Pyobjus Documentation

Release 1.0a1

Mathieu Virbel, Gabriel Pettier

May 13, 2023

Contents

Pyobjus is a Python library for accessing Objective-C class. If you want to access Java class, look at Pyjnius.

This documentation is divided into differents parts. We recommend you to start with *Installation*, and then head over to the *Quickstart*. If you'd rather dive into the internals of Pyobjus, check out the *API* documentation. There are also nice examples of using classes from Foundation framework. These you can find on the *Foundation framework examples* page.

CHAPTER 1

Installation

Pyobjus depends on Cython.

1.1 Installation on the Desktop

You need to install Cython first. Then, just type:

```
sudo python setup.py install
```

If you want to compile the extension within the directory for any development, just type:

```
make
```

You can run the tests suite to make sure everything is running correctly:

```
make tests
```

Or you can build the documentation yourself:

```
make html
open docs/build/html/index.html
```

1.2 Installation on iOS

Please see the *Pyobjus on iOS* section.

CHAPTER 2

Quickstart

Eager to get started? This page will give you a good introduction to Pyobjus. It's assumed you have already installed Pyobjus. If you haven't, head over the *Installation* section.

2.1 The simplest example

The simplest Pyobjus example looks something like this:

```
from pyobjus import autoclass

NSString = autoclass('NSString')
text = NSString.alloc().initWithUTF8String_('Hello world')
print text.UTF8String() # --> Hello world
```

Just save it as *test.py* (or something similar) and run it with your Python interpreter. Make sure not to call your application *pyobjus.py* because it would conflict with Pyobjus itself:

```
$ python test.py
Hello world
```

2.2 Using classes not in the standard Framework

If you want to use classes other than those available from the linked framework, you need to preload the framework first, or add the Framework to your application (iOS only). To preload the framework, you can use the pyobjus dylib_manager:

```
# We want to use NSAlert, but it's not a standard objective-C class
# so we need to either import the framework into our process (desktop)
# or link the framework to our app (iOS)
from pyobjus.dylib_manager import load_framework, INCLUDE
```

(continues on next page)

(continued from previous page)

```
load_framework(INCLUDE.AppKit)

# OR in this way
# from pyobjus.dylib_manager import load_framework
# load_framework('/System/Library/Frameworks/AppKit.framework')

# OR in this way
# from pyobjus.dylib_manager import load_dylib
# load_dylib('/System/Library/Frameworks/AppKit.framework/Versions/C/Resources/
↪BridgeSupport/AppKit.dylib')

# NOTE: there is a "Dynamic library manager" section dedicated explaining how to use␣
↪dylib_manager functions
```

Then we can use the NSAlert object:

```
from pyobjus import autoclass

# mimic the constant string operator (@"hello") in objective C
from pyobjus import objc_str

# get both nsalert and nsstring class
NSAlert = autoclass('NSAlert')
NSString = autoclass('NSString')

# create an NSAlert object, and show it.
alert = NSAlert.alloc().init()
alert.setMessageText_(objc_str('Hello world!'))
alert.runModal()
```

CHAPTER 3

How does Pyobjus work?

This part of the documentation introduces a basic understanding of how pyobjus works.

3.1 The autoclass function

So, autoclass is the heart of pyobjus. With this function, you load Objective C classes into pyobjus which then constructs a Python wrapper around these objects.

Let's say that we are in the situation where we need to load a NSString class belonging to the Foundation framework.

You can load external code into pyobjus using the `load_framework` function, or by using `load_dylib`. The `load_framework` function uses NSBundle for loading the framework into pyobjus, and the `load_dylib` function uses ctypes for loading external .dylib objects into pyobjus.

Notice that you don't need to explicitly load the Foundation framework into pyobjus because the Foundation framework is loaded by default. But if you want AppKit, for example, you can do something like this:

```
from pyobjus.dylib_manager import load_framework, INCLUDE
load_framework(INCLUDE.AppKit)
```

This will load code from the AppKit framework into pyobjus, so now we can use these classes.

But let's return to our NSString class from the Foundation framework. To load this class, type the following:

```
from pyobjus import autoclass
NSString = autoclass('NSString')
```

What happened here? So, pyobjus will call the `class_copyMethodList` function of the Objective C runtime. After that, it will create an ObjcMethod Python object for every method attached to the class as well as an ObjcProperty for every attached property. It will then return a Python representation of the NSString class with all the ObjcMethod and ObjcProperty objects attached.

So, maybe you don't want to use all the properties of the NSString class. In that case, you can call the `autoclass` function in the following way:

```
NSString = autoclass('NSString', copy_properties=False)
```

Perhaps you want to save memory and gather some speed with the autoclass method. In that case, you can specify exactly which methods you want to load. Say you want to load only the init and alloc methods of NSString. You can do that as follows:

```
NSString = autoclass('NSString',
                     load_class_methods=['alloc'],
                     load_instance_methods=['init'])
```

If you want to load only a few of the class methods, you can specify these with the *load_class_methods* optional argument. If you want to load only a few instance methods, you can specify these with the *load_instance_methods* optional argument.

So, say you want to load only the *alloc* class method and all instance methods, you can do that this way:

```
NSString = autoclass('NSString', load_class_methods=['alloc'])
```

But, maybe at some point you want to have all the NSString class methods available again. Okay, pyobjus can do that for you. You just need to call the `autoclass` method this way:

```
NSString = autoclass('NSString', reset_autoclass=True)
```

3.2 Calling Objective C methods

So, suppose that you find an appropriate way to load an Objective C class via the autoclass function. After that, you need to consider the following. In Objective C, you can do this:

```
NSString *string = [[NSString alloc] init];
```

In pyobjus, we have a similar scenario. Say that we loaded a `NSString` in the following way:

```
NSString = autoclass('NSString')
```

Now the `NSString` object contains all the *class* methods of the `NSString` Objective C class. Are you wondering how to get the *instance* methods? We can answer that question. In the same way as the native Objective C class.

So let's do this::

```
print NSString.alloc()
```

This will output:

```
>>> <__main__.NSPlaceholderString object at 0x10b372e90>
```

We now have an allocated object and can call it's instance methods, like `init`:

```
print NSString.alloc().init()
```

This will output:

```
>>> <__main__.__NSCFConstantString object at 0x10b4827d0>
```

You can also view the list of available methods with the Python `dir` function:

```
# view class methods
print dir(NSString)
# view instance methods
print dir(NSString.alloc())
```

So now we know how to use autoclass methods and how to access the class/instance methods of the loaded Objective C classes. In comparison to Python, Objective C has some additional syntax when you are passing arguments. How does pyobjus deal with this?

With Objective C, you can declare a function as follows:

```
- (void) sumNumber:(int)a and:(int)b { ... }
```

Internally, this method will be translated to `sumNumber:and:` because that's the actual method name. Okay, now things are little clearer.

So, if you remember, pyobjus calls the `class_copyMethodList` and will provide an ObjcMethod object for it. So, if you want to call this method from Python, you might suppose you can call it in this way:

```
sumNumber:and:(3, 5)
```

but that's wrong way to call Objective C methods using pyobjus. Pyobjus will internally convert every *:* into _, so now we can call it with Python in this way:

```
sumNumber_and_(3, 5)
```

So, if there is Objective C method declared in this way:

```
- (void) sumNumber:(int)a and:(int)b andAlso:(int)c { ... }
```

You will call this method with pyobjus in the way:

```
sumNumber_and_andAlso_(1, 2, 3)
```

So far we know how to call Objective C methods with pyobjus, and how to pass arguments to them. Let's try to do that with an NSString class using the *stringWithUTF8String:* class method:

```
text = NSString.stringWithUTF8String_('some string')
print text.UTF8String()
```

Here we call the *stringWithUTF8String:* class method, and after that the *UTF8String:* instance method. As you can see from the output, we will get *some string*, so we can see that method is making an NSString instance, and correctly calling and returning values from these methods which belong to NSString class.

3.3 Using Objective C properties

You may wonder if you can use Objective C properties with pyobjus, and if so, how?

Using Objective C properties is really simple. Let's first make an Objective C class:

```
#import <Foundation/Foundation.h>

@interface ObjcClass : NSObject {
}
@property (nonatomic) int some_objc_prop;
```

(continues on next page)

(continued from previous page)

```
@end

@implementation ObjcClass
@synthesize some_objc_prop;
@end
```

This really simple Objective C class has an Objective C property `some_objc_prop`. Save it as *test.m* for this example. Later we will explain `dylib_manager`, but for now, we will use its functions to load the above class into pyobjus:

```
from pyobjus.dylib_manager import load_dylib, make_dylib
from pyobjus import autoclass

# TODO: change path to your
make_dylib('/path/to/test.m', frameworks=['Foundation'])
# TODO: change path to your
load_dylib('/path/to/test.dylib')

ObjcClass = autoclass('ObjcClass')
o_cls = ObjcClass.alloc().init()

# now we can set property value
o_cls.some_objc_prop = 12345
# or retrieve value of that property
print o_cls.some_objc_prop
```

Here you can see that setting an Objective C property is very similar to how we set it in native Objective C code.

You may be wondering how pyobjus deals with Objective C properties? Pyobjus is calling getters and setters for that property because in Objective C, there are default names for getters/setters.

So for the mentioned property, the getter will be *some_objc_prop*, and the setter *setSome_objc_prop*. I suppose that you can figure out how Objective C generate names for getters and setters for properties. The getter will have the same name as the property, and the setter will be constructed in the following way: 'set' will be added as a prefix to the property name, the first letter of property will be capitalized and the rest of letters added. The result of that is the name of property setter.

Basically, that's how pyobjus manages things, and how to use pyobjus properties. It is really simple and intuitive.

CHAPTER 4

Pyobjus API tutorial

This part of documentation covers tutorials related to API of pyobjus

4.1 Using dylib_manager

You need to load code into pyobjus so it can actually find the appropriate class with the autoclass function.

Maybe you want to write some Objective C code, and you want to load it into pyobjus, or you want to use some exising *.dylib* or something similar.

These problems can be solved using the pyobjus dylib_manager. Currently it has a few functions, so let's see what we can do with them.

