type encoding.

If a C type has been registered for the encoding, that type is returned. Otherwise, if the type encoding represents a compound type (pointer, array, structure, or union), the contained types are converted recursively. A new C type is then created from the converted ctypes, and is registered for the encoding (so that future conversions of the same encoding return the same C type).

For example, the type encoding `{spam=ic}` is not registered by default. However, the contained types `i` and `c` are registered, so they are converted individually and used to create a new `Structure` with two fields of the correct types. The new structure type is then registered for the original encoding `{spam=ic}` and returned.

Raises

`ValueError` – if the conversion fails at any point

rubicon.objc.types.**encoding_for_ctype**(*ctype*)

Return the Objective-C type encoding for the given ctypes type.

If a type encoding has been registered for the C type, that encoding is returned. Otherwise, if the C type is a pointer type, its pointed-to type is encoded and used to construct the pointer type encoding.

Automatic encoding of other compound types (arrays, structures, and unions) is currently not supported. To encode such types, a type encoding must be manually provided for them using *register_preferred_encoding()* or *register_encoding()*.

Raises

ValueError – if the conversion fails at any point

rubicon.objc.types.**register_preferred_encoding**(*encoding*, *ctype*)

Register a preferred conversion between an Objective-C type encoding and a C type.

"Preferred" means that any existing conversions in each direction are overwritten with the new conversion. To register an encoding without overwriting existing conversions, use *register_encoding()*.

rubicon.objc.types.**with_preferred_encoding**(*encoding*)

Register a preferred conversion between an Objective-C type encoding and the decorated C type.

This is equivalent to calling *register_preferred_encoding()* on the decorated C type.

rubicon.objc.types.**register_encoding**(*encoding*, *ctype*)

Register an additional conversion between an Objective-C type encoding and a C type.

"Additional" means that any existing conversions in either direction are *not* overwritten with the new conversion. To register an encoding and overwrite existing conversions, use *register_preferred_encoding()*.

rubicon.objc.types.**with_encoding**(*encoding*)

Register an additional conversion between an Objective-C type encoding and the decorated C type.

This is equivalent to calling *register_encoding()* on the decorated C type.

rubicon.objc.types.**unregister_encoding**(*encoding*)

Unregister the conversion from an Objective-C type encoding to its corresponding C type.

Note that this does not remove any conversions in the other direction (from a C type to this encoding). These conversions may be replaced with *register_encoding()*, or unregistered with *unregister_ctype()*. To remove all ctypes for an encoding, use *unregister_encoding_all()*.

If the encoding was not registered previously, nothing happens.

rubicon.objc.types.**unregister_encoding_all**(*encoding*)

Unregister all conversions between an Objective-C type encoding and all corresponding ctypes.

All conversions from any C type to this encoding are removed recursively using *unregister_ctype_all()*.

If the encoding was not registered previously, nothing happens.

rubicon.objc.types.**unregister_ctype**(*ctype*)

Unregister the conversion from a C type to its corresponding Objective-C type encoding.

Note that this does not remove any conversions in the other direction (from an encoding to this C type). These conversions may be replaced with *register_encoding()*, or unregistered with *unregister_encoding()*. To remove all encodings for a C type, use *unregister_ctype_all()*.

If the C type was not registered previously, nothing happens.

rubicon.objc.types.**unregister_ctype_all**(*ctype*)

Unregister all conversions between a C type and all corresponding Objective-C type encodings.

All conversions from any type encoding to this C type are removed recursively using *unregister_encoding_all()*.

If the C type was not registered previously, nothing happens.

rubicon.objc.types.**get_ctype_for_encoding_map**()

Get a copy of all currently registered encoding-to-C type conversions as a map.

rubicon.objc.types.**get_encoding_for_ctype_map**()

Get a copy of all currently registered C type-to-encoding conversions as a map.

rubicon.objc.types.**split_method_encoding**(*encoding*)

Split a method signature encoding into a sequence of type encodings.

The first type encoding represents the return type, all remaining type encodings represent the argument types.

If there are any numbers after a type encoding, they are ignored. On PowerPC, these numbers indicated each argument/return value's offset on the stack. These numbers are meaningless on modern architectures.

rubicon.objc.types.**ctypes_for_method_encoding**(*encoding*)

Convert a method signature encoding into a sequence of ctypes.

This is equivalent to first splitting the method signature encoding using *split_method_encoding()*, and then converting each individual type encoding using *ctype_for_encoding()*.

Default registered type encodings

The following table lists Objective-C's standard type encodings for primitive types, and the corresponding registered ctypes. These mappings can be considered stable, but nonetheless users should not hard code these encodings unless necessary. Instead, the *encoding_for_ctype()* function should be used to encode types, because it is less error-prone and more readable than typing encodings out by hand.

