Twisted & Monads

Twisted

Asynchronous networking framework in Python

Asynchronous

Non-blocking IO

Blocking IO

```
line = remote.get_line()
```

Block a whole line is returned

It's synchronous

```
line = remote.get_line()
```

Want concurrency?

Use threads.

Not Twisted though

HATE HATE HATE

GIL

Correctness

Instead...

Cooperative multitasking

Non-blocking IO

Little chunks of IO…

…that yield control…

…to a central loop…

…and fire events when done.

The Reactor

```
select();
```

The code?

```
remote.get_line(
  get_line_callback)
```

uerrghh

```
□ = remote.get_line()
```

A placeholder

Promise

Future Value

Deferred

Mochikit

What do you do with □?

Lotto ticket

Make plans

□ → THIS

THIS(x) :: <something>

```
□.addCallback(THIS)
```

Not much better than passing a callback

Chaining

$\Box \rightarrow \text{THIS} \rightarrow \text{THAT}$

Synchronously...

THAT(THIS(□))

(THAT . THIS) □

Asynchronously...

- `.addCallback(THIS)`
- `.addCallback(THAT)`

```
□.addCallback(THIS
  ).addCallback(THAT)
```

Explicitly encoding ordering of actions

Callbacks can return Deferreds

```
def THIS(line):
    d = remote.get_line()
    return d.addCallback(
        lambda line2: (line1, line2))
```

```
def THAT((line1, line2)):
    print line2
    print line1
```

Important

Build asynchronous APIs

Interfaces are great

No way out

Asynchronous builds on asynchronous

succeed

Wraps a normal Python value in a Deferred

Added callbacks fire immediately

Haskell

Type Analysis!

THIS :: String → Deferred

addCallback :: Deferred → (String → Deferred) → Deferred

Hmm.

$$\text{addCallback} :: D \rightarrow (\text{String} \rightarrow D) \rightarrow D$$

$$\text{addCallback} :: D \rightarrow (a \rightarrow D) \rightarrow D$$

$$\text{addCallback} :: D\ a \rightarrow (a \rightarrow D\ b) \rightarrow D\ b$$

Hmmmm.

$$(>>=) :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$$

$$\text{succeed} :: a \rightarrow D\ a$$

$$\text{return} :: a \rightarrow M\ a$$

Type signatures match

Doesn't make it a monad though

Monad Laws!

$$(\text{return } x) >>= f \equiv f\ x$$

$$m >>= return \equiv m$$

The associative one

```
(m >>= f) >>= g
        ≡
m >>= (\x -> f x >>= g)
```

Stand back!

succeed(x).addCallback(f) ≡ f(x)

d.addCallback(succeed) ≡ d

d.addCallback(f).addCallback(g) ≡
d.addCallback(lambda x:
f(x).addCallback(g))

Q. E. D.

Deferreds *are* monads.

Coincidence?

I think not.

Deferreds abstract a sequence of operations

Monads abstract a sequence of operations

Deferreds are used when evaluation time is not guaranteed

Monads are used when evaluation time is not guaranteed

Both abstract “imperative” using functional language constructs

Differences

Deferreds mutate

addErrback

Type flimsiness

do {} notation

Twisted syntax is awful

Python not very malleable

Generator expressions

Introduced in Python 2.5

Before then...

```
def f():
    yield 1
    yield 2
```

```
>>> i = f()
>>> i.next()
  1
>>> i.next()
  2
```

In Python 2.5 and later...

```
def f():
    y = yield 'banana'
    z = yield y
```

```
>>> i = f()
>>> i.send('orange')
'banana'
>>> i.send('apple')
'orange'
```

Coroutines

(but I don't understand that)

inlineCallbacks

```
@inlineCallbacks
def foo():
    line1 = yield
remote.get_line()
    line2 = yield
remote.get_line()
    returnValue((line2,
line1))
```

Hmmmmmm.

do {} notation?

Specific to Deferreds

Conclusions

Haskell isn't *that* insane

… or Twisted is bonkers

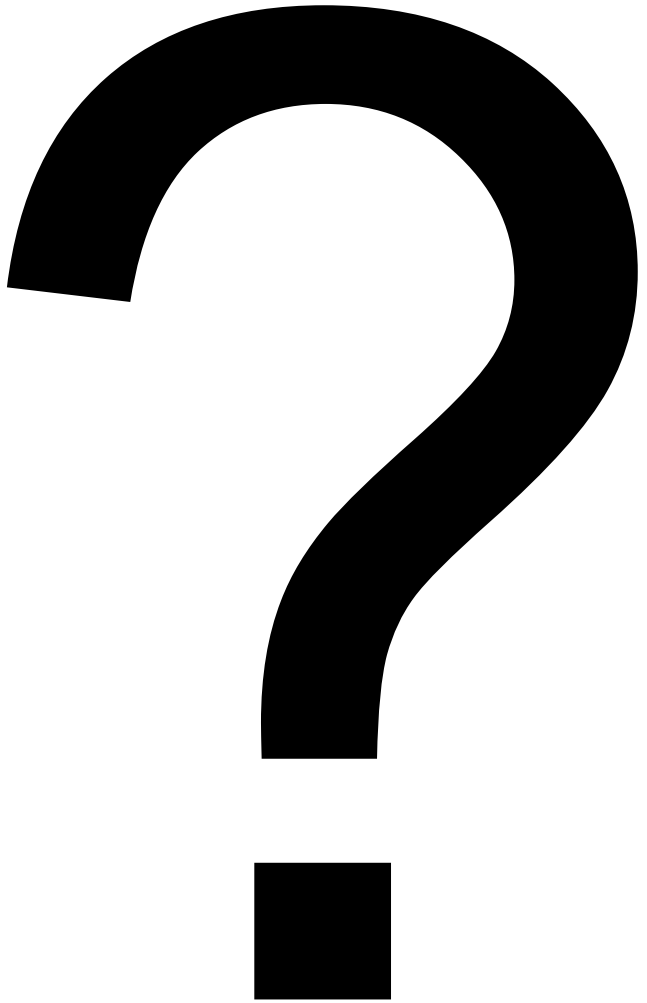
Open Questions

Generic do {} notation in Python?

How do errors fit in?

Immutable Deferreds?

What can we steal from each other?



http://twistedmatrix.com
http://code.mumak.net