

Valued Lessons

<http://www.valuedlessons.com/>

The Overlooked Reason for Google Navigation

A lot has been [written](#) lately about [Google Maps Navigation](#). Google is basically giving away an incredible mapping application with good mapping data for free. Why would they do such a thing? Most of the guesses I've seen basically say "they like to give stuff away for free to push more advertisements". That's close, but everyone seems to have missed **a huge detail, perhaps the most important detail of all**.

Google is an advertisement company, particularly skilled at targeted advertisements. Almost all of their revenue comes from being able to show you ads that you want to see when you want to see them. What does this have to do with maps and navigation? Well, this is going to seem really obvious once you read it, but no one seems to have mentioned it yet, so here it goes: Google will know everywhere you drive, and when.

Do you drive past particular stores to and from work every day? They'll know.

Do you often search for "in-and-out" while driving?. They'll know.

Do you email your friends (using gmail) about how much you love Brand X jeans and are driving past a Brand X outlet mall? They'll know.

Did you recently search (on Google) for Computer Part Y and there's a sale for it at a store on your way to work? They'll know.

Are you driving around and stopping at foreclosed houses? They'll know.

Do you drive the same route to work at the same time of day as 12 other people? Then try the new Google Carpool!

I made that last one up. But you get the idea. The more Google knows about you, the better the advertisements they can serve you. And **knowing everywhere you drive and when and what you are looking for is a lot of very powerful data**. I'm sure the big brains in Mountain View are salivating over the data mining possibilities that they'll probably have very soon. And once you start using it, you'll probably start getting very well targeted advertisements because they'll know so much about you.

Whether it's a good thing or not that they know so much about you is a topic of another day. For now, let us all at least realize one overlooked purpose of Google Maps Navigation: more data about you for better advertisements.

<http://www.valuedlessons.com/2009/11/overlooked-reason-for-google-navigation.html>

Introducing pyrec: The cure to the bane of `__init__`

I finally discovered how cool [github](#) is, and have started putting some code up there. My first entry is Record.py. I'm calling it the cure to the bane of `__init__` because Mutable data structures are a bane to concurrent (multi-threaded) code. Writing `self.foo = foo`, `self.bar = bar`, etc, is a huge waste of time. When you have lots of data structures in memory, tuple-based data uses 1/4 the memory of class-based data. As you can probably tell from my blog, I've experimented with a lot of wacky programming ideas and bending Python in ways it was never intended. **But if there's one experiment that's been a success, it's Record.py.** I use it for almost all of my classes. It's just so easy to use. I'm calling it "pyrec" because it's easier to write, google, etc.

So, go to [the github repo](#) or use it by following the really easy steps: Download Record.py from <http://github.com/pthatcher/pyrec/blob/master/Record.py> put from Record import Record at the top of your code. make a class by saying something like `class Person(Record("name", "age"))` Never write `__init__` again (unless you want mutability). Enjoy!

Update: A commenter (thanks Dan!) pointed out that this is a lot like [namedtuple](#), added in python 2.6. He asked why use this instead of namedtuple. Well, I have to admit that I probably would have never created pyrec if namedtuple existed 3 years ago. I try to avoid NIH syndrome. But it didn't exist, so I wrote pyrec. But I've been using pyrec for three years, so I have some experience on some little things that make a big difference (to me). Here are a few advantages pyrec has over namedtuple: It has a nicer interface. I prefer `new(val1, val2)` to `_make([val1, val2])`, `alter` to `_update`, and `class Person(Record("name", "age"))` to `Person = namedtuple("Person", "name, age")` I added the `setField` methods. That's what I use 90% of the time. Only about 10% of the time do I use `alter`. `setField` is a lot more convenient. With pyrec, you can safely override `__iter__` and `__getitem__`. For example, in Record.py, you'll see the implementation of a `LinkedList`. I tried doing that with namedtuple, but the overridden `__getitem__` clobbers the name lookup and `__iter__` the tuple unpacking. You can use `tuple.__iter__(rec)` to get around the latter, but pyrec's `.values` is a lot nicer. pyrec has `.namedValues` for ordered (field, value) pairs, unlike `_asdict()` which throws out the order. For many things I use pyrec for, this matters. You can improve it! Have looked at the code for namedtuple? Ugly. This is pretty clean, so you can improve it very easily if you need additional functionality which will work with all of your records. If you don't care about those things, use namedtuple. It's still way better than mutable classes. But having used pyrec for three years, these little things matter to me, and so I'm still going to use pyrec. But if you want most of both worlds, I added [NamedTuple](#) to pyrec, which is a subclass of namedtuple which adds most of the pyrec goodness (everything but safe `__getitem__` overloading). Thanks for update, Dan.

<http://www.valuedlessons.com/2009/10/introducing-pyrec-cure-to-bane-of-init.html>

SQL is now turing complete!

[David Fetter](#) at [Oscon 2009](#) just made a stunning claim: **With CTE and Windowing, SQL is Turing Complete**. He even offers a [proof](#) and an [amazing example](#) (although I'm not sure if the example requires turing completeness, it's still amazing).

This could be a big deal. Why? It means that you can do anything in an SQL query. While drawing the mandelbrot set is a nice demo, the first "killer app" will be tree traversal. Lots of relational tables end up looking like trees and are a royal pain to deal with in normal SQL. After that, only our imagination (and performance) is the limit. You can theoretically create any data view that runs in the DB all in one shot, without any of the troubles of stored procedures.

I see a similarity between turing-complete SQL in the DB and JavaScript on the browser. For years, we used JavaScript to do silly stuff, but then a few bright minds created amazing tools like gmail and google maps and our view of JavaScript was forever changed. Now JavaScript is everywhere and there's a race to make the best development framework and engines fastest engines.

Might the same thing happen with turning-complete SQL? **is the race on to be the first programming language to compile down to turing-complete SQL?**

Honestly, if I were a betting man, I'd say it won't come to anything significant. But I'll also guess that the lure of compiling down to SQL will eventually capture someone, somewhere. It's only a matter of time. In fact, if you're a programming language geek, the sirens are probably calling to you right now :). When it does happen, I expect the first language to be a functional one, especially one with a small core, like a variant of lisp. Once we get [clojure in clojure](#), maybe it will be a candidate. Or maybe C# will end up with LINQ-to-turning-complete-SQL (SQLINQ?).

What do you think?

<http://www.valuedlessons.com/2009/08/sql-is-now-turing-complete.html>

The Code for Reactive Programming in Python

I packaged some python code that you can run for each of the steps I've shown in my articles about Reactive Programming, [how you could have invented it](#) and [more ways to do it](#). I apologize that it's not in a more usable form. Go ahead and copy and paste to wherever you like. If you put it online somewhere more convenient, just put a link in the comments. I put the control of which example runs at the very end. Just comment the appropriate line for the example you want to run. And, here it is:

```

import time ## simplified event
stuff class Event: def __init__(self): self.handlers = [] def handle(self, handler): self.handlers.append(handler)
return self #so += will work def fire(self, val = None): for handler in self.handlers: handler(val) def echo(val):
print val return val def simple_click_event_example(): click = Event() click.handle(echo) click.fire("left") #prints
"left" def click_event_manipulation_example(): def left_double_click_from_click(click, threshold): dlclick =
Event() last_lclick_time = [0] #closure hack def click_handler(click_value): if click_value == "left": lclick_time =
time.time() if (lclick_time - last_lclick_time[0]) > value_filter_r("left") >> doublize_r(.01, lambda l1, l2: "double
left") keys >> click_choose_r(d = dlclicks, f = ["fake", "fake"]) >> echo clicks.fire("left") clicks.fire("left")
keys.fire("f") #prints "fake" and then "fake" again keys.fire("d") clicks.fire("right") clicks.fire("right")
time.sleep(.02) clicks.fire("left") clicks.fire("left") #print ("double left") ## basic consumer (receiver) using
generators receive = object() def receiver_example(): def receiver(gen_rcvr): def gen_and_start_rcvr(*args,
**kargs): rcvr = gen_rcvr(*args, **kargs) rcvr.send(None) return rcvr return gen_and_start_rcvr @receiver def
sum_r(title): total = 0 while True: total += yield receive print "%s: %d" % (title, total) @receiver def
count_r(title): count = 0 while True: yield receive count += 1 print "%s: %d" % (title, count) num_key = Event()
sum_nums = sum_r("total nums") num_key.handle(sum_nums.send) num_key.fire(1) #prints "total nums: 1"
num_key.fire(2) #prints "total nums: 3" num_key.fire(3) #prints "total nums: 6" ## make reiterators that can
also output values via an event fire def remitter_example(): class Remitter: def __init__(self,
receiver_from_event_out): self.receiverFromEventOut = receiver_from_event_out def __rrshift__(self,
event_in): event_out = Event() rcvr = self.receiverFromEventOut(event_out) event_in.handle(rcvr.send)
return event_out def remitter(gen_rcvr): def gen_remitter(*args, **kargs): def
receiver_from_event_out(event_out): rcvr = gen_rcvr(event_out, *args, **kargs) rcvr.send(None) return rcvr
return Remitter(receiver_from_event_out) return gen_remitter @remitter def
double_detect_r(double_click_event, threshold): last_click_time = 0 while True: yield receive
current_click_time = time.time() if (current_click_time - last_click_time) > print_r("left") mouse_click >>
double_detect_r(.01) >> print_r("double left") mouse_click.fire() #prints "left" time.sleep(.02)
mouse_click.fire() #prints "left" mouse_click.fire() #prints "left" and "double left" ## make reiterators out of
generators that can send and receive def remitter_example_yield_out(): from collections import defaultdict
class Remitter: def __init__(self, ritr): self.ritr = ritr self.eventOut = Event() def send(self, val_in): ritr = self.ritr
event_out = self.eventOut while True: val_out = ritr.send(val_in) if val_out is receive: break else:
event_out.fire(val_out) def handle(self, handler): self.eventOut.handle(handler) def handlein(self, *events): for
event in events: event.handle(self.send) def __rrshift__(self, event_in): try: self.handlein(*event_in) except:
self.handlein(event_in) return self def remitter(gen_rcvr): def gen_remitter(*args, **kargs): ritr =
gen_rcvr(*args, **kargs) ritr.send(None) return Remitter(ritr) return gen_remitter @remitter def
double_detect_r(threshold): last_click_time = 0 while True: yield receive current_click_time = time.time() if
(current_click_time - last_click_time) > label_r("single"), mouse_clicks >> double_detect_r(.01) >>
label_r("double")] >> label_count_r() >> map_r(fix_click_counts, "single", "double") >>
map_r(print_label_counts) #prints #0 double, 1 single #0 double, 2 single #0 double, 3 single #1 double, 1
single mouse_clicks.fire() time.sleep(.02) mouse_clicks.fire() mouse_clicks.fire() def
remitter_without_yield_in_hack_example(): class Receive: def __init__(self, val = None): self.d = val class
Remitter: def __init__(self, receive, ritr): self.receive = receive self.ritr = ritr self.eventOut = Event() def
send(self, val_in): self.receive.d = val_in ritr = self.ritr event_out = self.eventOut while True: val_out =
ritr.next() if isinstance(val_out, Receive): break else: event_out.fire(val_out) def handle(self, handler):
self.eventOut.handle(handler) def handlein(self, *events): for event in events: event.handle(self.send) def
__rrshift__(self, event_in): try: self.handlein(*event_in) except: self.handlein(event_in) return self def
remitter(gen_rcvr): def gen_remitter(*args, **kargs): receive = Receive() ritr = gen_rcvr(receive, *args,
**kargs) ritr.send(None) return Remitter(receive, ritr) return gen_remitter @remitter def
double_detect_r(receive, threshold): last_click_time = 0 while True: yield receive current_click_time =
time.time() gap = current_click_time - last_click_time if gap > double_detect_r(.05) >> average_r() >>
print_r("double left; average gap is %s seconds") mouse_clicks.fire() time.sleep(.1) mouse_clicks.fire()