4.1.1 make_dylib and load_dylib functions

For the first example, let's say that we want to write our class in Objective C, and after that we want to load that class into pyobjus. Okay, let's write a class:

```
#import <Foundation/Foundation.h>

@interface ObjcClass : NSObject {
}
- (void) printFromObjectiveC;
@end

@implementation ObjcClass

- (void) printFromObjectiveC {
        printf("Hello from Objective C\n");
}

@end
```

The next step is to make a .dylib for this class, and load that .dylib into pyobjus. Suppose that we have previously saved this code into an *objc_lib.m* file.

With pyobjus you can compile *objc_lib.m* into *objc_lib.dylib* in the following way:

```
make_dylib('objc_lib.m', frameworks=['Foundation'], options=['-current_version', '1.0
↪'])
```

Here, we are asking pyobjus to link *objc_lib.m* with the Foundation framework, and that we want to set the *-current_version* option to *1.0*. You can also specify others frameworks and options if you want by just adding these elements to array.

The previous command will create an *objc_lib.dylib* file in the same directory as the *objc_lib.m* file. If you want to save it to another directory or with a different name, you can call *make_dylib* in this way:

```
make_dylib('objc_lib.m', frameworks=['Foundation'], out='/path/to/dylib/dylib_name.
↪dylib')
```

After you make a .dylib with make_dylib function, you can load the code from the .dylib into pyobjus on following way:

```
load_dylib('objc_lib.dylib')

# or if you specified anothed loation and name for .dylib
# load_dylib('/path/to/dylib/dylib_name.dylib')
```

Great, we have created a .dylib, loaded it into pyobjus and can now use the ObjcClass from our *objc_lib.m* file:

```
ObjcClass = autoclass('ObjcClass')
o_instance = ObjcClass.alloc().init()
o_instance.printFromObjectiveC()
```

This will output with:

```
>>> Hello from Objective C
```

4.1.2 load_framework function

There often can be situations when you need to load classes into pyobjus which don't belong to the Foundation framework. For example, say you want to load a class from the AppKit framework.

In these cases you can use the load_framework function of dylib_manager.

So let's see one simple example of using this function:

```
from pyobjus.dylib_manager import load_framework, INCLUDE
load_framework(INCLUDE.AppKit)
```

You may wonder what INCLUDE is, and can we load all Frameworks in this way? So INCLUDE is an enum, which contains paths to various Frameworks. Currently, INCLUDE contains paths to the following frameworks:

```
Accelerate = '/System/Library/Frameworks/Accelerate.framework',
Accounts = '/System/Library/Frameworks/Accounts.framework',
AddressBook = '/System/Library/Frameworks/AddressBook.framework',
AGL = '/System/Library/Frameworks/AGL.framework',
AppKit = '/System/Library/Frameworks/AppKit.framework',
AppKitScripting = '/System/Library/Frameworks/AppKitScripting.framework',
```

(continues on next page)

(continued from previous page)

```
AppleScriptKit = '/System/Library/Frameworks/AppleScriptKit.framework',
AppleScriptObjC = '/System/Library/Frameworks/AppleScriptObjC.framework',
AppleShareClientCore = '/System/Library/Frameworks/AppleShareClientCore.framework',
AppleTalk = '/System/Library/Frameworks/AppleTalk.framework',
ApplicationServices = '/System/Library/Frameworks/ApplicationServices.framework',
AudioToolbox = '/System/Library/Frameworks/AudioToolbox.framework',
AudioUnit = '/System/Library/Frameworks/AudioUnit.framework',
AudioVideoBridging = '/System/Library/Frameworks/AudioVideoBridging.framework',
Automator = '/System/Library/Frameworks/Automator.framework',
AVFoundation = '/System/Library/Frameworks/AVFoundation.framework',
CalendarStore = '/System/Library/Frameworks/CalendarStore.framework',
Carbon = '/System/Library/Frameworks/Carbon.framework',
CFNetwork = '/System/Library/Frameworks/CFNetwork.framework',
Cocoa = '/System/Library/Frameworks/Cocoa.framework',
Collaboration = '/System/Library/Frameworks/Collaboration.framework',
CoreAudio = '/System/Library/Frameworks/CoreAudio.framework',
CoreAudioKit = '/System/Library/Frameworks/CoreAudioKit.framework',
CoreData = '/System/Library/Frameworks/CoreData.framework',
CoreFoundation = '/System/Library/Frameworks/CoreFoundation.framework',
CoreGraphics = '/System/Library/Frameworks/CoreGraphics.framework',
CoreLocation = '/System/Library/Frameworks/CoreLocation.framework',
CoreMedia = '/System/Library/Frameworks/CoreMedia.framework',
CoreMediaIO = '/System/Library/Frameworks/CoreMediaIO.framework',
CoreMIDI = '/System/Library/Frameworks/CoreMIDI.framework',
CoreMIDIServer = '/System/Library/Frameworks/CoreMIDIServer.framework',
CoreServices = '/System/Library/Frameworks/CoreServices.framework',
CoreText = '/System/Library/Frameworks/CoreText.framework',
CoreVideo = '/System/Library/Frameworks/CoreVideo.framework',
CoreWiFi = '/System/Library/Frameworks/CoreWiFi.framework',
CoreWLAN = '/System/Library/Frameworks/CoreWLAN.framework',
DirectoryService = '/System/Library/Frameworks/DirectoryService.framework',
DiscRecording = '/System/Library/Frameworks/DiscRecording.framework',
DiscRecordingUI = '/System/Library/Frameworks/DiscRecordingUI.framework',
DiskArbitration = '/System/Library/Frameworks/DiskArbitration.framework',
DrawSprocket = '/System/Library/Frameworks/DrawSprocket.framework',
DVComponentGlue = '/System/Library/Frameworks/DVComponentGlue.framework',
DVDPlayback = '/System/Library/Frameworks/DVDPlayback.framework',
EventKit = '/System/Library/Frameworks/EventKit.framework',
ExceptionHandling = '/System/Library/Frameworks/ExceptionHandling.framework',
ForceFeedback = '/System/Library/Frameworks/ForceFeedback.framework',
Foundation = '/System/Library/Frameworks/Foundation.framework',
FWAUserLib = '/System/Library/Frameworks/FWAUserLib.framework',
GameKit = '/System/Library/Frameworks/GameKit.framework',
GLKit = '/System/Library/Frameworks/GLKit.framework',
GLUT = '/System/Library/Frameworks/GLUT.framework',
GSS = '/System/Library/Frameworks/GSS.framework',
ICADevices = '/System/Library/Frameworks/ICADevices.framework',
ImageCaptureCore = '/System/Library/Frameworks/ImageCaptureCore.framework',
ImageIO = '/System/Library/Frameworks/ImageIO.framework',
IMServicePlugIn = '/System/Library/Frameworks/IMServicePlugIn.framework',
InputMethodKit = '/System/Library/Frameworks/InputMethodKit.framework',
InstallerPlugins = '/System/Library/Frameworks/InstallerPlugins.framework',
InstantMessage = '/System/Library/Frameworks/InstantMessage.framework',
IOBluetooth = '/System/Library/Frameworks/IOBluetooth.framework',
IOBluetoothUI = '/System/Library/Frameworks/IOBluetoothUI.framework',
IOKit = '/System/Library/Frameworks/IOKit.framework',
IOSurface = '/System/Library/Frameworks/IOSurface.framework',
```

(continues on next page)

(continued from previous page)

```
JavaFrameEmbedding = '/System/Library/Frameworks/JavaFrameEmbedding.framework',
JavaScriptCore = '/System/Library/Frameworks/JavaScriptCore.framework',
JavaVM = '/System/Library/Frameworks/JavaVM.framework',
Kerberos = '/System/Library/Frameworks/Kerberos.framework',
Kernel = '/System/Library/Frameworks/Kernel.framework',
LatentSemanticMapping = '/System/Library/Frameworks/LatentSemanticMapping.framework',
LDAP = '/System/Library/Frameworks/LDAP.framework',
MediaToolbox = '/System/Library/Frameworks/MediaToolbox.framework',
Message = '/System/Library/Frameworks/Message.framework',
NetFS = '/System/Library/Frameworks/NetFS.framework',
OpenAL = '/System/Library/Frameworks/OpenAL.framework',
OpenCL = '/System/Library/Frameworks/OpenCL.framework',
OpenDirectory = '/System/Library/Frameworks/OpenDirectory.framework',
OpenGL = '/System/Library/Frameworks/OpenGL.framework',
OSAKit = '/System/Library/Frameworks/OSAKit.framework',
PCSC = '/System/Library/Frameworks/PCSC.framework',
PreferencePanes = '/System/Library/Frameworks/PreferencePanes.framework',
PubSub = '/System/Library/Frameworks/PubSub.framework',
Python = '/System/Library/Frameworks/Python.framework',
QTKit = '/System/Library/Frameworks/QTKit.framework',
Quartz = '/System/Library/Frameworks/Quartz.framework',
QuartzCore = '/System/Library/Frameworks/QuartzCore.framework',
QuickLook = '/System/Library/Frameworks/QuickLook.framework',
QuickTime = '/System/Library/Frameworks/QuickTime.framework',
Ruby = '/System/Library/Frameworks/Ruby.framework',
RubyCocoa = '/System/Library/Frameworks/RubyCocoa.framework',
SceneKit = '/System/Library/Frameworks/SceneKit.framework',
ScreenSaver = '/System/Library/Frameworks/ScreenSaver.framework',
Scripting = '/System/Library/Frameworks/Scripting.framework',
ScriptingBridge = '/System/Library/Frameworks/ScriptingBridge.framework',
Security = '/System/Library/Frameworks/Security.framework',
SecurityFoundation = '/System/Library/Frameworks/SecurityFoundation.framework',
SecurityInterface = '/System/Library/Frameworks/SecurityInterface.framework',
ServerNotification = '/System/Library/Frameworks/ServerNotification.framework',
ServiceManagement = '/System/Library/Frameworks/ServiceManagement.framework',
Social = '/System/Library/Frameworks/Social.framework',
StoreKit = '/System/Library/Frameworks/StoreKit.framework',
SyncServices = '/System/Library/Frameworks/SyncServices.framework',
System = '/System/Library/Frameworks/System.framework',
SystemConfiguration = '/System/Library/Frameworks/SystemConfiguration.framework',
Tcl = '/System/Library/Frameworks/Tcl.framework',
Tk = '/System/Library/Frameworks/Tk.framework',
TWAIN = '/System/Library/Frameworks/TWAIN.framework',
vecLib = '/System/Library/Frameworks/vecLib.framework',
VideoDecodeAcceleration = '/System/Library/Frameworks/VideoDecodeAcceleration.
↪framework',
VideoToolbox = '/System/Library/Frameworks/VideoToolbox.framework',
WebKit = '/System/Library/Frameworks/WebKit.framework',
XgridFoundation = '/System/Library/Frameworks/XgridFoundation.framework'
```

If the Framework path which you want to load isn't present in the INCLUDE enum, you can specify it manually. Let's say that the path to AppKit isn't available via the INCLUDE enum. You could load the Framework in the following way:

```
load_framework('/System/Library/Frameworks/AppKit.framework')
```

4.2 Using struct types

Pyobjus currently support `NSRange`, `NSPoint`, `NSSize` and `NSRect` structures. They are defined via the `ctypes.Structure` type.

