Introduction to Deferreds

This document introduces `Deferred`s, Twisted's preferred mechanism for controlling the flow of asynchronous code. Don't worry if you don't know what that means yet - that's why you are here!

It is intended for newcomers to Twisted, and was written particularly to help people read and understand code that already uses `Deferred`s.

This document assumes you have a good working knowledge of Python. It assumes no knowledge of Twisted.

By the end of the document, you should understand what `Deferred`s are and how they can be used to coordinate asynchronous code. In particular, you should be able to:

- Read and understand code that uses `Deferred`s
- Translate from synchronous code to asynchronous code and back again
- Implement any sort of error-handling for asynchronous code that you wish

The joy of order¶

When you write Python code, one prevailing, deep, unassailled assumption is that a line of code within a block is only ever executed after the preceding line is finished.

```
pod_bay_doors.open()
pod.launch()
```

The pod bay doors open, and only *then* does the pod launch. That's wonderful. One-line-after-another is a built-in mechanism in the language for encoding the order of execution. It's clear, terse, and unambiguous.

Exceptions make things more complicated. If `pod_bay_doors.open()` raises an exception, then we cannot know with certainty that it completed, and so it would be wrong to proceed blithely to the next line. Thus, Python gives us `try`, `except`, `finally`, and `else`, which together model almost every conceivable way of handling a raised exception, and tend to work really well.

Function application is the other way we encode order of execution:

```
pprint(sorted(x.get_names()))
```

First `x.get_names()` gets called, then `sorted` is called with its return value, and then `pprint` with whatever `sorted` returns.

It can also be written as:

```
names = x.get_names()
sorted_names = sorted(names)
pprint(sorted_names)
```

Sometimes it leads us to encode the order when we don't need to, as in this example:

```
total = 0
for account in accounts:
    total += account.get_balance()
print "Total balance $%s" % (total,)
```

But that's normally not such a big deal.

All in all, things are pretty good, and all of the explanation above is laboring familiar and obvious points. One line comes after another and one thing happens after another, and both facts are inextricably tied.

But what if we had to do it differently?

A hypothetical problem

What if we could no longer rely on the previous line of code being finished (whatever that means) before we started to interpret & execute the next line of code? What if `pod_bay_doors.open()` returned immediately, triggering something somewhere else that would eventually open the pod bay doors, recklessly sending the Python interpreter plunging into `pod.launch()` ?

That is, what would we do if the order of execution did not match the order of lines of Python? If "returning" no longer meant "finishing"?

Asynchronous operations?

How would we prevent our pod from hurtling into the still-closed doors? How could we respond to a potential failure to open the doors at all? What if opening the doors gave us some crucial information that we needed in order to launch the pod? How would we get access to that information?

And, crucially, since we are writing code, how can we write our code so that we can build *other* code on top of it?

The components of a solution

We would still need a way of saying "do *this* only when *that* has finished".

We would need a way of distinguishing between successful completion and interrupted processing, normally modeled with `try`, `expect`, `else`, and `finally`.

We need a mechanism for getting return failures and exception information from the thing that just executed to the thing that needs to happen next.

We need somehow to be able to operate on results that we don't have yet. Instead of acting, we need to make and encode plans for how we would act if we could.

Unless we hack the interpreter somehow, we would need to build this with the Python language constructs we are given: methods, functions, objects, and the like.

Perhaps we want something that looks a little like this:

```
placeholder = pod_bay_doors.open()
placeholder.when_done(pod.launch)
```

One solution: Deferred

Twisted tackles this problem with `Deferred`s, a type of object designed to do one thing, and one thing only: encode an order of execution separately from the order of lines in Python source code.

It doesn't deal with threads, parallelism, signals, or subprocesses. It doesn't know anything about an event loop, greenlets, or scheduling. All it knows about is what order to do things in. How does it know that? Because we explicitly tell it the order that we want.