```

```
time.sleep(.01) mouse_clicks.fire() #prints #double left; average gap is 0.01... seconds time.sleep(.02)
mouse_clicks.fire() #double left; average gap is 0.015... seconds if __name__ == "__main__":
#simple_click_event_example() #click_event_manipulation_example()
#click_event_manipulation_refactored_example() #click_event_handle_with_result_example()
#click_event_choosing_by_returning_event_example() #click_event_looks_like_streams_example()
#remitter_example() #remitter_example_yield_out() remitter_without_yield_in_hack_example()
```

<http://www.valuedlessons.com/2009/08/code-for-reactive-programming-in-python.html>

More ways to do Reactive Programming in Python

In my last post, I covered a little bit of Rx and how you could have invented it. But you might invent a different way of doing the same thing. And since most languages don't have anything like LINQ, you might be interested in ways to do things in your programming language that don't require monads.

Let's explore some other ways to do Reactive Programming (Rx). What is Rx? Just to remind you what we're trying to accomplish, **Rx builds event handlers**. The LINQ version of Rx works by making an event look like a query (or stream).

It makes sense if you think about it. An event is a stream, of event occurrences. A list or enumerator or iterator is also a stream, of values. So if you squint hard enough, you see that events and enumerators are sort of the same thing. In fact, lists, streams, enumerators, events, channels, pipes, sockets, file handles, actors sending you messages are all pretty much the same thing: they are all producers of values. Consumers and

Receivers Now what do you do with producers of values? You consume them, of course! Usually with something that looks like this (in python):

```
sum = 0
for val in vals:
    sum += val
print sum
```

What we've created here is a consumer of vals. We can write it this way, as ordinary code, because vals is very flexible: it's anything that's iterable/enumerable. But what if instead of forcing the producer to be flexible, we forced the consumer to be flexible? What if we could write the consumer like this:

```
total = 0
while True:
    total += receive
    print total
```

Hmm... it sort of looks like the opposite of an iterator/generator/enumerator block. A mathematician might say something about "duals" at this point, but I'm not mathematician, so let's just go ahead and try and implement it. In fact, we'll use python generators to do just that. We'll call this a "receiver" and we'll spell "receive" as "yield receive":

```
class Event:
    def __init__(self):
        self.handlers = []
    def handle(self, handler):
        self.handlers.append(handler)
    def fire(self, val = None):
        for handler in self.handlers:
            handler(val)

receive = object()
def receiver(gen_rcvr):
    def gen_and_start_rcvr(*args, **kargs):
        rcvr = gen_rcvr(*args, **kargs)
        rcvr.send(None)
        return rcvr
    return gen_and_start_rcvr

@receiver
def sum_r(title):
    total = 0
    while True:
        total += yield receive
        print "%s: %d" % (title, total)

@receiver
def count_r(title):
    count = 0
    while True:
        yield receive
        count += 1
        print "%s: %d" % (title, count)

num_key = Event()
sum_nums = sum_r("total
nums")
num_key.handle(sum_nums.send)
num_key.fire(1) #prints "total
nums: 1"
num_key.fire(2) #prints "total
nums: 3"
num_key.fire(3) #prints "total
nums: 6"
```

It actually works! And because our consumer is very flexible, any producer, like an event, can use it. In fact, it's just a fancy event callback, just like everything else in Rx land.

Remitters But if we take this one step further and make a receiver wrap an event, we can make a receiver that's also a producer. We'll call it a "remitter", which is sort of like a receiver and an emitter.

```
class Remitter:
    def __init__(self, receiver_from_event_out):
        self.receiverFromEventOut = receiver_from_event_out
    def __rrshift__(self, event_in):
        event_out = Event()
        rcvr = self.receiverFromEventOut(event_out)
        event_in.handle(rcvr.send)
        return event_out
    def remitter(gen_rcvr):
        def gen_remitter(*args, **kargs):
            def receiver_from_event_out(event_out):
                rcvr = gen_rcvr(event_out, *args, **kargs)
                rcvr.send(None)
                return rcvr
            return Remitter(receiver_from_event_out)
        return gen_remitter

@remitter
def
```

```
double_detect_r(double_click_event, threshold):
    last_click_time = 0
    while True:
        (yield) current_click_time = time.time()
        if (current_click_time - last_click_time) > print_r("left")
        mouse_click >> double_detect_r(.01) >>
        print_r("double left")
        mouse_click.fire() #prints "left"
        time.sleep(.02)
        mouse_click.fire() #prints "left"
        mouse_click.fire() #prints "left" and "double left"
```

Great. But it is kind of annoying passing in the event like that. What if we had the remitter yield values out and yield values in? Remitters that yield out and in We could do that using little state machines built from python generators. "yield receive" will mean receive and "yield" of anything else will mean "emit".

```
from collections import defaultdict
class Remitter:
    def __init__(self, ritr):
        self.ritr = ritr
        self.eventOut = Event()
    def send(self, val_in):
        ritr = self.ritr
        event_out = self.eventOut
        while True:
            val_out = ritr.send(val_in)
            if val_out is receive:
                break
            else:
                event_out.fire(val_out)
    def handle(self, handler):
        self.eventOut.handle(handler)
    def handlein(self, *events):
        for event in events:
            event.handle(self.send)
    def __rrshift__(self, event_in):
        try:
            self.handlein(*event_in)
        except:
            self.handlein(event_in)
        return self
    def remitter(gen_rcvr):
        def gen_remitter(*args, **kargs):
            ritr = gen_rcvr(*args, **kargs)
            ritr.send(None)
            return Remitter(ritr)
        return gen_remitter

@remitter
def
```

```
double_detect_r(threshold):
    last_click_time = 0
    while True:
        yield receive
        current_click_time = time.time()
        if (current_click_time - last_click_time) > label_r("single"),
        mouse_clicks >> double_detect_r(.01) >> label_r("double")] >>
        label_count_r() >> map_r(fix_click_counts, "single", "double") >>
        map_r(print_label_counts)) #prints#0 double, 1 single#0 double, 2 single#0 double, 3 single#1 double, 1 single
        mouse_clicks.fire()
        time.sleep(.02)
        mouse_clicks.fire()
        mouse_clicks.fire()
        Sweet.
```

That looks pretty nice. But, it relies on the fact that Python allows you to yield values in to a generator. What if we have a programming language that only allows yielding values out (like any enumerator)? Remitters that

yield in by yielding out We'll introduce a simple hack to work around that. We'll yield out a mutable "receive" value that "receives" in the value for us.

```

class Receive:
    def __init__(self, val = None):
        self.d = val

class Remitter:
    def __init__(self, receive, ritr):
        self.receive = receive
        self.ritr = ritr
        self.eventOut = Event()

    def send(self, val_in):
        self.receive.d = val_in
        ritr = self.ritr
        event_out = self.eventOut
        while True:
            val_out = ritr.next()
            if isinstance(val_out, Receive):
                break
            else:
                event_out.fire(val_out)

    def handle(self, handler):
        self.eventOut.handle(handler)

    def handlein(self, *events):
        for event in events:
            event.handle(self.send)

    def __rrshift__(self, event_in):
        try:
            self.handlein(*event_in)
        except:
            self.handlein(event_in)
        return self

    def remitter(gen_rcvr):
        def gen_remitter(*args, **kargs):
            receive = Receive()
            ritr = gen_rcvr(receive, *args, **kargs)
            ritr.send(None)
            return Remitter(receive, ritr)
        return gen_remitter

    @remitterdef
    def double_detect_r(receive, threshold):
        last_click_time = 0
        while True:
            yield receive
            current_click_time = time.time()
            gap = current_click_time - last_click_time
            if gap > double_detect_r(.05) >> average_r() >> print_r("double click; average gap is %s seconds")
            mouse_clicks.fire()
            time.sleep(.1)
            mouse_clicks.fire()
            time.sleep(.01)
            mouse_clicks.fire()
            #prints #double click; average gap is 0.01... second
            time.sleep(.02)
            mouse_clicks.fire()
            #double click; average gap is 0.015... seconds
            It works! And it should work in any language with iterator blocks. You could even use this C# instead of using LINQ Rx, but then you'll have to type "yield return receive" :(. Conclusion Rx is all about making flexible consumers of values, which basically amounts to making event callbacks. LINQ Rx does so with monads, but that's not the only way. Here, we have shown how we can turn a generator or iterator block inside out and make it consume values rather than produce values. Using these is an alternative to LINQ Rx that might be more appropriate for your programming language. There are lots of other things to work out, like unhandling an event, error handling, catching the end of a stream, etc. But this is pretty good, simple foundation to show that the essence of reactive programming is making it easy to make flexible value consumers (basically event handlers). In both the case of Rx, and the code above, we've done so by making a little DSL in the host language. Next time... There are still other ways of making flexible consumers. If we had continuations or used CPS we could just use the current continuation as our consumer. So, scheme and Ruby ought to have easy solutions to this problem. We can do a similar thing with macros in any Lisp language that doesn't have continuations, like Clojure. In fact, I'd like to explore how to do Rx in clojure next time. And at some point, we need to figure out how concurrency fits into all of this. P.S. While I was researching all of this stuff, I was surprised to find that my friend, Wes Dyer, is right at the heart of it. You can see a video of him here. That was a surprise because I've never talked with him about this. In fact, I've only heard from him once in the last year because he's "gone dark" . I'm sure his work on Rx has something to do with that :). I just want to make it clear that all of my thoughts are mine alone, and not based on any communication with him. Don't blame him for my crazy stuff :).
```

<http://www.valuedlessons.com/2009/08/more-ways-to-do-reactive-programming-in.html>

Rx Simplified (Reactive Programming in Python)

Lately, there's been interest in "reactive programming", especially with [Rx](#). What is Rx? I've seen descriptions like "reactive programming with LINQ", "the dual of the enumerator/iterator" and even "a variation of the continuation monad". Oh right...uh...monad? dual? what's going on?