Consider the following. You have an Objective C class with the name ObjcClass and a useRange: method of that class which is defined in this way:

```
- (void) useRange:(NSRange)r {
    printf("location: %ld, length: %ld\n", r.location, r.length);
}
```

So, if you want to call this method from Python, you can do something like this:

```
from pyobjus.objc_py_types import NSRange
from pyobjus import autoclass

ObjcClass = autoclass('ObjcClass')
o_cls = ObjcClass.alloc().init()
range = NSRange(10, 20)
o_cls.useRange_(range)
```

This will output:

```
>>> location: 10, length: 20
```

A simmilar situation occurs when returning and using Objective C structure types. Let's say that our ObjcClass has another method with the name makeRange:

```
- (NSRange) makeRange {
    NSRange range;
    range.length = 123;
    range.location = 456;
    return range;
}
```

Using this method from Python is really simple. Let's say that we have included it from the previous Python code example:

```
range = o_cls.makeRange()
print range.length
print range.location
```

And this will output:

```
>>> 123
>>> 456
```

As you can see, dealing with Objective C structs from pyobjus is simple.

Let's see how to create a `NSRect` type:

```
point = NSPoint(30, 50)
size = NSSize(60, 70)
rect = NSRect(point, size)
```

4.3 Dealing with pointers

As you know, C has a very powerful feature with name pointers. Objective C is a superset of the C language, so Objective C also has this great feature.

But wait, we are using Python, so how we can deal with pointers from Python???

4.3.1 Passing pointers

Relax, pyobjus is doing that job for you. I think the best way to explain is to show some concrete examples of that. So, let's expand our ObjcClass class with another method:

```
- (void) useRangePtr:(NSRange*)r_p {
    NSRange r = r_p[0];
    printf("location: %ld, length: %ld\n", r.location, r.length);
}
```

In previous examples you have seen how to create an `NSRange` from Python, and you have sent values of the `NSRange` type. But now we have a situation when the method accepts a pointer to that type.

With pyobjus, you can call a method in the following way:

```
range = NSRange(40, 80)
o_cls.useRangePtr_(range)
```

And this will output:

```
>>> location:40, length: 80
```

So what has happened here? We pass the argument in the same way as with the `useRange:` method.

Pyobjus knows if a method accepts pointers to a type, or accepts values of that type. If a method accepts a pointer to a type, pyobjus will make a pointer to that type, point it to your type and pass that pointer to the method for you. So with this, you don't need to care whether argument types are pointers or values.

You can also return pointers to types from Objective C methods. Let's add another method to ObjcClass:

```
- (NSRange*) makeRangePtr {
    NSRange *r_p = malloc(sizeof(NSRange));
    NSRange r;
    r.length = 123;
    r.location = 567;
    *r_p = r;
    return r_p;
}
```

As you can see, this method creates a `NSRange` pointer, assigns a value to it, and at the end, it returns a pointer to the user. From Python, you can consume this method in this way:

```
range_ptr = o_cls.makeRangePtr()
# let we see actual type of returned object
print range_ptr
```

This will output following:

```
>>> <pyobjus.ObjcReferenceToType object at 0x10f34bcb0>
```

So here we can see another type -> ObjcReferenceToType. When we have a method which returns a pointer to some type, pyobjus will wrap that pointer with an ObjcReferenceToType object. This object contains the actual address of the C pointer. We can now pass that type to a function which accepts pointers.

Example:

```
# note that range_ptr is of ObjcReferenceToType type
o_cls.useRangePtr_(range_ptr)
```

But you may now wonder how to dereference the pointer to get the actual value?

The answer is. . . .by using the dereference function.

4.3.2 Dereferencing pointers

To dereference a pointer we use the dereference function:

```
from pyobjus import dereference
```

If a function returns a pointer to some known type, in other words, the return type isn't void*, you can use the dereference function in this way:

```
range_ptr = o_cls.makeRangePtr()
range = dereference(range_ptr)
```

Pyobjus will extract the return type from the method signature, and will thus know which type to convert the pointer value to. If it returns a void pointer, you will need to specify the type which you want pyobjus to convert the actual value to.

Consider adding this method:

```
- (void*) makeIntVoidPtr {
    int *a = malloc(sizeof(int));
    *a = 12345;
    return (void*)a;
}
```

Now we can retrieve the value and dereference it:

```
int_ptr = o_cls.makeIntVoidPtr()
int_val = dereference(int_ptr, of_type=ObjcInt)
print int_val
```

This will output with:

```
>>> 12345
```

Notice that you can specify the `of_type` optional argument even though the method returns a `NSRange` pointer. With this, you can be sure that pyobjus will convert the value to that type.

Here is the list of possible types:

```
'ObjcChar',
'ObjcInt',
'ObjcShort',
'ObjcLong',
'ObjcLongLong',
```

(continues on next page)

(continued from previous page)

```
'ObjcUChar',
'ObjcUInt',
'ObjcUShort',
'ObjcULong',
'ObjcULongLong',
'ObjcFloat',
'ObjcDouble',
'ObjcBool',
'ObjcBOOL',
'ObjcVoid',
'ObjcString',
'ObjcClassInstance',
'ObjcClass',
'ObjcSelector',
'ObjcMethod'
```

Those already listed types are defined inside the pyobjus module, so you can import them in the following way:

```
from pyobjus import ObjcChar, ObjcInt # etc...
```

Inside the `pyobjus.objc_py_types` module we define the struct and union types. Here is a list of them:

```
'NSRange',
'NSPoint',
'NSRect',
'NSSize'
```

You can import them with:

```
from pyobjus.objc_py_types import NSRange # etc...
```

4.4 Objective C <-> pyobjus literals

If you are familiar with Objective C literals, then you know that they are a great feature, because literals reduce the amount of code you write. You may wonder is there some equvivalent with pyobjus. The answer is YES.

The next example illustrates how to use pyobjus literals, and what their Objective C equivalents are:

```
from pyobjus import *

# The following examples demonstrate the pyobjus literals feature
# The first line denotes native objective c literals, and the second pyobjus literals
# SOURCE: http://clang.llvm.org/docs/ObjectiveCLiterals.html

# NSNumber *theLetterZ = @'Z';          // equivalent to [NSNumber numberWithChar:'Z']
objc_c('Z')

# NSNumber *fortyTwo = @42;             // equivalent to [NSNumber numberWithInt:42]
objc_i(42)

# NSNumber *fortyTwoUnsigned = @42U;    // equivalent to [NSNumber␣
↪numberWithUnsignedInt:42U]
objc_ui(42)
```

(continues on next page)

(continued from previous page)

```
# NSNumber *fortyTwoLong = @42L;          // equivalent to [NSNumber numberWithLong:42L]
objc_l(42)

# NSNumber *fortyTwoLongLong = @42LL;    // equivalent to [NSNumber␣
↪numberWithLongLong:42LL]
objc_ll(42)

# NSNumber *piFloat = @3.141592654F;     // equivalent to [NSNumber numberWithFloat:3.
↪141592654F]
objc_f(3.141592654)

# NSNumber *piDouble = @3.1415926535;    // equivalent to [NSNumber numberWithDouble:3.
↪1415926535]
objc_d(3.1415926535)

# NSNumber *yesNumber = @YES;            // equivalent to [NSNumber numberWithBool:YES]
objc_b(True)

# NSNumber *noNumber = @NO;              // equivalent to [NSNumber numberWithBool:NO]
objc_b(False)

# NSString *string = @"some string";
objc_str('some string')

# NSArray *array = @[ @"Hello", NSApp, [NSNumber numberWithInt:42] ];
objc_arr(objc_str('Hello'), objc_str('some str'), objc_i(42))

# NSDictionary *dictionary = @{
#     @"name" : NSUserName(),
#     @"date" : [NSDate date],
#     @"processInfo" : [NSProcessInfo processInfo]
# };
objc_dict({
    'name': objc_str('User name'),
    'date': autoclass('NSDate').date(),
    'processInfo': autoclass('NSProcessInfo').processInfo()
})
```

We have tried to make the build names for these literals clear and intuitive. We start with the prefix `objc_` followed by the letter/letters which denote the Objective C type. For example, i for `int`, f for `float`, arr for `NSArray`, dict for `NSDictionary`, etc...

4.5 Unknown types

Let's say that we have defined the following structures in our ObjcClass.

Note that we haven't specified a type name for the structs, so their types will be missing from any method signatures which use them:

```
typedef struct {
    float a;
    int b;
    NSRect rect;
} unknown_str_new;
```

(continues on next page)

(continued from previous page)

```
typedef struct {
    int a;
    int b;
    NSRect rect;
    unknown_str_new u_str;
} unknown_str;
```

Let's play. Suppose that we have defined the following Objective C methods:

```
- (unknown_str) makeUnknownStr {
    unknown_str str;
    str.a = 10;
    str.rect = NSMakeRect(20, 30, 40, 50);
    str.u_str.a = 2.0;
    str.u_str.b = 4;
    return str;
}
```

The purpose of this method is to create an unknown type struct and add some values to it's members. If you look at the debug logs from pyobjus, you will notice that the method returns the following type:

```
{?=ii{CGRect={CGPoint=dd}{CGSize=dd}}{?=fi{CGRect={CGPoint=dd}{CGSize=dd}}}}
```

From this we can see that method returns some type which contains two integers and two structs. One struct is a `CGRect`, and another is some unknown type which contains a float, an integer and a `CGRect` struct. So, if the user hasn't defined this struct, pyobjus can generate the type for them. Let's call this function:

```
ret_type = o_cls.makeUnknownStr()
```

But wait, how will pyobjus know about the field names in the struct, because from the method signature we see only types, not actual names? Well, pyobjus will generate some 'random' names in alphabetical order.