Ctype	Type encoding	Notes
None (void)	v	
c_bool	B	This refers to the bool type from C99 and C++. It is not necessarily the same as the *BOOL* type, which may be either c_byte or c_bool depending on the system and architecture.
c_byte	c	
c_ubyte	C	
c_short	s	
c_ushort	S	
c_long	l	
c_ulong	L	
c_int	i	On 32-bit systems, c_int is an alias for c_long, and will be encoded as such.
c_uint	I	On 32-bit systems, c_uint is an alias for c_ulong, and will be encoded as such.
c_longlon	q	On 64-bit systems, c_longlong is an alias for c_long, and will be encoded as such.
c_ulonglo	Q	On 64-bit systems, c_ulonglong is an alias for c_ulong, and will be encoded as such.
c_float	f	
c_double	d	
c_longdou	D	On ARM, c_longdouble is an alias for c_double, and will be encoded as such.
c_char	c	Only when encoding. Decoding c produces c_byte, to allow using signed char as a Boolean value.
c_char_p	*	
POINTER(c	*	Only when encoding. Decoding * produces c_char_p for easier use of C strings.
POINTER(c	*	Only when encoding. Decoding * produces c_char_p for easier use of C strings.
POINTER(c	*	Only when encoding. Decoding * produces c_char_p for easier use of C strings.
c_wchar	i	Only when encoding. Decoding i produces c_int.
c_wchar_p	^i	Only when encoding. Decoding ^i produces POINTER(c_int).
c_void_p	^v	
UnknownPo	^?	This encoding stands for a pointer to a type that cannot be encoded, which in practice means a function pointer.
UnknownPo	^{?}, ^(?)	Only when decoding. These encodings stand for pointers to a structure or union with unknown name and fields.
objc_id	@	Class name suffixes in the encoding (e. g. @"NSString") are ignored.
objc_bloc	@?	Block signature suffixes in the encoding (e. g. @?<v@?>) are ignored.
SEL	:	
Class	#	

class rubicon.objc.types.**UnknownPointer**(*value=None*)

Placeholder for the "unknown pointer" types ^?, ^{?} and ^(?).

Not to be confused with a ^v void pointer.

Usually a ^? is a function pointer, but because the encoding doesn't contain the function signature, you need to manually create a CFUNCTYPE with the proper types, and cast this pointer to it.

^{?} and ^(?) are pointers to a structure or union (respectively) with unknown name and fields. Such a type also cannot be used meaningfully without casting it to the correct pointer type first.

In addition, the following types defined by Rubicon are registered, but their encodings may vary depending on the system and architecture:

- ctypes.py_object
- *NSInteger*
- *NSUInteger*
- *CGFloat*
- *NSPoint*
- *CGPoint*
- *NSSize*
- *CGSize*
- *NSRect*
- *CGRect*
- *UIEdgeInsets*
- *NSEdgeInsets*
- *NSTimeInterval*
- *CFIndex*
- *UniChar*
- *unichar*
- *CGGlyph*
- *NSRange*

Conversion of Python sequences to C structures and arrays

This function is used to convert a Python sequence (such as a `tuple` or `list`) to a specific C structure or array type. This function is mainly used internally by Rubicon, to allow passing Python sequences as method parameters where a C structure or array would normally be required. Most users will not need to use this function directly.

rubicon.objc.types.**compound_value_for_sequence**(*seq*, *tp*)

Create a C structure or array of type `tp`, initialized with values from `seq`.

If `tp` is a `Structure` type, the newly created structure's fields are initialized in declaration order with the values from `seq`. `seq` must have as many elements as the structure has fields.

If `tp` is a `Array` type, the newly created array is initialized with the values from `seq`. `seq` must have as many elements as the array type.

In both cases, if a structure field type or the array element type is itself a structure or array type, the corresponding value from `seq` is recursively converted as well.

Python to ctypes type mapping

These functions are used to map Python types to equivalent `ctypes` types, and to add or remove such mappings. This mechanism is mainly used internally by Rubicon, to for example allow *ObjCInstance* to be used instead of *objc_id* in method type annotations. Most users will not need to use these functions directly.

rubicon.objc.types.**ctype_for_type**(*tp*)

Look up the C type corresponding to the given Python type.

This conversion is applied to types used in *objc_method* signatures, *objc_ivar* types, etc. This function translates Python built-in types and *rubicon.objc* classes to their `ctypes` equivalents. Unregistered types (including types that are already ctypes) are returned unchanged.

rubicon.objc.types.**register_ctype_for_type**(*tp*, *ctype*)

Register a conversion from a Python type to a C type.

rubicon.objc.types.**unregister_ctype_for_type**(*tp*)

Unregister a conversion from a Python type to a C type.

rubicon.objc.types.**get_ctype_for_type_map**()

Get a copy of all currently registered type-to-C type conversions as a mapping.

Default registered mappings

The following mappings are registered by default by Rubicon.

Python type	*Ctype*
`int`	`c_int`
`float`	`c_float`
`bool`	`c_bool`
`bytes`	`c_char_p`
`ObjCInstance`	*`objc_id`*
`ObjCClass`	*`Class`*

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

Note that the *`rubicon.objc`* package also contains other submodules not documented here. These are for internal use only and not part of the public API; they may change at any time without notice.

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

O

P

R

S

T

U

V

W

X

Y