If you like things like "monads", "duals", and category theory, go watch [this video](#), especially until the end. It's pretty funny.

But if those things make your eyes glaze over and you're left wondering what Rx really is, **I want to give you a simple explanation of what Rx is all about. In fact, I'll show how you could have invented it yourself.** We'll do so with simple event-based code written in Python.

Step 1: write simple event handlers Imagine we have a mouse click event that fires either "left" or "right", and we want to make a new event that fires "double left" when there's a double left click. We might write something like this (including a simple Event class).
`import time
class Event:
 def __init__(self):
 self.handlers = []
 def handle(self, handler):
 self.handlers.append(handler)
 def fire(self, val = None):
 for handler in self.handlers:
 handler(val)
 def echo(val):
 print val
 return val
 def left_double_click_from_click(click, threshold):
 dlclick = Event()
 last_lclick_time = [0] #closure hack
 def click_handler(click_value):
 if click_value == "left":
 lclick_time = time.time()
 if (lclick_time - last_lclick_time[0]) > threshold:
 dlclick.fire("left")
 last_lclick_time[0] = lclick_time
 else:
 dlclick.fire("right")
 return dlclick
 def choose_clicks(keys, clicks):
 def key_handler(char):
 if char == "l":
 return filter_event("left", clicks)
 elif char == "r":
 return filter_event("right", clicks)
 elif char == "f":
 return ["fake", "fake"]
 return handle_with_result(keys, key_handler)`

If we change `handle_with_result` to handle this, we might get something like this:
`def handle_with_result(evt, handler_with_result):
 evt_out = Event()
 def handler(value):
 result = handler_with_result(value)
 if result is None:
 pass #ignore
 elif isinstance(result, Event):
 result.handle(evt_out.fire)
 elif isinstance(result, list):
 for value_out in result:
 evt_out.fire(value_out)
 else:
 evt_out.fire(result)
 evt.handle(handler)
 return evt_out
def filter_r(evt, predicate):
 def handler(value):
 if predicate(value):
 return value
 return handle_with_result(evt, handler)`

def value_filter_r(evt, value):
 return filter_r(evt, lambda val : val == value)
def choose_clicks(keys, clicks):
 def key_handler(char): #TODO:
 if char == "l":
 return value_filter_r(clicks, "left")
 elif char == "r":
 return value_filter_r(clicks, "right")
 elif char == "f":
 return ["fake", "fake"]
 return handle_with_result(keys, key_handler)

keys = Event()
clicks = Event()
chosen_clicks = choose_clicks(keys, clicks)
chosen_clicks.handle(echo)
clicks.fire("left")
keys.fire("a")
clicks.fire("right")
keys.fire("l")
clicks.fire("left") # print "left"
clicks.fire("right")
clicks.fire("left") # print "left"
keys.fire("f") #prints "fake" and then "fake" again
Great. Now if we just add a little bit of syntax sugar to this, we can make events look like streams:

Step 4: add some syntax sugar
`class Event:
 def __init__(self):
 self.handlers = []
 def handle(self, handler):
 self.handlers.append(handler)
 return self #so += will work
 def fire(self, val = None):
 for handler in self.handlers:
 handler(val)
 def bind(evt, handler_with_result):
 evt_out = Event()
 def handler(value):
 result = handler_with_result(value)
 if result is not None:
 Event.unit(result).handle(evt_out.fire)
 evt.handle(handler)
 return evt_out
 @classmethod
 def unit(cls, val):
 if isinstance(val, cls):
 return val
 elif isinstance(val, list):
 return MockEvent(*val)
 else:
 return MockEvent(val)
 __rshift__ = bind
class MockEvent:
 def __init__(self, *vals):
 self.vals = vals
 def handle(self, handler):
 for val in self.vals:
 handler(val)
 def double_r(threshold, combine):
 last_fire = EventFireRecord(None, 0)
 def handler(value):
 fire_time = time.time()
 try:
 if (fire_time - last_fire.time) > value_filter_r("left") >> double_r(0.01, lambda l1, l2: "double click"):
 keys.fire("left")
 last_fire = EventFireRecord(fire_time, 1)
 except:
 pass
 return handle_with_result(keys, handler)`

keys >> click_choose_r(d = dlclicks, f = ["fake", "fake"]) >> echo
clicks.fire("left")
clicks.fire("left")
keys.fire("f") #prints "fake" and then "fake" again
keys.fire("d")
clicks.fire("right")
clicks.fire("right")
time.sleep(.02)
clicks.fire("left")
clicks.fire("left") #print ("left", "left")

So what have we made? Wonderful. **We've made events look like streams by making a slick way of creating event handlers.** In fact, if you look closely at what I did in that last step, you'll notice that I renamed "handle_with_result" to "bind" and moved some code into a method called "unit". That's all it takes to turn Event into a monad, which is exactly what Rx does. Congratulations, **we just reinvented monads and Rx, just by refactoring our event handler code** and in the process **we've discovered what Rx really is: a fancy way of writing event handlers**, specifically event handlers that fire events that trigger other event handlers that fire events, and so on in big chain that looks like a query.

So when your eyes glaze over about "duals" and "monads" and "reactive programming", just say to yourself: I'm making a fancy event handler. Because in the end, that's all you're really doing.

In fact, if you want to do so in Python, now you have a basic implementation to start with! Of course, this is

just a toy implementation. It lack error handling, unsubscribing, end-of-stream, and concurrency. But it ain't bad for just 50 lines of code. And it lets you see the essence of Rx fairly easily.

Oh, and what's the big deal with monads, you ask? Nothing much. It's just that if you provide "bind" and "unit" (called "select many" and "select" in LINQ, I think), LINQ gives you some nice syntax sugar that makes your event handler look like a query. It's really pretty slick, especially now that they've added "extension methods". And next time... In future posts, I'll explore different ways of making slick event handlers, but without monads. And hopefully we'll get make this handle concurrency, which is what asynchronous programming is all about. In fact, I expect we'll start to see a serious blurring of lines between Rx, message passing, and data flow programming.

For now, when you start working with Rx, just remember: I'm making a big, fancy event handler. An "Observable" is just like an event and an "Observer" is just like an event handler. P.S. What's with the lame names? They started off with cool names like "Reactive" and "Rx" and then give us Observable, Observer, Subscribe, OnNext, OnDone, and OnError. Yuck. Think what an opportunity they missed! We could have had names like Emitter, Reactor, Chain, Emit, Extinguish, and Explode. Judge for yourself:

```
observable.Subscribe(observer)observer.OnNext(val)observer.OnDone()observer.OnError(e) or  
emitter.chain(reactor)reactor.emit("foo")reactor.extinguish()reactor.explode(e)
```

<http://www.valuedlessons.com/2009/08/simple-rx-reactive-programming-in.html>

Popcount in Python (with benchmarks)

Purpose A slightly uncommon bit-twiddling operation is "popcount", which counts the number high bits. While uncommon, it's crucial for implementing Array Mapped Tries, which is a way to implement immutable maps and vectors. I'm trying to implement immutable maps and vectors, so I needed to implement popcount. I found A TON of ways, but had no idea which was fastest. So, I implemented them all and tested them. The short answer is to do this:

```
#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTablePOPCOUNT_TABLE16 = [0] *
2**16
for index in xrange(len(POPCOUNT_TABLE16)): POPCOUNT_TABLE16[index] = (index & 1) +
POPCOUNT_TABLE16[index >> 1]
def popcount32_table16(v): return (POPCOUNT_TABLE16[v & 0xffff] +
POPCOUNT_TABLE16[(v >> 16) & 0xffff])
```

And here's the long answer: Results I ran popcount on 30,000 random ints between 0 and 231-1. Here are the results from my Linux Desktop with a 3.2Ghz Core 2 Quad:

Name	Time	Max Bits	URL
c_bagwell	0.003032	32	http://lamp.epfl.ch/papers/idealhashtrees.pdf
c_bsdmacro	0.003032	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
c_parallel	0.003132	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel
c_ops64	0.003632	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSet64
table16	0.009832	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable
c_table8	0.011032	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive
table+kern	0.0132	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable
table8	0.016332	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable
bsdmacro	0.019032	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
parallel	0.019932	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
ops64	0.020732	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSet64
bagwell	0.024232	32	http://lamp.epfl.ch/papers/idealhashtrees.pdf
freebsd	0.025732	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
timpeters3	0.027732	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
timpeters2	0.058032	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
timpeters	0.0724	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
kernighan	0.0745	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
elklund	0.0889	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetKernighan
naive	0.1519	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
seistrup	0.2447	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive

And here are the results for my MacBook with a 2.0Ghz Core 2 Duo:

Name	Time	Max Bits	URL
table16	0.027932	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable
table+kern	0.0372	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable
table8	0.041732	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable
bsdmacro	0.048132	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
bagwell	0.059632	32	http://lamp.epfl.ch/papers/idealhashtrees.pdf
timpeters3	0.074432	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
parallel	0.080732	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
ops64	0.129032	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSet64
timpeters2	0.143932	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
kernighan	0.1527	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
table8	0.1668	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetKernighan
timpeters	0.1668	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
freebsd	0.1772	32	http://mail.python.org/pipermail/python-list/1999-July/007696.html
elklund	0.1913	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
naive	0.2889	32	http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.html
seistrup	0.6106	32	http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive

Observations Implementations written in C (using [Pyrex](#)) are 5-6 times faster than Python. My 3.2 Ghz Linux Desktop is 200% faster than my 2.0 Ghz MacBook even though it only has a 60% clockspeed advantage. I'm not sure what to make of that. Python doesn't use multiple cores well, so I'm sure it's not that. If you want to run popcount on 32-bit values in pure Python, table16 is the fastest by far, and only 3 times slower than implementations in C. If you need to run popcount on arbitrarily large integers, kernighan is the best, but doing a hybrid of table16 and kernighan is probably better.