Thus, instead of writing:

```
pod_bay_doors.open()
pod.launch()
```

We write:

```
d = pod_bay_doors.open()
d.addCallback(lambda ignored: pod.launch())
```

That introduced a dozen new concepts in a couple of lines of code, so let's break it down. If you think you've got it, you might want to skip to the next section.

Here, `pod_bay_doors.open()` is returning a `Deferred`, which we assign to `d`. We can think of `d` as a placeholder, representing the value that `open()` will eventually return when it finally gets around to finishing.

To "do this next", we add a *callback* to `d`. A callback is a function that will be called with whatever `open()` eventually returns. In this case, we don't care, so we make a function with a single, ignored parameter that just calls `pod.launch()`.

So, we've replaced the "order of lines is order of execution" with a deliberate, in-Python encoding of the order of execution, where `d` represents the particular flow and `d.addCallback` replaces "new line".

Of course, programs generally consist of more than two lines, and we still don't know how to deal with failure.

Getting it right: The failure cases

In what follows, we are going to take each way of expressing order of operations in normal Python (using lines of code and `try/except`) and translate them into an equivalent code built with `Deferred` objects.

This is going to be a bit painstaking, but if you want to really understand how to use `Deferred`s and maintain code that uses them, it is worth understanding each example below.

One thing, then another, then another

Recall our example from earlier:

```
pprint(sorted(x.get_names()))
```

Also written as:

```
names = x.get_names()
sorted_names = sorted(names)
pprint(sorted_names)
```

What if neither `get_names` nor `sorted` can be relied on to finish before they return? That is, if both are asynchronous operations?

Well, in Twisted-speak they would return `Deferred`s and so we would write:

```
d = x.get_names()
d.addCallback(sorted)
d.addCallback(pprint)
```

Eventually, `sorted` will get called with whatever `get_names` finally delivers. When `sorted` finishes, `pprint` will be called with whatever it delivers.

We could also write this as:

```
x.get_names().addCallback(sorted).addCallbac
```

Since `d.addCallback` returns `d`.

Simple failure handling

We often want to write code equivalent to this:

```
try:
    x.get_names()
except Exception, e:
    report_error(e)
```

How would we write this with `Deferred`s?

```
d = x.get_names()
d.addErrback(report_error)
```

errback is the Twisted name for a callback that is called when an error is received.

This glosses over an important detail. Instead of getting the exception object `e`, `report_error` would get a `Failure` object, which has all of the useful information that `e` does, but is optimized for use with `Deferred`s.

We'll dig into that a bit later, after we've dealt with all of the other combinations of exceptions.

Handle an error, but do something else on success

What if we want to do something after our `try` block if it actually worked? Abandoning our contrived examples and reaching for generic variable names, we get:

```
try:
    y = f()
except Exception, e:
    g(e)
else:
    h(y)
```

Well, we'd write it like this with `Deferred`s:

```
d = f()
d.addCallbacks(h, g)
```

Where `addCallbacks` means "add a callback and an errback at the same time". `h` is the callback, `g` is the errback.

Now that we have `addCallbacks` along with `addErrback` and `addCallback`, we can match any possible combination of `try`, `except`, `else`, and `finally` by varying the order in which we call them. Explaining exactly how it works is tricky (although the *Deferred reference* does rather a good job), but once we're through all of the examples it ought to be clearer.

Handle an error, then proceed anyway

What if we want to do something after our `try/except` block, regardless of whether or not there was an exception? That is, what if we wanted to do the equivalent of this generic code:

```
try:
    y = f()
except Exception, e:
    y = g(e)
h(y)
```

And with `Deferred`s:

```
d = f()
d.addErrback(g)
d.addCallback(h)
```

Because `addErrback` returns `d`, we can chain the calls like so:

```
f().addErrback(g).addCallback(h)
```

The order of `addErrback` and `addCallback` matters. In the next section, we can see what would happen when we swap them around.