In our case, the first member will have the name 'a', the second the name 'b' and the third the name `CGRect`. `CGRect` is used because it can help the user as an indicator of actual type if it is missing. The last one is another unknown type, so pyobjus will generate the name 'c'.

Notice that in the case of the `CGRect`, it will have `origin` and `size` members because it is already defined so we know about these. This is not true for the last member, and pyobjus will thus choose the next alphabetical name for this member.

Perhaps you are asking yourself how you would know what the actual generated name is? Pyobjus will help you with this. There is a `getMembers` function which returns the name and types of some of the fields in the struct:

```
print ret_type.getMembers()
```

Python will output:

```
>>> [('a', <class 'ctypes.c_int'>), ('b', <class 'ctypes.c_int'>), ('CGRect', <class
↪'pyobjus.objc_py_types.NSRect'>), ('c', <class 'pyobjus.objc_py_types.UnknownType'>
↪)]
```

If you want to provide your field names, you can do it this way:

```
ret_type = o_cls.makeUnknownStr(members=['first', 'second', 'struct_field', 'tmp_field
↪'])
```

And if we now run the `getMembers` command, it will return this:

```
[('first', <class 'ctypes.c_int'>), ('second', <class 'ctypes.c_int'>), ('struct_field
↪', <class 'pyobjus.objc_py_types.NSRect'>), ('tmp_field', <class 'pyobjus.objc_py_
↪types.UnknownType'>)]
```

If you don't need types, only names, you can call method in following way:

```
print ret_type.getMembers(only_fields=True)
```

Python will output:

```
>>> ['a', 'b', 'CGRect', 'c']
```

Also, if you want to know only names, you can do that the following way:

```
print ret_type.getMembers(only_types=True)
```

Python will output:

```
>>> [<class 'ctypes.c_int'>, <class 'ctypes.c_int'>, <class 'pyobjus.objc_py_types.
↪NSRect'>, <class 'pyobjus.objc_py_types.UnknownType'>]
```

If you want to use the returned type to pass it as an argument to some function, there might be some problems. Pyobjus uses ctypes structures, so we can get the actual pointer to the C structure from Python objects, but if we want to get the correct values of the passed arg, we need to cast the pointer to the appropriate type.

If the type is defined in `pyobjus/objc_cy_types.pxi`, pyobjus will convert it for us, but if it isn't, we will need to convert it manually. For example, inside any Objective C methods where we are passing the struct value. Lets see an example of this:

```
- (void) useUnknownStr:(void*)str_vp {
    unknown_str *str_p = (unknown_str*)str_vp;
    unknown_str str = str_p[0];
    printf("%f\n", str.rect.origin.x);
}
```

And from Python:

```
o_cls.useUnknownStr_(ret_type)
```

And Python will output with:

```
>>> 20.00
```

4.6 Using class

As you know, `class` is a Python keyword, so that might be a problem.

Let's say that we want to get the Class type for an `NSString` instance...

We can use following:

```
NSString = autoclass('NSString')
text = NSString.alloc().init()
text.oclass()
```

This will return:

```
<pyobjus.ObjcClass object at 0x1057361b0>
```

So, now we can use the isKindOfClass: method:

```
text.isKindOfClass_(NSString)
```

This will output `True`. Let's see another example:

```
NSArray = autoclass('NSArray')
text.isKindOfClass_(NSArray)
```

And this will output `False`.

So, as you can see, if you want to use `class` with pyobjus, you will need to use the `some_object.oclass()` method.

4.7 Using @selector

There may be situations when you need to use `@selector`, which is an Objective C feature. With pyobjus you can also get the SEL type for a method. Let's say that we want to get the SEL for the init method:

```
from pyobjus import selector
selector('init')
```

This will output:

```
<pyobjus.ObjcSelector object at 0x1057361c8>
```

So, instead of using `@selector(init)` with Objective C, you can use `selector('init')` with pyobjus and Python to get the SEL type for that method (in this case the 'init' method).

If you want get the SEL for `initWithUTF8String:` you can use:

```
selector('initWithUTF8String:')
```

Other cases are the same for all methods.

4.8 Using @protocol

Objective C protocols provide what other languages call interfaces. They specify a list of methods which should be implemented in order to support that protocol.

Protocols define the interface which is then usually implemented by a delegate. Pyobjus provides us with the protocol decorator to handle this, enabling us to use Python objects as delegates:

```
@protocol('<protocol_name>')
```

Pyobjus will firstly try to use runtime introspection to determine the protocol methods. If this fails, it will revert to the list of protocols contained in the *pyobjus/protocols.py* file in your pyobjus checkout folder. Of course, many libraries define their own protocols, so cannot be included by default. For a complete list of protocols available on you system, run the *tools/build_protocols.py* file and then rebuild pyobjus (as per the install).

So, how do we use this decorator? We add functions with names that correspond to the protocol method names, then decorate these functions with the required protocol:

```
@protocol('NSURLConnectionDelegate')
def connection_didFailWithError_(self, connection, error):
```

Here, we specify that our object method *connection_didFailWithError_* handles the *connection:didFailWithError:* delegation of the *NSURLConnectionDelegate* protocol. Pyobjus then redirects this Objective-C message to our method.

For a complete example, please see the *examples/delegate.py* file.

4.9 Using enum types

Pyobjus currently supports `NSComparisonResult` and `NSStringEncoding` enums. If you want to use any others, you need to expand pyobjus with additional types by adding then to the `pyobjus/objc_py_types.py` file.

But, let's first see how to use the supported enum types with pyobjus. Consider the following example:

```
from pyobjus import autoclass, objc_str
from pyobjus.objc_py_types import NSComparisonResult

def enum_example():
    text = objc_str('some text')
    text_to_compare = objc_str('some text')
    if text.compare_(text_to_compare) == NSComparisonResult.NSOrderedSame:
        print 'the same strings'

    text_to_compare = objc_str('text')
    if text.compare_(text_to_compare) == NSComparisonResult.NSOrderedAscending:
        print 'NSOrderedAscending strings'

if __name__ == '__main__':
    enum_example()
```

You can see that we use the `NSComparisonResult` enum in the above example to compare two strings. The Enum is defined in this way:

```
NSComparisonResult = enum("NSComparisonResult", NSOrderedAscending=-1,␣
↪NSOrderedSame=0, NSOrderedDescending=1)
```

The first argument of the `enum` function is the name of new enum type, and the rest of the arguments are the field declarations of that enum. As you can see it is pretty simple to declare enum's with pyobjus, so you can add new enum types to pyobjus. Keep in mind you will bo re-compile pyobjus is order to see these changes in your Python environment.

4.10 Using vararg methods

Objective C supports vararg (Variable Arguments) methods, so it would be great if you could use vararg methods from pyobjus. Fortunately, you can.

Let's we say that we want to use the `arrayWithObjects:` method, which is a varargs method:

```
from pyobjus import autoclass, objc_str

NSArray = autoclass('NSArray')
array = NSArray.arrayWithObjects_(objc_str('first string'), objc_str('second string'),
↪ None)

text = array.objectAtIndex_(1)
print text.UTF8String()
```

Note that the last argument of a varargs methods must be `None`.

4.11 Using C array

In this section we will explain how to use a C array from pyobjus.

Let's say that we made a library `CArrayTestlib.dylib` which contains test functions for a C array. Let's load it:

```
import ctypes
from pyobjus import autoclass, selector, dereference, CArray, CArrayCount
from pyobjus.dylib_manager import load_dylib

load_dylib('CArrayTestlib.dylib', usr_path=False)
CArrayTestlib = autoclass("CArrayTestlib")
_instance = CArrayTestlib.alloc()
```

Now we can call the `setIntValues:` method:

```
- (void) setIntValues:(int[10])val_arr
{
    NSLog(@"Setting int array values...");
    memcpy(self->values, val_arr, sizeof(int) * 10);
    NSLog(@"Values copied...");
}
```

in this way:

```
nums = [0, 2, 1, 5, 4, 3, 6, 7, 8, 9]
array = (ctypes.c_int * 10)(*nums)
_instance.setIntValues_(array)
```

We can also return array values from this function:

```
- (int*) getIntValues
{
    if (!self->values)
    {
        NSLog(@"Values have not been set.");
        return NULL;
    }
    else
        return self->values;
}
```

and consume them this way:

```
returned_PyList = dereference(_instance.getIntValues(), of_type=CArray, return_
↪count=10)
print returned_PyList
```

Note that here we passing a `return_count` optional argument, which holds the number of array items which are returned from the `getIntValues` method.

But what if we don't know the array count? In that case we need to have some argument in which the method will put the array count value.

Consider following method:

```
- (int*) getIntValuesWithCount:(unsigned int*) n
{
    NSLog(@" ... ... [+] getIntValuesWithCount(n=%zd)", n);
    NSLog(@" ... ... [+] *n=%zd", *n);
    if (!self->values)
    {
        NSLog(@"Values have not been set");
        return NULL;
    }
    else
    {
        *n = 10;
        NSLog(@" ... ... [+] getIntValuesWithCount(n=%zd)", n);
        NSLog(@" ... ... [+] *n=%zd", *n);
        return self->values;
    }
}
```

The first argument of this function will contain the array count when the return statement is reached. So let's call it:

```
returned_PyList_withCount = dereference(_instance.getIntValuesWithCount_(CArrayCount),
↪ of_type=CArray)
print returned_PyList_withCount
```

Pyobjus will internally read from that argument and convert the returned C array into a python list.

If the method returns values/an array count over reference or you don't provide `CArrayCount` in the right position in the method signature, you will get an `IndexError: tuple index out of range` or segmentation fault, so don't forget to provide `CArrayCount` in the right position.