Conclusion If you don't mind using about 64KB of memory, here's is the fastest popcount in pure python:

```

#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTablePOPCOUNT_TABLE16 = [0] *
2**16for index in xrange(len(POPCOUNT_TABLE16)): POPCOUNT_TABLE16[index] = (index & 1) +
POPCOUNT_TABLE16[index >> 1] def popcount32_table16(v): return (POPCOUNT_TABLE16[ v & 0xffff] +
POPCOUNT_TABLE16[(v >> 16) & 0xffff]) If you do mind the memory usage, here's a slightly slower version:
#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTablePOPCOUNT_TABLE8 = [0] * 2**8for
index in xrange(len(POPCOUNT_TABLE8)): POPCOUNT_TABLE8[index] = (index & 1) +
POPCOUNT_TABLE8[index >> 1] def popcount32_table8(v): return (POPCOUNT_TABLE8[ v & 0xff] +
POPCOUNT_TABLE8[(v >> 8) & 0xff] + POPCOUNT_TABLE8[(v >> 16) & 0xff] + POPCOUNT_TABLE8[ v
>> 24 ]) And if you need to handle values of more than 32 bits, one of these two are the best:
#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetKernighandef popcount_kernighan(v): c =
0 while v: v &= v - 1 c += 1 return c POPCOUNT32_LIMIT = 2**32-1POPCOUNT_TABLE8 = [0] * 2**8for
index in xrange(len(POPCOUNT_TABLE8)): POPCOUNT_TABLE8[index] = (index & 1) +
POPCOUNT_TABLE8[index >> 1] def popcount_hybrid(v): if v > 16) & 0xffff) else: c = 0 while v: v &= v - 1 c
+= 1 return c If it needs to be faster than that, write in C! Test For Yourself If you want to test these yourself,
here's some code you can run. #http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaivedef
popcount_naive(v): c = 0 while v: c += (v & 1) v >>= 1 return c
#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetKernighandef popcount_kernighan(v): c =
0 while v: v &= v - 1 c += 1 return c
#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTablePOPCOUNT_TABLE8 = [0] * 2**8for
index in xrange(len(POPCOUNT_TABLE8)): POPCOUNT_TABLE8[index] = (index & 1) +
POPCOUNT_TABLE8[index >> 1] def popcount32_table8(v): return (POPCOUNT_TABLE8[ v & 0xff] +
POPCOUNT_TABLE8[(v >> 8) & 0xff] + POPCOUNT_TABLE8[(v >> 16) & 0xff] + POPCOUNT_TABLE8[ v
>> 24 ]) POPCOUNT_TABLE16 = [0] * 2**16for index in xrange(len(POPCOUNT_TABLE16)):
POPCOUNT_TABLE16[index] = (index & 1) + POPCOUNT_TABLE16[index >> 1] def
popcount32_table16(v): return (POPCOUNT_TABLE16[ v & 0xffff] + POPCOUNT_TABLE16[(v >> 16) &
0xffff]) POPCOUNT32_LIMIT = 2**32-1def popcount_table16_kernighan(v): if v > 16) & 0xffff) else: c = 0
while v: v &= v - 1 c += 1 return c #http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSet64def
popcount32_ops64(v): return (((v & 0xffff) * 0x1001001001001 & 0x84210842108421) % 0x1f) + (((v &
0xffff000) >> 12) * 0x1001001001001 & 0x84210842108421) % 0x1f) + (((v >> 24) * 0x1001001001001 &
0x84210842108421) % 0x1f)
#http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel#also
http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.htmldef popcount32_parallel(v): v = v - ((v >> 1) &
0x55555555) v = (v & 0x33333333) + ((v >> 2) & 0x33333333) #v = (v & 0x0F0F0F0F) + ((v >> 4) &
0x0F0F0F0F) #v = v + ((v >> 4) & 0x0F0F0F0F) v = (v + (v >> 4)) & 0x0F0F0F0F v = (v * 0x1010101) >> 24
return v % 256 #I added %256. I'm not sure why it's needed. It's probably because of signed ints in Python
def popcount32_bagwell(v): v = v - ((v >> 1) & 0x55555555) v = (v & 0x33333333) + ((v >> 2) & 0x33333333)
v = (v & 0x0F0F0F0F) + ((v >> 4) & 0x0F0F0F0F) v = v + (v >> 8) v = (v + (v >> 16)) & 0x3F return v
#http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.htmldef popcount_elklund(v): c = 0 while v: v ^= v & -v c
+= 1 return c #http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.htmldef popcount32_freebsd(v): v = (v &
0x55555555) + ((v & 0xaaaaaaaa) >> 1); v = (v & 0x33333333) + ((v & 0xcccccccc) >> 2); v = (v & 0x0f0f0f0f)
+ ((v & 0xf0f0f0f0) >> 4); v = (v & 0x00ff00ff) + ((v & 0xff00ff00) >> 8); v = (v & 0x0000ffff) + ((v & 0xffff0000)
>> 16); return v #http://resnet.uoregon.edu/~gurney_j/jmpc/bitwise.htmldef popcount32_bsdmacro(v): #define
BITCOUNT(x) (((BX_(x)+(BX_(x)>>4)) & 0x0F0F0F0F) % 255) #define BX_(x) ((x) - (((x)>>1)& -
(((x)>>2)&0x33333333) - (((x)>>3)&0x11111111)) # #def bx(v): (v - ((v >> 1) & 0x77777777) - ((v >> 2) &
0x33333333) - ((v >> 3) & 0x11111111)) #def bc(v): ((bx(v) + (bx(v) >> 4)) & 0x0F0F0F0F) % 255 v = (v - ((v
>> 1) & 0x77777777) - ((v >> 2) & 0x33333333) - ((v >> 3) & 0x11111111)) v = ((v + (v >> 4)) &
0x0F0F0F0F) return v % 255 #http://mail.python.org/pipermail/python-list/1999-July/007696.htmlimport
marshal, array, struct, stringPOPCOUNT8_TRANS_TABLE = "".join(map(chr, POPCOUNT_TABLE8))
#changed by me to match new dumps() and use sum()def popcount_timpeters(v): counts = array.array("B",
string.translate(marshal.dumps(v), POPCOUNT8_TRANS_TABLE)) # overwrite type code counts[0] = 0
return sum(counts) #changed by me to add loop unrolling and not setting digitcounts[0]def
popcount32_timpeters2(v): counts = array.array("B", string.translate(marshal.dumps(v),

```

```

POPCOUNT8_TRANS_TABLE)) return counts[1] + counts[2] + counts[3] + counts[4] #improved by me: no
need to translate type chardef popcount32_timpeters3(v): dumped = marshal.dumps(v) return
POPCOUNT_TABLE8[ord(dumped[1])] + POPCOUNT_TABLE8[ord(dumped[2])] +
POPCOUNT_TABLE8[ord(dumped[3])] + POPCOUNT_TABLE8[ord(dumped[4])]
#http://mail.python.org/pipermail/python-list/1999-July/007696.html _run2mask = {1: 0x5555555555555555L,
2: 0x3333333333333333L, 4: 0x0F0F0F0F0F0F0F0FL, 8: 0x00FF00FF00FF00FFL} def buildmask2(run, n):
run2 = run + run k = (n + run2 - 1) / run2 n = k * run2 try: answer = _run2mask[run] k2 = 64 / run2 except
KeyError: answer = (1L k2: k2 = k2 + k2 if k >= k2: answer = answer | (answer > (run2 * (k2 - k)) return
answer, n def nbits(v): return 32 - (v >> 31) & 1 def popcount_seistrup(v): lomask, n = buildmask2(1, nbits(v)) v = v -
((v >> 1) & lomask) target = 2 while n > target: lomask, n = buildmask2(target, n) v = (v & lomask) + ((v >>
target) & lomask) target = target + target for i in range(1, target/2): if n > 1 n = (n + target - 1) / target * target v
= (v & ((1L n) return int(v) if __name__ == "__main__": import time, random def time_func(func): before =
time.time() result = func() after = time.time() span = after - before return result, span popcounts =
[popcount_naive, popcount32_table8, popcount32_table16, popcount_table16_kernighan,
popcount_kernighan, popcount32_ops64, popcount32_parallel, popcount32_bagwell, popcount_elklund,
popcount32_freebsd, popcount32_bsdmacro, popcount_seistrup, popcount_timpeters,
popcount32_timpeters2, popcount32_timpeters3, ] test_count = 30000 max_int = 2**31 - 2 ints = range(0,
257) + [random.randint(0, max_int) for _ in xrange(test_count)] + range(max_int - 100, max_int+1)
expected_counts = map(popcount_naive, ints) for popcount in popcounts: counts, timespan =
time_func(lambda : map(popcount, ints)) print "%5.5f %s" % ((timespan if (counts == expected_counts) else
-1), popcount.__name__) #-1 means failure