Handle an error for the entire operation

What if we want to wrap up a multi-step operation in one exception handler?

```
try:
    y = f()
    z = h(y)
except Exception, e:
    g(e)
```

With `Deferred`s, it would look like this:

```
d = f()
d.addCallback(h)
d.addErrback(g)
```

Or, more succinctly:

```
d = f().addCallback(h).addErrback(g)
```

Do something regardless

What about `finally`? How do we do something regardless of whether or not there was an exception? How do we translate this:

```
try:
    y = f()
finally:
    g()
```

Well, roughly we do this:

```
d = f()
d.addBoth(g)
```

This adds `g` as both the callback and the errback. It is equivalent to:

```
d.addCallbacks(g, g)
```

Why "roughly"? Because if `f` raises, `g` will be passed a `Failure` object representing the exception. Otherwise, `g` will be passed the asynchronous equivalent of the return value of `f()` (i.e. `y`).

Inline callbacks - using 'yield'

Twisted features a decorator named `inlineCallbacks` which allows you to work with Deferreds without writing callback functions.

This is done by writing your code as generators, which *yield* `Deferreds` instead of attaching callbacks.

Consider the following function written in the traditional `Deferred` style:

```
def getUsers():
    d = makeRequest("GET", "/users")
    d.addCallback(json.loads)
    return d
```

using `inlineCallbacks`, we can write this as:

```
from twisted.internet.defer import inlineCal

@inlineCallbacks
def getUsers(self):
    responseBody = yield makeRequest("GET", '
    returnValue(json.loads(responseBody))
```

a couple of things are happening here:

1. instead of calling `addCallback` on the `Deferred` returned by `makeRequest`, we *yield* it. This causes Twisted to return the `Deferred`'s result to us.
2. we use `returnValue` to propagate the final result of our function. Because this function is a generator, we cannot use the return statement; that would be a syntax error.

Note

New in version 15.0.

On Python 3.3 and above, instead of writing

`returnValue(json.loads(responseBody))` you can instead write `return json.loads(responseBody)`. This can be a significant readability advantage, but unfortunately if you need compatibility with Python 2, this isn't an option.

Both versions of `getUsers` present exactly the same API to their callers: both return a `Deferred` that fires with the parsed JSON body of the request. Though the `inlineCallbacks` version looks like synchronous code, which blocks while waiting for the request to finish, each `yield` statement allows other code to run while waiting for the `Deferred` being yielded to fire.

`inlineCallbacks` become even more powerful when dealing with complex control flow and error handling. For example, what if `makeRequest` fails due to a connection error? For the sake of this example, let's say we want to log the exception and return an empty list.

```
def getUsers():
    d = makeRequest("GET", "/users")

    def connectionError(failure):
        failure.trap(ConnectionError)
        log.failure("makeRequest failed due to
                    failure)
        return []

    d.addCallbacks(json.loads, connectionErro
    return d
```

With `inlineCallbacks`, we can rewrite this as:

```
@inlineCallbacks
def getUsers(self):
    try:
        responseBody = yield makeRequest("GE
    except ConnectionError:
        log.failure("makeRequest failed due to
        returnValue([])

    returnValue(json.loads(responseBody))
```

Our exception handling is simplified because we can use Python's familiar `try` / `except` syntax for handling `ConnectionError`s.

Conclusion

You have been introduced to asynchronous code and have seen how to use `Deferred`s to:

- Do something after an asynchronous operation completes successfully
- Use the result of a successful asynchronous operation

- Catch errors in asynchronous operations
- Do one thing if an operation succeeds, and a different thing if it fails
- Do something after an error has been handled successfully
- Wrap multiple asynchronous operations with one error handler
- Do something after an asynchronous operation, regardless of whether it succeeded or failed
- Write code without callbacks using `inlineCallbacks`

These are very basic uses of `Deferred`. For detailed information about how they work, how to combine multiple Deferreds, and how to write code that mixes synchronous and asynchronous APIs, see the *Deferred reference*. Alternatively, read about how to write functions that *generate Deferreds*.