You may wonder, can you use multidimensional arrays from pyobjus? Yes, you can. Consider following method:

```
- (void) set2DIntValues: (int[10][10]) val_arr
{
    NSLog(@"Setting 2D int array values...");
    memcpy(self->int_2d_arr, val_arr, sizeof(int) * 10 * 10);
    NSLog(@"Values copied...");
}
- (int*) get2DIntValues
{
    if (!self->int_2d_arr)
    {
        NSLog(@"Values have not been set for int 2d array.");
        return NULL;
    }
    else
```

(continues on next page)

(continued from previous page)

```
    {
        return (int*)self->int_2d_arr;
    }
}
```

To call this method first we need to make a multidimensional array from python using nested lists:

```
twoD_array = [
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
    [21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
    [31, 32, 33, 34, 35, 36, 37, 38, 39, 40],
    [41, 42, 43, 44, 45, 46, 47, 48, 49, 50],
    [51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
    [61, 62, 63, 64, 65, 66, 67, 68, 69, 70],
    [71, 72, 73, 74, 75, 76, 77, 78, 79, 80],
    [81, 82, 83, 84, 85, 86, 87, 88, 89, 90],
    [91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
]
```

This represents an `int[10][10]` array, so let's call the above method:

```
_instance.set2DIntValues_(twoD_array)
returned_2d_list = dereference(_instance.get2DIntValues(), of_type=CArray,␣
↪partition=[10,10])
print returned_2d_list
```

Note the optional `partition` argument of the dereference function. This argument contains the format of the C array, in this case `[10, 10]`.

You can find additional examples on this link.

CHAPTER 5

Foundation framework examples

This part of the documentation covers examples of using classes from the Foundation framework.

5.1 NSArray example

Here is an example of using the NSArray class:

```
from pyobjus import autoclass

NSString = autoclass('NSString')
NSArray = autoclass("NSArray")

string_for_array = NSString.alloc().initWithUTF8String_("some text for NSArray")
array = NSArray.arrayWithObject_(string_for_array)

returnedObject = array.objectAtIndex_(0)
value = returnedObject.UTF8String()
contain_object = array.containsObject_(string_for_array)

returnedNSStringObject = array.objectAtIndex_(0)
value = returnedNSStringObject.UTF8String()

print "string value of returned object -->", value
print "return value of containsObject method -->", contain_object
```

This will output:

```
>>> string value of returned object --> some text for NSArray
>>> return value of containsObject method --> True
```

5.1.1 NSArray with pyobjus literals

If you want, you can use something like an Objective C literal to create an NSArray:

```
from pyobjus import objc_arr, objc_str

array = objc_arr(objc_str('some string'), objc_str('some other string'))
print array
```

As you can see here, `objc_arr(...)` is equivalent to `autoclass('NSArray').arrayWithObjects_(...)`, and will output:

```
>>> <__main__.__NSArrayI object at 0x10a22d350>
```

5.2 NSDictionary example

Here is an example of using a NSDictionary class:

```
from pyobjus import autoclass

NSString = autoclass('NSString')
NSArray = autoclass("NSArray")
NSDictionary = autoclass("NSDictionary")

string_object = NSString.stringWithUTF8String_("some text for NSDictionary")
string_key = NSString.stringWithUTF8String_("someKey")

array_object = NSArray.arrayWithObject_(string_object)
array_key = NSArray.arrayWithObject_(string_key)

# we are passing array with objects and keys
dictionary = NSDictionary.dictionaryWithObjects_forKeys_(array_object, array_key)

returned_nsstring = dictionary.objectForKey_(array_key.objectAtIndex_(0))
str_value = returned_nsstring.UTF8String()
print str_value
```

This will output:

```
>>> some text for NSDictionary
```

5.2.1 NSDictionary with pyobjus literals

We can also use pyobjus *literals* with the NSDictionary class.

So let's add two elements to the dictionary. The first one has the key 'first_key' and the value @'string value of first key', and the second one has the key 'second_key' with the value NSArray:

```
from pyobjus import objc_dict, objc_str, objc_arr

dictionary = objc_dict({
    'first_key': objc_str('string value of first key'),
    'second_key': objc_arr(objc_str('string element of NSArray'))
})
```

(continues on next page)

(continued from previous page)

```
first_key_value = dictionary.objectForKey_(objc_str('first_key'))
second_key_value = dictionary.objectForKey_(objc_str('second_key'))

print first_key_value
print second_key_value
```

This will output:

```
>>> <__main__.__NSCFString object at 0x101025d10>
>>> <__main__.__NSArrayI object at 0x101169290>
```

So, say you want to call the UTF8String method of the NSString object that resides in the 'first_key', you can simply call:

```
str_val = first_key_value.UTF8String()
print 'String value is: {0}'.format(str_val)
```

This will output:

```
>>> String value is: string value of first key
```

5.3 NSMutableArray example

This class is often useful if you need to add elements after you have created an array. So let's look at an example of using this class with pyobjus:

```
from pyobjus import autoclass

NSString = autoclass('NSString')
NSMutableArray = autoclass("NSMutableArray")

array = NSMutableArray.arrayWithCapacity_(5)
text_val_one = NSString.alloc().initWithUTF8String_("some text for NSMutableArray")
text_val_two = NSString.alloc().initWithUTF8String_("some other text for␣
↪NSMutableArray")

# we add some objects to NSMutableArray
array.addObject_(text_val_one)
array.addObject_(text_val_one)
array.addObject_(text_val_two)

count = array.count()
print "count of array before object delete -->", count

# then we remove some of them
array.removeObjectAtIndex_(0)
array.removeObject_(text_val_two)

count = array.count()
print "count of array after object delete -->", count

returnedObject = array.objectAtIndex_(0)
value = returnedObject.UTF8String()
```

(continues on next page)

(continued from previous page)

```
print "string value of returned object -->", value

# call method which accepts multiple arguments
array.insertObject_atIndex_(text_val_two, 1)
returnedObject = array.objectAtIndex_(1)
value = returnedObject.UTF8String()
print "string value of returned object at index 1 -->", value
```

This will output:

```
>>> number of array before object delete --> 3
>>> number of array after object delete --> 1
>>> string value of returned object --> some text for NSMutableArray
>>> string value of returned object at index 1 --> some other text for NSMutableArray
```

5.4 NSMutableDictionary example

As with the class above, you can also add and delete elements from the NSMutableDictionary after you've created it.:

```
from pyobjus import autoclass

NSString = autoclass('NSString')
NSMutableDictionary = autoclass("NSMutableDictionary")

# notice that you can instead of this line use objc_str('some text for NSDictoinary')
string_object = NSString.stringWithUTF8String_("some text for NSDictionary")
string_key = NSString.stringWithUTF8String_("someKey")

string_object_second = NSString.stringWithUTF8String_("some other text for␣
↪NSDictionary")
string_key_second = NSString.stringWithUTF8String_("someOtherKey")

objects_dict = {
    string_key: string_object,
    string_key_second: string_object_second
}

mutable_dictionary = NSMutableDictionary.dictionaryWithCapacity_(10)

# we can add objects to dict now
for key in objects_dict:
    mutable_dictionary.setObject_forKey_(objects_dict[key], key)

# let's return some object
returned_nsstring = mutable_dictionary.objectForKey_(string_key)

# we can iterate over dict values
enumerator = mutable_dictionary.objectEnumerator()
obj = enumerator.nextObject()
while obj:
    str_value = obj.UTF8String()
    print str_value
    obj = enumerator.nextObject()
```

So this will output:

```
>>> some other text for NSDictionary
>>> some text for NSDictionary
```

5.5 Other

Other examples can be found here.

CHAPTER 6

Pyobjus on iOS

You may be wondering how to run pyobjus on iOS devices? The solution to this problem is to use kivy-ios.

As you can see, kivy-ios contains scripts for building kivy, pyobjus and other libraries needed for running your app. It also provides scripts for making xcode projects from which you can run your python/kivy/pyobjus applications. Sounds great, and it is.

6.1 Example with Kivy UI

Let's first build kivy-ios. Execute following command:

```
git clone https://github.com/kivy/kivy-ios.git
cd kivy-ios
./toolchain.py build kivy pyobjus
```

This can take some time.

You can build your UI with the kivy framework, and access device hardware using pyobjus. So, let's look at one simple example of this. Notice that a tutorial describing how to use kivy-ios exists as part of the official kivy-ios documentation, but here we will provide another one, with focus on pyobjus.

Let's first make one simple example of using pyobjus with kivy.:

```
from pyobjus import autoclass, objc_f, objc_str
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.graphics import Color, Ellipse, Line

NSArray = autoclass('NSArray')
array = NSArray.arrayWithObjects_(objc_f(0.3), objc_f(1), objc_f(1), None)

class MyPaintWidget(Widget):
```

(continues on next page)

(continued from previous page)

```
    def on_touch_down(self, touch):
        color = (array.objectAtIndex_(0).floatValue(), array.objectAtIndex_(1).
↪floatValue(), array.objectAtIndex_(2).floatValue())
        with self.canvas:
            Color(*color, mode='hsv')
            d = 30.
            Ellipse(pos=(touch.x - d / 2, touch.y - d / 2), size=(d, d))
            touch.ud['line'] = Line(points=(touch.x, touch.y))

    def on_touch_move(self, touch):
        touch.ud['line'].points += [touch.x, touch.y]


class MyPaintApp(App):

    def build(self):
        parent = Widget()
        painter = MyPaintWidget()
        btn_text = objc_str('Clear')
        clearbtn = Button(text=btn_text.UTF8String())
        parent.add_widget(painter)
        parent.add_widget(clearbtn)

        def clear_canvas(obj):
            painter.canvas.clear()
        clearbtn.bind(on_release=clear_canvas)

        return parent

if __name__ == '__main__':
    MyPaintApp().run()
```

Please save this code inside a file with the name `main.py`. You will need to make a directory which will hold your python application code. For example, you can do the following:

```
mkdir ~/paint
mv main.py ~/paint
```

So now `paint` contains `main.py` file which holds your python code.