```

<http://www.valuedlessons.com/2009/01/popcount-in-python-with-benchmarks.html>

Easy Python Decorators with the decorator decorator

Decorators in python are very powerful, but are often a pain to get right, especially when you want to pass arguments to the decorator. Typically, it involves defining a function in a function in a function, etc, up to 4 layers deep. Can we make it easier? What if we even made a decorator to make decorators?

```
Perhaps we could use it catch and log errors: @decoratordef log_error(func, *args, **kargs): try: return func(*args, **kargs) except Exception, error: print "error occurred: %r" % error @log_errordef blowup(): raise Exception("blew up") blowup() # this gets caught and prints the error
```

```
Or maybe we'd like to synchronize a function to make it thread-safe: @decoratordef synchronized(lock, func, *args, **kargs): lock.acquire() try: return func(self, *args, **kargs) finally: lock.release() missile_lock = thread.RLock() @synchronized(missile_lock)def launch_missiles(): pass
```

```
Or we could even use arguments to do some object __init__ trickery (something I've had to do when working with wxpython): @decoratordef inherits_format(method, *args, **kargs): self, parent = args[:2] self.format = parent.format return method(*args, **kargs) class Child: @inherits_format def __init__(self, parent): pass class Parent: def __init__(self, format): self.format = format format = object() child = Child(Parent(format)) assert child.format is format
```

If you've never worked python decorators, these are few examples of how powerful they are. But you'll probably never want to write your own because it can be a pain. If that's the case, then **the decorator decorator is for you!** Here's the code for it. It's a little tricky, but all you have to do is import it, slap @decorator before your decorator, make sure you call func(*args, **kargs), and you're all set. **This is as easy as decorators can get.**

Update: [Dave](#) left a comment and notified me that there is another, much more complete, implementation of the same idea at <http://www.phyast.pitt.edu/~micheles/python/documentation.html>. So, if you want a very complete module, go there. If you want something small and simple to cut and paste into your own code, use the following.

```
def decorator(decorator_func): decorator_expected_arg_count = decorator_func.func_code.co_argcount - 1 if decorator_expected_arg_count: def decorator_maker(*decorator_args): assert len(decorator_args) == decorator_expected_arg_count, "%s expects %d args" % (decorator_func.__name__, decorator_expected_arg_count) def _decorator(func): assert callable(func), "Decorator not given a function. Did you forget to give %r arguments?" % (decorator_func.__name__) def decorated_func(*args, **kargs): full_args = decorator_args + (func,) + args return decorator_func(*full_args, **kargs) decorated_func.__name__ = func.__name__ return decorated_func return _decorator return decorator_maker else: def _decorator(func): def decorated_func(*args, **kargs): return decorator_func(func, *args, **kargs) decorated_func.__name__ = func.__name__ return decorated_func return _decorator
```

<http://www.valuedlessons.com/2008/11/easy-python-decorators-with-decorator.html>

The manycore era is already upon us (and python isn't keeping up)

For many months, perhaps years, we as programmers have realized our trouble ahead coming with the "["manycore era"](#)". The attitude seems to be "someday we're really going to have to figure this concurrency thing out". But this month I have been hit with a terrible realization: **the manycore era is already upon us**.

Technically, it's the "multicore", not "manycore", but I think that's a small distinction.

Why the sudden epiphany? A few weeks ago, I built myself a new computer and put in an Intel quad-core CPU. My computer is much faster now, and I'm very happy with it, but after a few weeks I have realized something: **I never use more than 1/4 of my CPU**. I have a little graph on my screen at all times showing me the CPU usage. It goes up to 25% all of the time, but I have never seen it go higher. Ever.

On one hand this is a good thing. The new computer is so fast that everything I do is instant, and only uses a tiny bit of power. If it isn't instant, it's usually disk or network bound, not CPU bound.

On the other hand, **even my own software isn't using more than 25%, and it is CPU bound at times**. I'd like to fix it, but it's written in python, and the short version of a long, boring story is that python has a thing called The GIL which makes it so python as currently implemented cannot use more than 25% of the CPU except under very rare circumstances.

It seems that programming languages takes a long time to be adopted, but I think **concurrency is a big enough rule changer to shake up which programming languages are dominant**. In my specific case, if python doesn't fix its concurrency problems soon, I'm going to have to stop considering it because I'll never be able to get it to use more than 1 little piece of the CPU. Right now, that's 25%, but in a few years, it will be only 3%. Either python is going to have to change, or I'm going to have to change programming languages (or use something like Jython or IronPython, I suppose).

A few weeks ago, sitting behind my single core computer, I was in the "someday, we'll have to tackle concurrency" camp. Now, sitting in front of my quad core machine, never using more than 25% of its power, everything has changed. **Concurrency is no longer a question of if or when, it's here right now**. If you don't believe me, get yourself a quad-core machine and watch the CPU usage graph. I think you'll be surprised.

<http://www.valuedlessons.com/2008/10/manycore-era-is-already-upon-us-and.html>

How to DTrace Python in OSX

[DTrace](#) is an incredible tool. It basically lets you do profiling of a live application with no performance penalty. I'm writing a Python that needed some profiling, and I found the "normal" techniques like the `profile/cProfile` module very lacking. Luckily, Mac OSX comes with DTrace and it even works with Python. The only snag is that it's hard to find how to use the darn thing. I finally figured it out, so I figured it pass on the knowledge.

So, here's how you use `dtrace` on your python application in Mac OSX: Get [DTrace Toolkit](#). Edit `Python/py_cputime.d` by replacing "function-entry" with "entry" and "function-return" with "exit". Call "`sudo dtrace -s Python/py_cputime.d`" Let it sit there a while and hit `ctrl-c`. Enjoy the results

I can only assume you have to edit the file because of some difference between Solaris and OSX. You can try files other than `py_cputime.d`, but you might have to edit them too. Not all of them work, but most do.

The last thing to know is that you have to use the python that comes with OSX. A custom-built python doesn't seem to work.

Hope that helps!

<http://www.valuedlessons.com/2008/10/how-to-dtrace-python-in-osx.html>

Python Memory Usage: What values are taking up so much memory?

.nobr br { display: none }

Python seems to use a lot of memory. So what exactly is the overhead of each type of value? Short answer:
int 24 float 24 tuple 63 list 101 dict 298 old-style class 345 new-style class 336 subclassed tuple 79 Record
79 Record with old class mixin 79 Record with new class mixin 79 Measured in bytes using Python 2.5 in
64-bit Ubuntu Linux

I measured these by running a simple program that loaded up 1,000,000 values in a list and then did
time.sleep(1000). I ran that for different value types and then ran "top" to see how much memory was being
used. I took that value, subtracted the memory usage for a list of all the same value (14 bytes each),
subtracted the value of a child value (usually an int, 24 bytes each), and then divided by 1,000,000. I'll include
the code I ran at the end if you want to cut and paste. So what lessons do we learn from this?

Python objects are very expensive at over 300 bytes each. Tuples have 1/5 as much overhead. Records are
almost as good as tuples, even when a mixin is added.

So, **if you want to have lots of values in memory without using lots of memory, use [Record](#).**

If you want to run the test for yourself, here's the code. Just comment out the "make_val" that you want to
test.

```
import time
from Record import Record
class TupleClass(tuple): pass
class RecordClass(Record("val")): pass
class OldClass:
    def __init__(self, val):
        self.val = val
    def method(self):
        pass
class NewClass(object):
    def __init__(self, val):
        self.val = val
    def method(self):
        pass
class RecordWithOldClass(Record("val"), OldClass):
    pass
class RecordWithNewClass(Record("val"), NewClass):
    pass
make_val = lambda i : 1 #nothing (base overhead)
#make_val = lambda i : i #make_val = float
#make_val = lambda i : (i,) #make_val = lambda i : [i]
#make_val = lambda i : {i:i} #make_val = TupleClass
#make_val = RecordClass
#make_val = OldClass
#make_val = NewClass
#make_val = RecordWithOldClass
#make_val = RecordWithNewClass
count = 1000000
lst = [make_val(i) for i in xrange(count)]
time.sleep(100000)
```

<http://www.valuedlessons.com/2008/10/blog-post.html>

Message Passing Concurrency (Actor Model) in Python

As I wrote previously, I'm a fan of message-passing concurrency. But to use it in python, I had to write my own infrastructure. It's far from perfect, but it's working for me. I've simplified it a bit and prepared some code to share with you.

The basic idea of my framework is to be as simple and easy to use as possible. There's no Erlang-style pattern matching or process linking. There isn't even any built-in error handling. But, there's a lot of support for the typical "react loop" style of message handling.

The architecture is very simple. There's one thread per actor. An actor is just an object you can send a message to. The actor receives the messages in the order they were sent. You always know the sender of the message; it's built right in. There's also "Bridge" that allows you to interact with actors from non-actor code.

The only tricky thing you'll see is a "handles" decorator. This is used to help you identify how you want an actor to handle a given message type. It helps you avoid re-creating a message loop and dispatcher yourself for every actor.