The example above is borrowed from this tutorial but we have added some pyobjus things to it. So we are now using a `NSArray` to store information about the line color, and we are using a `NSString` to set the text of the button.

Now you can create an xcode project which will hold our python application. kivy-ios comes with script for creating xcode projects for you. You only need to specify the project name and the absolute path to your app.

Execute the following command:

```
./toolchain.py create paintApp ~/paint/
```

Notice the following. The second parameter which we are passing to the script is the name of our app. In this case, the name of our iOS app will be *paintApp*. The third parameter is the absolute path to our python app which we want to run on iOS.

After executing this command you will get output similar to this:

```
-> Create /Users/myName/kivy-ios/paintApp-ios directory
-> Copy templates
-> Customize templates
-> Done !

Your project is available at /Users/myName/kivy-ios/paintapp-ios

You can now type: open /Users/myName/kivy-ios/paintapp-ios/paintapp.xcodeproj
```

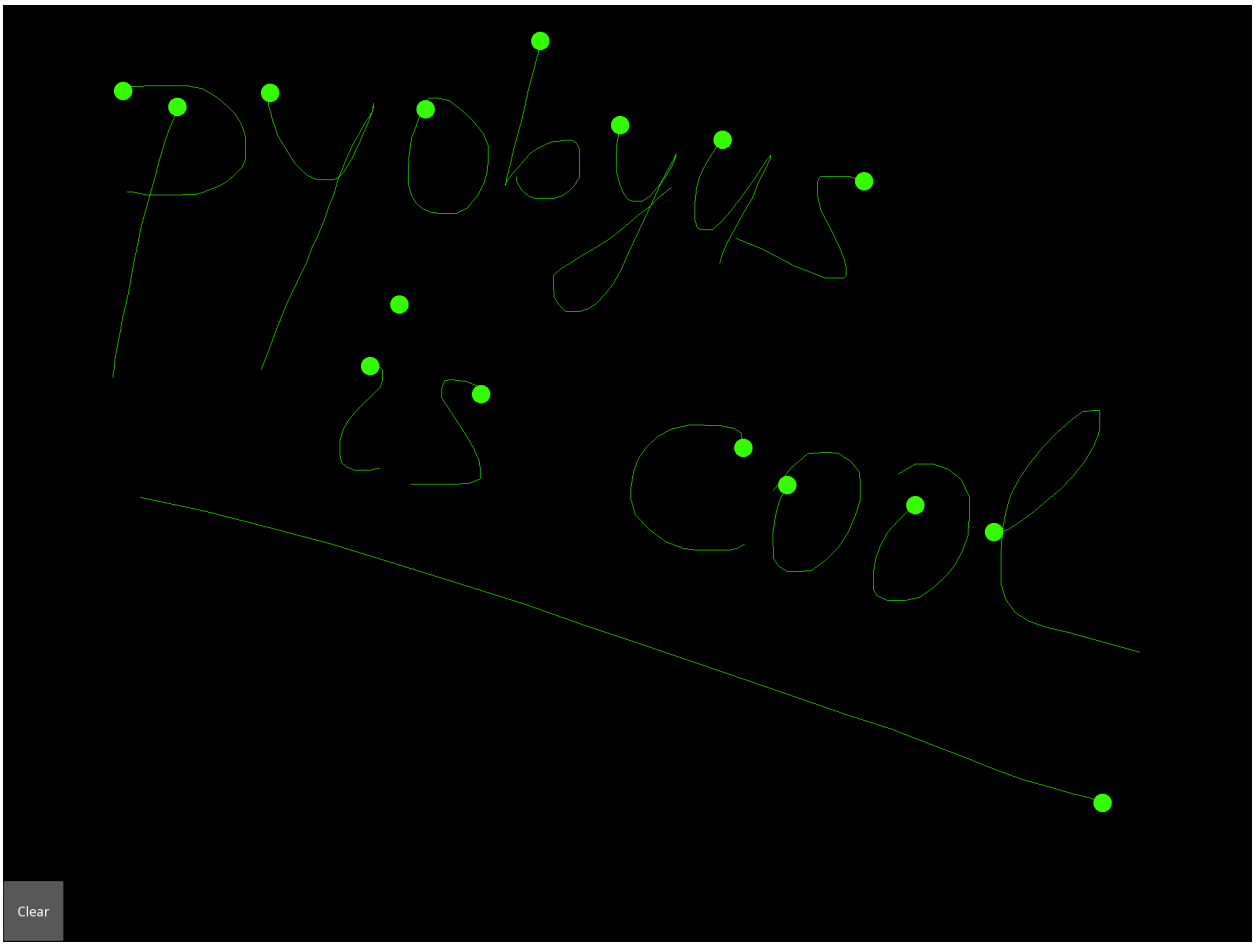
Note that the name is converted to lower case. If you enter the *paintapp-ios* directory you will see that there are `main.m`, `bridge.m` and other resources.

You can open this project with xcode as follows:

```
open /Users/myName/kivy-ios/paintapp-ios/paintapp.xcodeproj
```

If you have setup your developer account, you only need to click play and the app will be deployed on your iOS device.

This is screenshot from my iPad.



6.2 Accessing the accelerometer

To access the accelerometer on iOS devices, you use the CoreMotion framework. The CoreMotion framework is added by default in the project template which ships with kivy-ios.

Let's say that we have a class interface with the following properties and variables:

```
@interface bridge : NSObject {
    NSOperationQueue *queue;
}

@property (strong, nonatomic) CMMotionManager *motionManager;
@property (nonatomic) double ac_x;
@property (nonatomic) double ac_y;
@property (nonatomic) double ac_z;
@end
```

Also, let's say that we have an init method which inits the `motionManager` and the `queue`, and we have a method for running the accelerometer, and that method is declared as follows:

```
- (void)startAccelerometer {
    if ([self.motionManager isAccelerometerAvailable] == YES) {
        [self.motionManager startAccelerometerUpdatesToQueue:queue withHandler:^
↪(CMAccelerometerData *accelerometerData, NSError *error) {
            self.ac_x = accelerometerData.acceleration.x;
            self.ac_y = accelerometerData.acceleration.y;
            self.ac_z = accelerometerData.acceleration.z;
        }];
    }
}
```

You can see here that we are specifying a handler which will be called when we get some updates from the accelerometer. Currently you can't implement this handler from pyobjus, so that may be a problem.

But, we have solution for this. We have added a bridge class for this purpose: to implement handlers in pure Objective C, and then call methods of the bridge class so we can get the actual data in Python. In this example, we are storing the *x*, *y* and *z* values from the accelerometer in the `ac_x`, `ac_y` and `ac_z` class properties. We can then easily access these class properties.

So let's see a basic example how to read accelerometer data from pyobjus:

```
from pyobjus import autoclass

def run():
    Bridge = autoclass('bridge')
    br = Bridge.alloc().init()
    br.motionManager.setAccelerometerUpdateInterval_(0.1)
    br.startAccelerometer()

    for i in range(10000):
        print 'x: {0} y: {1} z: {2}'.format(br.ac_x, br.ac_y, br.ac_z)

if __name__ == "__main__":
    run()
```

So if you run this script on an iPad, in the way we have shown above, you'll get output similar to this in the xcode console:

```
x: 0.0219268798828 y: 0.111801147461 z: -0.976440429688
x: 0.0219268798828 y: 0.111801147461 z: -0.976440429688
x: 0.0219268798828 y: 0.111801147461 z: -0.976440429688
x: 0.0219268798828 y: 0.111801147461 z: -0.964920043945
x: 0.145629882812 y: -0.00624084472656 z: -0.964920043945
x: 0.145629882812 y: -0.00624084472656 z: -0.964920043945
x: 0.145629882812 y: -0.00624084472656 z: -0.964920043945
x: 0.145629882812 y: -0.00624084472656 z: -0.964920043945
```

As you can see, we have data from the accelerometer, so you can use it for some practical purposes if you want.

6.3 Accessing the gyroscope

In a similar way as we accessed the accelerometer, we can access the gyroscope. So let's expand our bridge class interface with properties which will hold gyro data:

```
@property (nonatomic) double gy_x;
@property (nonatomic) double gy_y;
@property (nonatomic) double gy_z;
```

Then in the bridge class implementation, add the following method:

```
- (void)startGyroscope {
    if ([self.motionManager isGyroAvailable] == YES) {
        [self.motionManager startGyroUpdatesToQueue:queue withHandler:^(CMGyroData␣
↪*gyroData, NSError *error) {
            self.gy_x = gyroData.rotationRate.x;
            self.gy_y = gyroData.rotationRate.y;
            self.gy_z = gyroData.rotationRate.z;
        }];
    }
}
```

This method is probably familiar to you because it is very similar to the method used for getting accelerometer data. Let's write some python code to read this data from python:

```
from pyobjus import autoclass

def run():
    Bridge = autoclass('bridge')
    br = Bridge.alloc().init()
    br.startGyroscope()

    for i in range(10000):
        print 'x: {0} y: {1} z: {2}'.format(br.gy_x, br.gy_y, br.gy_z)

if __name__ == "__main__":
    run()
```

You will get output similar to this:

```
x: 0.019542276079 y: 0.0267431973505 z: 0.00300590992237
x: 0.019542276079 y: 0.0267431973505 z: 0.00300590992237
x: 0.019542276079 y: 0.0267431973505 z: 0.00300590992237
x: 0.019542276079 y: 0.0267431973505 z: 0.00300590992237
```

(continues on next page)

(continued from previous page)

```
x: 0.019542276079 y: 0.0267431973505 z: 0.00300590992237
x: 0.019542276079 y: 0.018291389315 z: -0.00338913880323
x: 0.018301243011 y: 0.018291389315 z: -0.00338913880323
x: 0.018301243011 y: 0.018291389315 z: -0.00338913880323
x: 0.018301243011 y: 0.018291389315 z: -0.00338913880323
x: 0.018301243011 y: 0.018291389315 z: -0.00338913880323
x: 0.018301243011 y: 0.018291389315 z: -0.00338913880323
x: 0.0183009766949 y: 0.0170807162834 z: -0.00339499775763
x: 0.0183009766949 y: 0.0170807162834 z: -0.00339499775763
```

So now you can use gyro data in your Python kivy application.