I've included a little example of how to use the framework. It obviously quite simple, but keep in mind that I used little more for the foundation of a rather complex file synchronization application. A little bit goes a long way.

```
So, here it is. Cut, paste, and run.
from threading import Thread, Event
from Queue import Queue, Empty
class IActor:
    pass
# send(message, sender)
class HandlerRegistry(dict):
    # should be used as a decorator
    def __call__(self, typ):
        def register(func):
            self[typ] = func
            return func
        return register
    OtherMessage = "OtherMessage"
class Stop(Exception):
    def __repr__(self):
        return "Stop()"
class Stopped:
    def __repr__(self):
        return "Stopped()"
class ActorNotStartedError(Exception):
    def __init__(self):
        Exception.__init__(self, "actor not started")
class Actor(IActor):
    @classmethod
    def spawn(cls, *args, **kwargs):
        self = cls(*args, **kwargs)
        self.mailbox = Mailbox()
        start_thread(target = self.act, as_daemon = True)
        return self
    def send(self, message, sender):
        if self.mailbox is None:
            raise ActorNotStartedError()
        else:
            self.mailbox.send(message, sender)
    def receive(self):
        if self.mailbox is None:
            raise ActorNotStartedError()
        else:
            return self.mailbox.receive()
    # override if necessary
    def act(self):
        self.handleMessages()
    handles = HandlerRegistry()
    @classmethod
    def makeHandles(*classes):
        return HandlerRegistry((typ, handler) for cls in classes
                                for (typ, handler) in cls.handles.iteritems())
    def handleMessages(self):
        try:
            while True:
                message, sender = self.receive()
                self.handleMessageWithRegistry(message, sender)
        except Stop:
            pass
    def handleMessageWithRegistry(self, message, sender):
        registry = self.__class__.handles
        handler = registry.get(message.__class__)
        or registry.get(OtherMessage)
        if handler is not None:
            handler(self, message, sender)
    @handles(OtherMessage)
    def onOther(self, message, sender):
        pass
    @handles(Stop)
    def onStop(self, message, sender):
        sender.send(Stopped(), self)
        raise message
    def start_thread(target, as_daemon, name = None):
        thread = Thread(target = target)
        if name:
            thread.setName(name)
        thread.setDaemon(as_daemon)
        thread.start()
        return thread
class Mailbox:
    def __init__(self):
        self.mailbox = Queue()
    def send(self, message, sender):
        self.mailbox.put((message, sender), block = False)
    def receive(self, timeout = None):
        return self.mailbox.get(block = True, timeout = timeout)
class Bridge(IActor):
    def __init__(self):
        self.mailbox = Mailbox()
    def send(self, message, sender):
        self.mailbox.send(message, sender)
    def call(self, target, request, timeout, default = None):
        self.sendRequest(target, request)
        return self.receiveResponse(timeout, default)
    # targeted_requests can be an iterator
    def multiCall(self, targeted_requests, timeout, default = None):
        count = 0
        for target, request in targeted_requests:
            self.sendRequest(target, request)
            count += 1
        for _ in xrange(count):
            yield self.receiveResponse(timeout, default)
    def stop(self, actors, timeout):
        stop = Stop()
        return list(self.multiCall(((actor, stop) for actor in actors),
                                timeout, default = None))
    def sendRequest(self, target, request):
        target.send(request, self)
    def receiveResponse(self, timeout, default):
        try:
            message, sender = self.mailbox.receive(timeout = timeout)
            return message
        except Empty:
            return default
if __name__ == "__main__":
    import time
    class GetInventory:
        pass
    class Task:
        def __init__(self, input, destination):
            self.input = input
            self.destination = destination
    class Worker(Actor):
        handles = Actor.makeHandles()
        def __init__(self, skill):
            self.skill = skill
        @handles(Task)
        def onTask(self, task, sender):
            output = self.skill(task.input)
            task.destination.send(output, self)
    class Warehouse(Actor):
        handles = Actor.makeHandles()
        def __init__(self):
            self.inventory = []
        @handles(GetInventory)
        def onGetInventory(self, message, sender):
            # copy the inventory to avoid anyone mutating it
            sender.send(list(self.inventory), self)
        @handles(OtherMessage)
        def onTaskResult(self, result, sender):
            self.inventory.append(result)
    worker = Worker.spawn(lambda x : x * 2)
    positives = Warehouse.spawn()
    negatives = Warehouse.spawn()
    bridge = Bridge()
    for val in [1, 2, 3, -2, -4,
```

```
-6]: warehouse = positives if val >= 0 else negatives worker.send(Task(val, warehouse), sender = None) print
bridge.call(positives, GetInventory(), 1.0) #should be [ 2, 4, 6] print bridge.call(negatives, GetInventory(), 1.0)
#should be [-4, -8, -12] print bridge.stop([worker, positives, negatives], 1.0) #should be [Stopped(), Stopped(),
Stopped()] class Start: def __init__(self, target): self.target = target class Ping: def __repr__(self): return
"Ping()" class Pong: def __repr__(self): return "Pong()" class Pinger(Actor): handles = Actor.makeHandles()
@handles(Start) def onStart(self, start, sender): start.target.send(Ping(), self) @handles(Pong) def
onPong(self, pong, sender): print "-", sender.send(Ping(), self) class Ponger(Actor): handles =
Actor.makeHandles() @handles(Ping) def onPing(self, ping, sender): print "+", sender.send(Pong(), self) #
should print lots of +-+-- ping = Pinger.spawn() ponger = Ponger.spawn() ping.send(Start(ponger),
sender = None) time.sleep(0.1) bridge.stop([pinger, ponger], 1.0)
```

<http://www.valuedlessons.com/2008/06/message-passing-concurrency-actor.html>

My Experience with Message Passing Concurrency

I'm working on a peer-to-peer file synchronization program. It's really concurrent and distributed, and it's forced me to learn a thing or two about the concurrency models that we use as programmers. Over the past few years, I've tried both the shared-state model common to C, C++, Java, C#, Python, etc, and the message-passing model unique to Erlang and Scala. In my experience, **the message-passing model (aka Actor Model) is far superior to the shared-state model for writing distributed applications.** After having used both extensively, I'd go so far as to say that **for distributed programming, shared-state concurrency is upside down and backwards.**

Here's my informal proof that message-passing concurrency is necessary in a distributed system:

Since the system is distributed, real shared state is impossible. The only way for the distributed components of an application to communicate is by sending a message from one to another. Everything else is an abstraction on top of message passing. HTTP is an abstraction on top of message-passing. AJAX is an abstraction on top of HTTP on top of message-passing. XML-RPC is an abstraction on top of HTTP on top of message-passing. SOAP is an abstraction on top of an abstraction on top of an abstraction on top of HTTP on top of message-passing. Any RPC mechanism is an abstraction on top of message-passing. SQL queries to a remote database are an abstraction on top of message-passing. CORBA is a nasty, tangled mess on top of a foundation of message-passing.

See a pattern?

Not only are all of these abstractions, but they are leaky abstractions. Just about all RPC (Remote Procedure Call) frameworks try to pretend that remote objects are local. But the facade is impossible to keep from leaking. Calls to a remote object might fail or take arbitrarily long. If you want to make the same call to two different remote nodes, those calls must be made synchronously and sequentially; in order to call them in parallel on the remote nodes, they must be called in parallel on the local node. If a call to a remote node triggers a call back to the local node, which may trigger a call to a third node, you end up with a huge spaghetti mess of calls and threads.

I've been down that road. It wasn't pretty. There is a better way.

I think AJAX has opened our eyes a bit. It's a lot closer to message-passing than it is to RPC. In a REST architecture, you "send" messages by POSTing to a URL and you "receive" messages by GETing a URL. All messages are clearly local copies that were serialized and deserialized from the remote data. There isn't any leaky abstraction of data or classes pretending to be in two places at once. Data, timeouts, and failures are in your face and you have to deal with them. So, AJAX is a lot less leaky.

But AJAX is far from perfect. I can't help but think of the Greenspun's Tenth Rule of Programming with a twist on distributed programming: "Any sufficiently complicated distributed platform contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of a message-passing system (Erlang)". Once you get message-passing in your head, you can't help but think of AJAX in that way.

I've been down a better road. It was much more pleasant.

About six months ago, I rewrote major portions of our application with message-passing concurrency. I took a long time. It was tricky. It hurt my head. But it worked. It's a success. It's capable of things that the shared-state system could never do.

Having done it, I can emphatically say that if I could start all over, I would go with message-passing. Most of the work was building the infrastructure and wrapping my head around a new way of thinking. But now that I've done both of those, I've paid the costs and I'm reaping the rewards. We're moving the application in directions that would have been nearly impossible with the old concurrency model.

In my experience, message-passing concurrency is the best way to write distributed applications.

But it isn't an easy road to go down. Support for it just isn't there in most programming languages and environments. So far, Erlang has been the pioneer. I would humbly agree that the way I've implemented message-passing in Python, compared to Erlang, looks like an ad-hoc, bug-ridden half-implementation. But for various reasons, I can't use Erlang. I'm hoping for someone to create Erlang++ or E# or Eython or something that combines the concurrency model of Erlang with a modern programming language. Until then, I'll just keep on cobbling together what I can onto the programming language I happen to be using.

More on that in another article.

<http://www.valuedlessons.com/2008/06/my-experience-with-message-passing.html>

Wii Fit From A Programmer's Perspective (after one day)

I bought Wii Fit. It came in the mail yesterday and I had my first try at it last night and my first workout this morning. Although I haven't had a lot of time with it so far, I think I've had enough that I can share some observations.

I have read a lot about Wii Fit from a video game or fitness perspective, but I've never read about it from the perspective of a software developer. I think as software developers, we should have extra interest in it for three reasons:

Most software developers are physically weak. The Wii Fit Balance Board is a new and unique user interface device. There might be some money to be made! Wii Fit as an antidote to programmer physical weakness

I talked myself into buying Wii Fit because of #1. **As a programmer, I sit all day with little or no physical activity.** Most of my life, in fact, has involved sitting with little or no physical activity. If you're reading this, I'm guessing that you probably don't get much physical activity either. Well guess what: that makes us weak!

I know not all programmers are weak. My good friend Wes is a programmer, and he's pretty buff. But I'm weak, and I'm sure I'm not the only one. I didn't understand how weak until I played Wii Fit the first time. Just about every activity (and there are many) involved a seldom-used muscle. After a short while, I realized that **I have a lot of unused muscles.**

Will Wii Fit make me strong? I don't know. I suppose that depends on how much I use it. But if it pushes me to use lots of muscles that I never did, then it can only make me stronger, right? In other words, if I keep using it, **my strength will increase monotonically.** Eventually it will plateau, and its limit is probably lower than going to a gym. But I'm not going to a gym, and this is sure better than nothing.

Plus, it's surprisingly enjoyable! I enjoyed my first workout this morning so much that an hour sped by before I looked at the time. Part of that is novelty, I'm sure, but there's something more to it, and I'm optimistic that I'll keep it up. I'm already looking forward to my next round tomorrow.

Wow. Did you hear what I just said? I said **I'm looking forward to getting up early to have time to exercise.** Either I got hit in the head with a coconut or Wii Fit is really on to something.

Wii Fit Balance Board as a user interface device.

Although I bought it for the exercise, I'm interested in the potential of the Balance Board that comes with Wii Fit. It's one of the most unique user interface devices I've ever used.

Nintendo seems to be on a role with user interface devices. Its success at turning a gyroscope into sports simulation led to the huge success of the Wii. The availability of Wiimotes led to an explosion of [interesting uses](#) and [interest in hacking it](#). It's certainly an interface device success story.

The big question is: how useful with the Balance Board be? In Wii Fit, there seem to be two kinds of uses:

A measuring device. A control device. As a measure device, I think it's very useful. By "measure", I mean that the board measures how well you do a task. For example, I was doing "lunges", and it noticed that I was bending my leg too far, and told me so. That's pretty impressive. When I doing push-ups, it had a rhythm to follow and it really helped to know that someone was "watching"; I couldn't wimp out. Even just standing on it let me know that my whole life I've been standing too much on my heels. How's that for some realization!

Just like Wii Sports was the perfect application for the Wiimote, I think Wii Fit is the perfect application for the Balance Board. I can't think of an activity better suited to its strong point: measuring.

But what else can it do? When we think of a user interface device, we usually think of controlling something, whether it be a web browser or a game character. As a developer, this is the possibility I was most excited about. After having used it, how well does the Balance Board work as a controlling device? I'm not so sure.

Wii Fit comes with a couple of "balance games" showing various ways of using the board to control something. There's leaning left and right to ski, crouching and standing to ski jump, and leaning in all directions to roll a ball into a hole. Some of them are actually quite impressive. But I still doubt the board's effectiveness for precise control.

The problem is **I find it's really hard to control my balance.** I'm not kidding. Just rolling a little ball down into the hole was really hard. My first time on the ski slope was a disaster. I read that a skateboard game is going to support the board. That might end up being so realistic that I can't control it, just like in real life!

I think a big part of it is that **controlling my balancing is something I've never really done before.** Imagine handing someone who has never played video games an Xbox controller and watching them play Halo. It won't be pretty. Similarly, I remember when I was a kid and the mouse was a new input device. My teacher at school couldn't handle it. She'd never had to control anything like it. Like her, I am facing a new

control device, and there's a learning curve.

Long story short: **The Wii Fit Balance Board is hard to use for precise control, but maybe it's just me.** Hopefully, my skill with it will improve, and we as software developers will think of interesting ways of using it.

Profit There's definitely some money to be made writing software for the Balance Board. How so?

Millions of households already have a Balance Board Millions more will soon. They will show it to **everyone they know**. Many will want more beyond Wii Fit. Even though it's only one player at a time, it a hilarious game to play in a group. Some family happened to be over last night, and they couldn't get enough of it. It was a surprisingly fun watching each other try and hula hoop and dodge shoes. The first decent, cheap party game featuring compatibility with the Balance Board is going to sell like hot cakes (which...sell fast, right?).

I'm not saying who is going to make a lot of money. Nintendo might make it all. But there's probably room for little guys too. With WiiWare, games like [LostWinds](#) have shown that a small developer can take a simple idea for a new interface, add a little bit of style, and make a great game. If someone made a game like LostWinds for the Balance Board for \$10, I wouldn't hesitate to buy it.

I'd be excited just to have some Linux drivers and an interesting open source hack to control my computer somehow, even if it's something silly like that MacBook-turned-into-lightsaber trick. Someone might even think of way to really "surf the web". I don't think there's much many in that, though :).

So far, I'm glad I bought Wii Fit. I hope to keep it up and strengthen my muscles. It's a very fun party game, which should lead to widespread adoption and an opportunity to create interesting uses for the Balance Board. It's hard to control precisely, but I hope it will get easier as we learn to control our balance (something that some of us have never done before).

<http://www.valuedlessons.com/2008/05/wii-fit-from-programmers-perspective.html>

Events in Python

I'm a huge fan of the [Actor Model](#). I think that for most applications, it's the best way to do concurrency. As CPUs get more cores, concurrency becomes more important for programmers, and I think the Actor Model will become more important, too.

But, sometimes you need something more light-weight for handling "events". For example, imagine you have some code listening for file changes in a particular directory. What you'd like to do is make an "event" that is "fired" whenever a file change is detected. When fired, there may be a "handler" or "listener" which is notified of the event. That "handler" is ultimately just some code which is executed when the event occurs.

A while ago, **I wanted an event system like this for Python**. I didn't see anything builtin or any library available, so I decided to write my own. I'd like to share what I created with you.

But first, I want to follow an example through other programming languages to give you an idea of what I was trying to accomplish and how it compares with what's out there. After that, I'll give you my implementation in Python. Our example will be listening for changes on the file system. We want to keep the "watcher" code decoupled from the rest of the code, so we use events.

The best implementation of events that I've used is in C#, so we'll start there. In C# 1.0, our file watcher would look something like this:

```
public delegate void FileChangeHandler(string source_path); class FileWatcher{ public event
FileChangeHandler FileChanged; public void WatchForChanges() { ... if(FileChanged != null) {
FileChanged(source_path); } ... }} class FileChangeLogger{ public void OnFileChanged(string source_path) {
Console.WriteLine(String.Format("{0} changed.", source_path)); }} watcher = FileWatcher();logger =
FileChangeLogger();watcher.FileChanged += new
```

```
FileChangeHandler(logger.OnFileChanged);watcher.WatchForChanges(); It's pretty nice. The best part is at
the end, where you can write "watcher.FileChange += ...". But, you need to type the completely useless "new
FileChangeHandler", and you also need to wrap it all in a separate FileChangeLogger class. Luckily, in C#
2.0, they added Anonymous Delegates, which makes this much nicer:
```

```
watcher.FileChanged += delegate(string source_path){ Console.WriteLine(String.Format("{0} changed.",
source_path));} And in C# 3.0, they've made it even nicer! watcher.FileChanged += (source_path =>
Console.WriteLine(String.Format("{0} changed.", source_path))); C# 3.0 has an event system that's
downright slick, with type-inferencing and everything. I don't think it gets much better than that.
```

Actually, there is one thing. If there's no handler registered with the event, calling "FileChanged()" will blow up because it sees FileChanged == null until a handler is registered. This means that you have to write "if(Event != null){Event(...):}" every single time you fire the event. Every single time. If you forget, your code will seem to work fine until there's no handler, in which case it will blow up and you'll smack your forehead because you forgot that you have to repeat that line of code every single time you fire an event. I really mean every single time. It's by far the worst wart on an otherwise elegant system. I have no idea why the designers of C# thought this was a good idea. When would you ever want to fire an event and have it blow up in your face?

Anyway, let's try a different programming language, perhaps Java. It has the worst implementation of events I've ever seen. Here's our FileWatcher example:

```
...
...
```

Ok, nevermind, I don't have the heart. I can imagine the code full of IListeners, and implementations, and keeping an array of them, and iterating over them, etc, and I just can't do it. I got seriously upset at one useless line in C#. In Java, it's at least 10 times worse. If you really have the stomach for it, go look at <http://java.sun.com/docs/books/tutorial/uiswing/events/index.html>. To me, it appears that **in Java, events are one giant hack around the lack of first-class functions**. If Java had first-class functions, none of that nonsense would be necessary.

Now that we've seen a good implementation of events in C# and avoided a bad one in Java, let's make one for Python. I'd rather it be like the C# event system, so let's see what our example would look like:

```
from Event import Event class FileWatcher: def __init__(self): self.fileChanged = Event() def watchFiles(self):
... self.fileChanged(source_path) ... def log_file_change(source_path): print "%r changed." % (source_path,)
watcher = FileWatcher()watcher.fileChanged += log_file_change I think that looks pretty good. So what does
the implementation of Event look like?
```

```
class Event(IEvent): def __init__(self): self.handlers = set() def handle(self, handler):
```

```
self.handler.add(handler) return self def unhandle(self, handler): try: self.handlers.remove(handler) except: raise ValueError("Handler is not handling this event, so cannot unhandle it.") return self def fire(self, *args, **kargs): for handler in self.handlers: handler(*args, **kargs) def getHandlerCount(self): return len(self.handlers) __iadd__ = handle __isub__ = unhandle __call__ = fire __len__ = getHandlerCount Wow. That was pretty short. Actually, this is one of the reasons I love Python. If the language doesn't have a feature, we can probably add it. We just added one of C#'s best features to Python in 26 lines of code. More importantly, we now have a nice, light-weight, easy-to-use event system for Python.
```

For all of you how like full examples that you can cut and paste, here is one that you can run. Enjoy!

```
class Event: def __init__(self): self.handlers = set() def handle(self, handler): self.handlers.add(handler) return self def unhandle(self, handler): try: self.handlers.remove(handler) except: raise ValueError("Handler is not handling this event, so cannot unhandle it.") return self def fire(self, *args, **kargs): for handler in self.handlers: handler(*args, **kargs) def getHandlerCount(self): return len(self.handlers) __iadd__ = handle __isub__ = unhandle __call__ = fire __len__ = getHandlerCount class MockFileWatcher: def __init__(self): self.fileChanged = Event() def watchFiles(self): source_path = "foo" self.fileChanged(source_path) def log_file_change(source_path): print "%r changed." % (source_path,) def log_file_change2(source_path): print "%r changed!" % (source_path,) watcher = MockFileWatcher()watcher.fileChanged += log_file_change2watcher.fileChanged += log_file_changewatcher.fileChanged -= log_file_change2watcher.watchFiles()
```

<http://www.valuedlessons.com/2008/04/events-in-python.html>

Banana Cream Pie Shake in 30 Seconds

I don't intend this blog to be purely about programming matters. I intend it to be about anything I've learned and would like to share. Today I'd like to share with you an incredibly delicious and easy desert recipe that I've never seen before recently but stumbled upon. You'll love it. But if you are really only interested in programming topics, then I suggest you change your bookmark to [a programming-label-only link](#) or your feed reader to [a programming-label-only feed](#) to enjoy just my posts about programming (at least I hope you enjoy them :). No about that recipe... If you like banana cream pie or a banana shake, this recipe is right up your alley. It's a banana cream pie shake. It tastes exactly like a banana cream pie, except that you can drink it, it's healthy for you, and it only takes 30 seconds. Here's the recipe: 2 frozen bananas (I chop them a bit to blend easier) 1 cup of liquid (I use soy milk) blend until smooth I'm still astounded that something that costs so little (clearly less than \$1) and takes so little time (a few minutes if you are slow) is so delicious and healthy. Finally, a little FAQ: Q: Does it really only takes 30 seconds? A: It only takes 30 seconds to blend, but obviously bananas take a few hours to freeze, and you may spend a minute or two cutting, pouring, and cleaning up. Not counting the freezing, the whole process is just a few minutes. Q: What kind of blender do I need? I have a [Blendtec TotalBlender](#). Have you seen them [blend an iPhone](#)? That's my blender. I'll hopefully review it for you someday; it really is that good. It's some of the best \$400 I've ever spent. Frozen bananas are no challenge for it. If you have a lesser blender and it doesn't work at first, I'd recommend that you let the bananas thaw for a little while. Q: Does it really taste that good? Yes. It's so good I'm about to stop writing this to make some more. I drank the last one so fast it didn't even make it through this whole post. Q: Is it really healthy? If the ingredients are healthy, it's healthy. Bananas are healthy. Just use healthy liquid. You can even toss in extra healthy ingredients like flax seed and make it even healthier without altering the taste much. Adding chocolate might make it more tasty, but it will also make it less healthy. I prefer it without chocolate anyway. Q: I tried it and it's **too** thick and creamy. A: Add ice. It dilutes it while keeping it cold and smooth. Enjoy! Update: I made it again with a different batch of bananas, and it tasted different. It appears that the ripeness of the bananas makes a big difference. More ripe is more tasty. So, wait until the bananas are very ripe before freezing them. Update 2: I sprinkled on a little cinnamon, and it tastes like a mix between a shake, banana bread, and even egg nog. It's delicious!