6.4 Accessing the magnetometer

You can probably guess that this will be almost identical to the previous two examples. Let's add two new properties to the interface of the bridge class:

```
@property (nonatomic) double mg_x;
@property (nonatomic) double mg_y;
@property (nonatomic) double mg_z;
```

And then add the following method to the bridge class:

```
- (void)startMagnetometer {
    if (self.motionManager.magnetometerAvailable) {
        [self.motionManager startMagnetometerUpdatesToQueue:queue withHandler:^
↪(CMMagnetometerData *magnetometerData, NSError *error) {
            self.mg_x = magnetometerData.magneticField.x;
            self.mg_y = magnetometerData.magneticField.y;
            self.mg_z = magnetometerData.magneticField.z;
        }];
    }
}
```

Now we can use the methods above from pyobjus to get the data from the magnetometer:

```
from pyobjus import autoclass

def run():
    Bridge = autoclass('bridge')
    br = Bridge.alloc().init()
    br.startMagnetometer()

    for i in range(10000):
        print 'x: {0} y: {1} z: {2}'.format(br.mg_x, br.mg_y, br.mg_z)

if __name__ == "__main__":
    run()
```

You will get output similar to this:

```
x: 29.109375 y: -46.694519043 z: -27.4476470947
x: 29.109375 y: -46.694519043 z: -27.4476470947
x: 29.109375 y: -47.7679595947 z: -24.6468658447
```

(continues on next page)

(continued from previous page)

```
x: 28.03125 y: -47.7679595947 z: -24.6468658447
x: 28.03125 y: -47.7679595947 z: -24.6468658447
: 28.03125 y: -47.7679595947 z: -24.6468658447
x: 28.03125 y: -47.7679595947 z: -24.6468658447
x: 28.03125 y: -48.3046875 z: -27.4476470947
x: 27.4921875 y: -48.3046875 z: -27.4476470947
x: 27.4921875 y: -48.3046875 z: -27.4476470947
x: 27.4921875 y: -48.3046875 z: -27.4476470947
x: 27.4921875 y: -48.3046875 z: -27.4476470947
x: 27.4921875 y: -47.2312469482 z: -28.5679626
```

You can add additional bridge methods to your pyobjus iOS app by changing the content of the *bridge.m/.h* files, or by adding completely new files and classes to your xcode project. After that, you can consume them with pyobjus using the methods illustrated above.

6.5 Pyobjus-ball example

We've made a simple example using the accelerometer to control a ball on screen. In addition, with this example, you can set you screen brightness using a kivy slider.

We won't go into the details of the kivy language or kivy itself as you can find excellent examples and docs on the official kivy site.

So, here is the code of the `main.py` file:

```
from random import random
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.properties import NumericProperty, ReferenceListProperty, ObjectProperty
from kivy.vector import Vector
from kivy.clock import Clock
from kivy.graphics import Color
from pyobjus import autoclass

class Ball(Widget):

    velocity_x = NumericProperty(0)
    velocity_y = NumericProperty(0)
    h = NumericProperty(0)
    velocity = ReferenceListProperty(velocity_x, velocity_y)

    def move(self):
        self.pos = Vector(*self.velocity) + self.pos

class PyobjusGame(Widget):

    ball = ObjectProperty(None)
    screen = ObjectProperty(autoclass('UIScreen').mainScreen())
    bridge = ObjectProperty(autoclass('bridge').alloc().init())
    sensitivity = ObjectProperty(50)
    br_slider = ObjectProperty(None)

    def __init__(self, *args, **kwargs):
        super(PyobjusGame, self).__init__()
        self.bridge.startAccelerometer()
```

(continues on next page)

(continued from previous page)

```
    def __dealloc__(self, *args, **kwargs):
        self.bridge.stopAccelerometer()
        super(PyobjusGame, self).__dealloc__()

    def reset_ball_pos(self):
        self.ball.pos = self.width / 2, self.height / 2

    def on_bright_slider_change(self):
        self.screen.brightness = self.br_slider.value

    def update(self, dt):
        self.ball.move()
        self.ball.velocity_x = self.bridge.ac_x * self.sensitivity
        self.ball.velocity_y = self.bridge.ac_y * self.sensitivity

        if (self.ball.y < 0) or (self.ball.top >= self.height):
            self.reset_ball_pos()
            self.ball.h = random()

        if (self.ball.x < 0) or (self.ball.right >= self.width):
            self.reset_ball_pos()
            self.ball.h = random()


class PyobjusBallApp(App):

    def build(self):
        game = PyobjusGame()
        Clock.schedule_interval(game.update, 1.0/60.0)
        return game


if __name__ == '__main__':
    PyobjusBallApp().run()
```

And the contents of `pyobjusball.kv` are:

```
<Ball>:
    size: 50, 50
    h: 0
    canvas:
        Color:
            hsv: self.h, 1, 1,
        Ellipse:
            pos: self.pos
            size: self.size

<PyobjusGame>:
    ball: pyobjus_ball
    br_slider: bright_slider

    Label:
        text: 'Screen brightness'
        pos: bright_slider.x, bright_slider.y + bright_slider.height / 2
    Slider:
        pos: self.parent.width / 4, self.parent.height / 1.1
```

(continues on next page)

(continued from previous page)

```
        id: bright_slider
        value: 0.5
        max: 1
        min: 0
        width: self.parent.width / 2
        height: self.parent.height / 10
        on_touch_up: root.on_bright_slider_change()

    Ball:
        id: pyobjus_ball
        center: self.parent.center
```

Now create a directory with the name `pyobjus-ball` and place the files above in it:

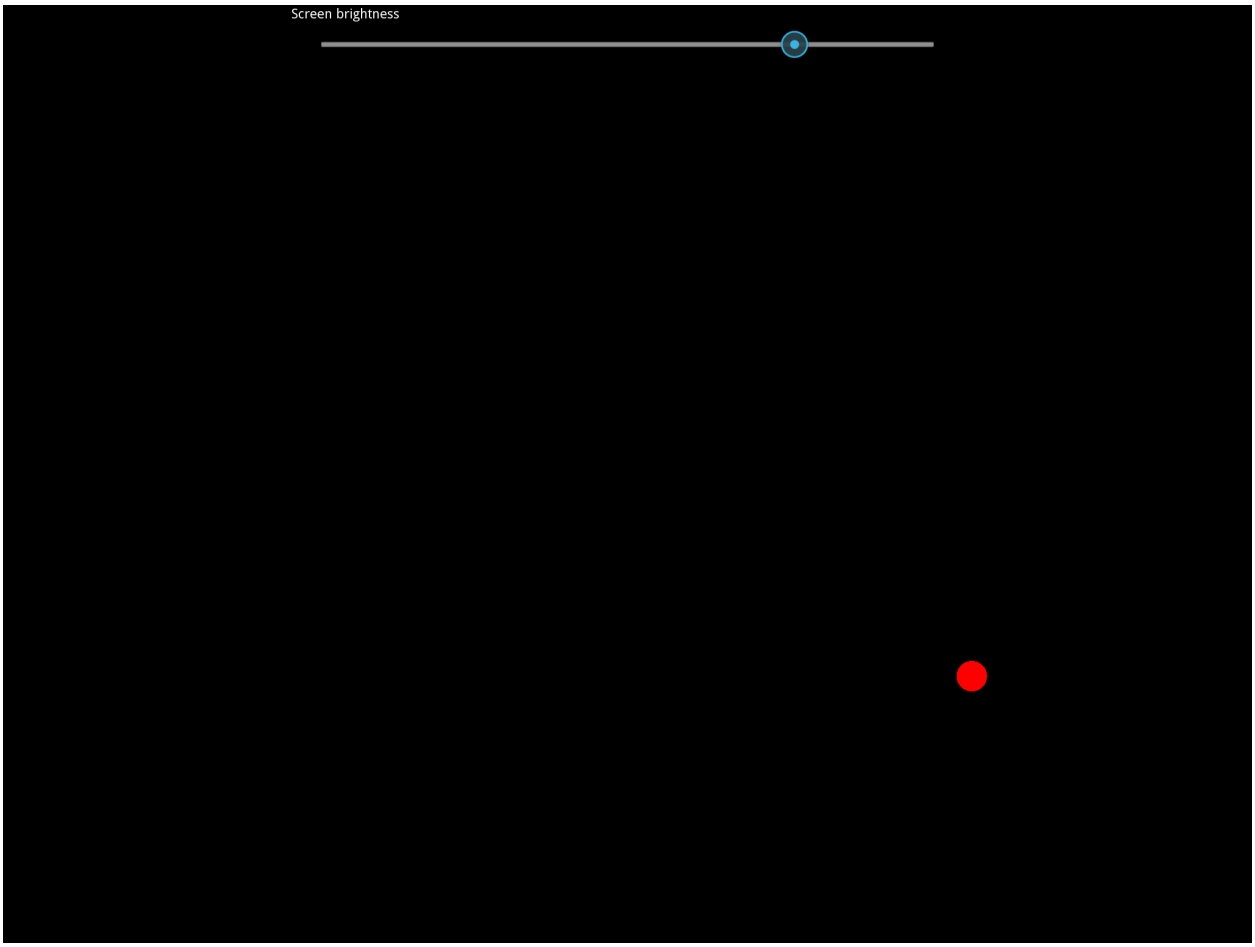
```
mkdir pyobjus-ball
mv main.py pyobjus-ball
mv pyobjusball.kv pyobjus-ball
```

In this step, we assume that you have already have downloaded and built `kivy-ios`. Navigate to the directory where `kivy-ios` is located, then execute the following commands:

```
tools/create-xcode-project.sh pyobjusBall /path/to/pyobjus-ball
open app-pyobjusball/pyobjusball.xcodeproj/
```

After this step, xcode will open and, if you have connected your iOS device to your computer, you can run the project and will see your app running on your device.

This is a screenshot from an iPad.


Screen brightness

CHAPTER 7

API

This part of the documentation covers all the interfaces of Pyobjus.

7.1 Reflection functions

pyobjus.**autoclass**(*name*[, *copy_properties=None*, *load_class_methods=None*, *load_instance_methods=None*, *reset_autoclass=None*])

Get and load an Objective C class

Parameters

- **name** – Name of the Objective C class which you want to load.
- **copy_properties** (*None or Boolean*) – Denotes whether to copy the properties of the Objective C class or not. The default is to copy all properties.
- **load_class_methods** (*None or List*) – If this argument is omitted or *None*, all class methods will be loaded. You can use it to force only certain class methods to be loaded eg. *load_class_methods=['alloc']*.
- **load_instance_methods** (*None or List*) – If this argument is omitted or set to *None*, all instance methods will be loaded. You can use it force only instance methods to be loaded, eg. *load_instance_methods=['init']*.
- **reset_autoclass** (*None or Boolean*) – If this argument is set to True and the class was previously loaded with a restricted subset of methods, when you call the autoclass function again with this argument for the same class, all the methods will be loaded.

Return type Return a ObjClass that represents the class passed from *name*.

```
>>> from pyobjus import autoclass
>>> autoclass('NSString')
<class '__main__.NSString'>
```

7.2 Utility functions

pyobjus.**selector**(*objc_method*)
Get the selector for the method specified by the objc_method parameter

Parameters **objc_method** (*String*) – Name of the Objective C method for which we want to get the SEL.

Return type *ObjcSelector*, which is a Python representation for the Objective C SEL type.

pyobjus.**dereference**(*objc_reference*[, *of_type=None*, *return_count=None*, *partition=None*])
Dereference the C pointer to get the actual values

Parameters

- **objc_reference** – *ObjcReferenceToType* Python representation of the C pointer.
- **of_type** – If the function which you call returns a value, for example an int, float, etc., pyobjus can determine the type which to convert it to. But if you return a void pointer for eg. then you need to specify the type to which you want to convert it. An example of this is: *dereference(someObjcReferenceToType, of_type=ObjcInt)*.
- **return_count** (*Integer*) – When you are returning a C array, you can/need specify number of returned values with this argument.
- **partition** – When you want to dereference a multidimensional array, you need to specify it's dimensions. Provide a list with numbers which denote it's dimensions. For example, with *int array[10][10]*, you need to specify *partition=[10, 10]*.

Return type Actual value for some *ObjcReferenceToType* type.

pyobjus.**objc_c**(*some_char*)
Initialize *NSNumber* with a *Char* type.

Parameters **some_char** – Char parameter

Return type NSNumber.numberWithChar: Python representation

pyobjus.**objc_i**(*some_int*)
Initialize *NSNumber* with an *Int* type.

Parameters **some_int** – Int parameter

Return type NSNumber.numberWithInt: Python representation

pyobjus.**objc_ui**(*some_unsigned_int*)
Initialize *NSNumber* with an *Unsigned Int* type.

Parameters **some_unsigned_int** – Unsigned Int parameter

Return type NSNumber.numberWithUnsignedInt: Python representation

pyobjus.**objc_l**(*some_long*)
Initialize *NSNumber* with a *Long* type.

Parameters **some_char** – Long parameter

Return type NSNumber.numberWithLong: Python representation

pyobjus.**objc_ll**(*some_long_long*)
Initialize *NSNumber* with a *Long Long* type.

Parameters **some_long_long** – Long Long parameter

Return type NSNumber.numberWithLongLong: Python representation

pyobjus.**objc_f**(*some_float*)
Initialize *NSNumber* with a *Float* type.

Parameters **some_float** – Float parameter

Return type NSNumber.numberWithFloat: Python representation

pyobjus.**objc_d**(*some_double*)
Initialize *NSNumber* with a *Double* type.

Parameters **some_double** – Double parameter

Return type NSNumber.numberWithDouble: Python representation

pyobjus.**objc_b**(*some_bool*)
Initialize *NSNumber* with a *Bool* type.

Parameters **some_char** – Bool parameter

Return type NSNumber.numberWithBool: Python representation

pyobjus.**objc_str**(*some_string*)
Initialize *NSNumber* with a *NSString* type.

Parameters **some_string** – String parameter

Return type NSString.stringWithUTF8String: Python representation

pyobjus.**objc_arr**(*some_array*)
Initialize a *NSArray* type

Parameters **some_array** – List of parameters. For eg:

```
objc_arr(objc_str('Hello'), objc_str('some str'), objc_i(42))
```

Return type NSArray Python representation

pyobjus.**objc_dict**(*some_dict*)
Initialize a *NSDictionary* type

Parameters **some_dict** – Dict parameter. For eg:

```
objc_dict({
    'name': objc_str('User name'),
    'date': autoclass('NSDate').date(),
    'processInfo': autoclass('NSProcessInfo').processInfo()
})
```

Return type NSDictionary Python representation

7.3 Global variables

pyobjus.**dev_platform**
Platform for which pyobjus is compiled

7.4 Pyobjus Objective C types

class pyobjus.**ObjcChar**
Objective C `char` representation

class pyobjus.**ObjcInt**
Objective C `int` representation

class pyobjus.**ObjcShort**
Objective C `short` representation

class pyobjus.**ObjcLong**
Objective C `long` representation

class pyobjus.**ObjcLongLong**
Objective C `long long` representation

class pyobjus.**ObjcUChar**
Objective C `unsigned char` representation

class pyobjus.**ObjcUInt**
Objective C `unsigned int` representation

class pyobjus.**ObjcUShort**
Objective C `unsigned short` representation

class pyobjus.**ObjcULong**
Objective C `unsigned long` representation

class pyobjus.**ObjcULongLong**
Objective C `unsigned long long` representation

class pyobjus.**ObjcFloat**
Objective C `float`` representation

class pyobjus.**ObjcDouble**
Objective C `double` representation

class pyobjus.**ObjcBool**
Objective C `bool` representation

class pyobjus.**ObjcBOOL**
Objective C `BOOL` representation

class pyobjus.**ObjcVoid**
Objective C `void` representation

class pyobjus.**ObjcString**
Objective C `char*` representation

class pyobjus.**ObjcClassInstance**
Representation of an Objective C class instance

class pyobjus.**ObjcClass**
Representation of an Objective C `Class`

class pyobjus.**ObjcSelector**
Representation of an Objective C `SEL`

class pyobjus.**ObjcMethod**
Representation of an Objective C method

class pyobjus.**CArray**
Representation of an Objective C (C) array

class pyobjus.**CArrayCount**
Representation of a type which holds `outCount*` for some C array -> number of received array elements

exception pyobjus.**ObjcException**
Representation of some Objective C exception

7.5 Structure types

class pyobjus.objc_py_types.**NSRange**

unsigned long long **location**

unsigned long long **length**

class pyobjus.objc_py_types.**NSPoint**

double **x**

double **y**

class pyobjus.objc_py_types.**NSSize**

double **width**

double **height**

class pyobjus.objc_py_types.**NSRect**

NSPoint **origin**

NSSize **size**

7.6 Enumeration types

class pyobjus.objc_py_types.**NSComparisonResult**

NSOrderedAscending = -1

NSOrderedSame = 0

NSOrderedDescending = 1

class pyobjus.objc_py_types.**NSStringEncoding**

NSASCIIStringEncoding = 1

NSNEXTSTEPStringEncoding = 2

NSJapaneseEUCStringEncoding = 3

NSUTF8StringEncoding = 4

NSISOLatin1StringEncoding = 5

NSSymbolStringEncoding = 6

NSNonLossyASCIIStringEncoding = 7

NSShiftJISStringEncoding = 8

NSISOLatin2StringEncoding = 9

NSUnicodeStringEncoding = 10

NSWindowsCP1251StringEncoding = 11

NSWindowsCP1252StringEncoding = 12

NSWindowsCP1253StringEncoding = 13

NSWindowsCP1254StringEncoding = 14

NSWindowsCP1250StringEncoding = 15

NSISO2022JPStringEncoding = 21

NSMacOSRomanStringEncoding = 30

NSUTF16StringEncoding = 10

NSUTF16BigEndianStringEncoding = 0x90000100

NSUTF16LittleEndianStringEncoding = 0x94000100

NSUTF32StringEncoding = 0x8c000100

NSUTF32BigEndianStringEncoding = 0x98000100

NSUTF32LittleEndianStringEncoding = 0x9c000100

NSProprietaryStringEncoding = 65536

7.7 Dynamic library manager

pyobjus.dylib_manager.**load_dylib**(*path*)
Function for loading a user defined dylib.

Parameters **path** – Path to some dylib.

pyobjus.dylib_manager.**make_dylib**(*path*[, *frameworks=None*, *out=None*, *options=None*])
Function for making a dylib from Python.

Parameters

- **path** – Path to the file.
- **frameworks** (*List*) – List of frameworks to link with.
- **options** (*List*) – List of options for the compiler
- **out** – Out location. The default is to write to the location specified by the path argument.

pyobjus.dylib_manager.**load_framework**(*framework*)
Function that loads an Objective C framework via NSBundle.

Parameters **framework** (*String*) – Path to the framework.

Raises ObjcException if the framework can't be found.

7.8 Objective-C signature format

Objective C signatures have a special format that can be difficult to understand at first. Let's look into the details. A signature is in the format:

```
<return type><offset0><argument1><offset1><argument2><offset2><...>
```

The offset represents how many bytes the previous argument is from the start of the allocated memory.

All the types for any part of the signature can be one of:

- c = represent a char
- i = represent an int
- s = represent a short
- l = represent a long (l is treated as a 32-bit quantity on 64-bit programs.)
- q = represent a long long
- c = represent an unsigned char
- i = represent an unsigned int
- s = represent an unsigned short
- l = represent an unsigned long
- q = represent an unsigned long long
- f = represent a float
- d = represent a double
- b = represent a c++ bool or a c99 _bool
- v = represent a void
- * = represent a character string (char *)
- @ = represent an object (whether statically typed or typed id)
- # = represent a class object (class)
- : = represent a method selector (sel)
- [array type] = represent an array
- {name=type...} = represent a structure
- (name=type...) = represent a union
- bnum = represent a bit field of num bits
- ^ = represent type a pointer to type
- ? = represent an unknown type (among other things, this code is used for function pointers)

CHAPTER 8

Remarks

8.1 Missing things

- Support for bit fields -> libffi restriction
- Support for passing unions by value -> libffi restriction

8.2 Issues

- Currently there is only support for few structure, union and enumeration types.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

W

X

Y