<http://www.valuedlessons.com/2008/04/banana-cream-pie-shake-in-30-seconds.html>

You Could Have Invented Monadic Parsing

Even though Pysec (and functional programming in python in general) [turned out to be 2-3x slower](#), monadic parsing is still great for certain tasks. After I wrote about Pysec [the first time](#), I received comments regarding [pyparsing](#), which is a great parsing library that looks very similar. Today, I'd like to show you a very simple example where monadic parsing really shines above all other parsing techniques, including pyparsing. This is a real-world need that I had, not just an academic thought-experiment.

The situation is that I'm writing a peer-to-peer file synchronizer and I need to serialize binary data in an efficient way. I've chosen to use what I call "sized binary", which is basically [netstrings](#) without the comma or [bencode's byte string](#). The byte-string "ABC", for example, would be encoded as "3:ABC", or "Hello World!" would be "12:Hello World!". This is very efficient because "ABC" or "Hello World!" can be **any** bytes `\x00-\xFF`, without any sort of encoding or decoding like [uencode](#).

Stop and think for a minute how you would define a grammar to parse "3:ABC". It's so simple, right? Well, try and write a grammar for it. Try using BNF, EBNF, yacc/bison, pyparsing, SimpleParse, or any parsing library you know of, in any programming language. If you manage, please write me an email and let me know how you did it, because I'm not so sure it's even possible. Maybe it is in the [Perl6 grammars](#); they're throwing the kitchen sink into that thing :). The best I've seen is that the creator of pyparsing hacked a grammar object to change how it parses the "ABC" part after a different object parsed the "3" part. This basically uses the grammar object as a global variable. It works in a single-core world, but it won't work in the many-core world we're headed for.

But now you're getting bored with me because it's silly ringing my hands over such a simple format. I bet you could parse this format in just 4 lines of code:

```
def parse_sized_binary(stream): size_bytes, stream = stream.readUntil(":") size = int(size_bytes) bytes, stream = stream.read(size) return bytes, stream
```

You're right, it's really simple, at least until you have to embed this little "sized binary language" inside of a bigger language. Perhaps, like me, you need to embed binary data inside of a bigger data structure, and so you need to define a serialization for things like ints, floats, lists, and hash tables. For that stuff, you can define a grammar for a language like [JSON](#) or [YAML](#) and hand it over to a grammar-based parser library. But now you need a way to tell the library, "OK, when you see my binary data, hand control over to my four lines of code, and then I'll hand control back". You might even say "You know, why don't you just let me hand you any function of the form `stream -> (value, stream)` and let me control the parsing for a while".

Congratulations, you just invented monadic parsing. The function you wrote, `parse_sized_binary`, of the form `stream -> (bytes, stream)`, is a Parser Monad. Monadic parsing is just combining lots of these little parsers together. And so, it's incredibly easy to insert arbitrary parsing code, like `parse_sized_binary`, into any parser.

As a complete example, using Pysec, here's a full parser for a language just like [bencode](#) but with support for floats and unicode. Notice that our `parse_size_binary` is renamed to `bencode_sized` and is given a more functional definition.

```
from Pysec import Stub, until, lift, read, many_until, branch bencode_any = Stub() bencode_sized = until(":")
>> lift(int) >> read bencode_unsized = until("e") bencode_list = many_until(bencode_any,
"e") bencode_any.set(branch({"s" : bencode_sized, "u" : bencode_sized >> lift(lambda bytes :
bytes.decode("utf8")), "i" : bencode_unsized >> lift(int), "f" : bencode_unsized >> lift(float), "l" : bencode_list,
"d" : bencode_list >> lift(dict)}))
```

I think this is sort of like [Greenspun's Tenth Rule](#). Any implementation of netstrings/sized-binary-data probably has an ad-hoc implementation of monadic parsing in it. Even my own non-monadic version of this parser (which I use for performance reasons) has an ad-hoc implementation of monadic parsing. I'm convinced that if you write a parser for a similar format, you'll implement/invent monadic parsing as well, even if you don't know it.

<http://www.valuedlessons.com/2008/04/you-could-have-invented-monadic-parsing.html>

Why are my monads so slow?

If you've read the last few posts, you know that I choose to use monadic parsing for some application code I am writing. At an abstract level, it turned out beautifully. I was able to write the parser very elegantly and compactly despite the serialization format having some unique requirements. But at a practical level, it failed pretty miserably. **It was horribly slow.** With a deadline looming, I had to abandon my monadic parser and go back to using an older serialization format. But not content to give up so easily, I decided to figure out **why are my monads so slow?** I hope that my investigation may be of help to you if you choose to use similar techniques in your code.

First, I tried running a profiler like [cProfile](#). While such profilers normally work quite well, they don't work so well with lots of first-class functions. And since monadic code is full of first-class functions, the profiler didn't give much valuable information.

So, I had to do optimizing the "old fashioned" way. I wrote a benchmark and ran both of my serializers against it. To start, I had two data points: one in traditional imperative code and one in fully monadic code, and this is what I got:

imperative: 0.465 seconds
monadic: 3.893 seconds
Monadic code was 8x slower! But why? To narrow it down, I added some more data points by implementing the serializer in gradually more monadic ways. For example, I wrote a serializer in a functional/immutable style that passes around a "stream" rather than a string, and then another that passed around "state" in a very monadic way, but without using the monad class that I used in Pysec. Then I got this:

imperative: 0.465 seconds
functional with stream: 1.018 seconds
almost monadic with state: 2.450 seconds
monadic: 3.893 seconds
This gave some answers. One answer it gave was that passing around an immutable "stream" object is twice as expensive as merely passing around a string. Also, passing around a state object on top of that is even more expensive. With these clues, I was able to optimize in certain ways and reduce it to this:

imperative: 0.465 seconds
functional with stream: 1.018 seconds
almost monadic with state: 1.098 seconds
monadic: 1.283 seconds

That's a nice 3x improvement. What things were I able to do? By far the biggest cost that I was able to eliminate was object creation. I flattened two layers of abstraction into one, and thus split the number of objects created in half. The second thing I did was I created a "read until" operation that could be used more efficiently in the "grammar" of the parser. This is a form of pushing performance-critical code down the layers of abstraction. Finally, I didn't use decorators, for some often-called functions.

In the end, **it looks like monadic code is about 2-3x slower in Python.** Almost all of that is actually the result of just doing things in a functional/immutable way. In particular, creating data structures appears to be the main bottleneck. In functional languages, these types of operations are very common and optimized heavily, and so are typically very fast. It looks like it's not the same in Python. Just like you have to avoid recursion because there's no tail recursion, it looks like you have to avoid a functional/immutable coding style if you care about performance because object creation is so slow. On the other hand, if you don't mind the performance hit, it makes the code much more elegant, just like recursion usually does.

One thing of interest is that there is essentially no performance penalty for using monads over using a functional/immutable code style. The 20% penalty seen between "almost monadic" and "monadic" is only because I'm wrapping the monad in a Parser class which allows nice operator overloading.

Here's a summary of what you can do to speed up any functional/immutable-style code, including monadic code when writing in Python: Make object creation as fast as possible. Don't do anything fancy in `__init__`. Use as few layers of abstraction as possible, especially when there is an object created in each layer. Push common or expensive operations down the layers of abstraction, especially if it avoids creating objects. Avoid using decorators for heavily used functions. Don't use wrapper classes if you don't have to.

As a final thought, I'd like to mention that while there's currently a substantial performance penalty for using immutable data structures, that style is going to become increasingly important as we enter the many-core era. No matter what style of concurrency you like, immutable data is always easier to work with when concurrency is involved. Concurrency and mutable data are just a bad combo. I think that it's going to be very important for language designers to address this when working on the performance of their languages. I certainly hope future versions Python are much faster with immutable data. If they are, then the performance penalty of using Monads will almost disappear.

Pysec: Monadic Combinatoric Parsing in Python (aka Parsec in Python)

Update: It turns out that there's a similar technique being used by [pyparsing](#). I hadn't seen it before and when I first saw it I thought had reinvented the wheel and wasted my time. But upon further inspection, Pysec does a few things a much better than pyparsing, which happen to be the exact things that I need. There's no coincidence, of course, that Pysec matches my needs well. I'll be covering this in more detail in a future article.

Update 2: I got [@do syntax](#) to work! Again, stay tuned for another article on how.

I've talked about monads in the past, but I never really covered what purpose they serve. I covered the "how" in [Python](#) and [Ruby](#), but I doubt I'll ever full cover the "why" because it's simply too big of a subject. But today, I'd like to share with you one example of how monads are useful. In fact, it's the example that motivated me to do all of the monad stuff in the first place: parsing.

Parsing is a topic that's been around pretty much forever in computer science and most people think it's pretty much "solved". My experience is that we've still got a long way to go. Specifically, I'm writing an application with lots of distributed concurrency, which requires lots of data serialization and deserialization (aka parsing). There are very few good serialization libraries out there, and I've been through three or four versions of various techniques. **Finally, I think I have found a parsing technique that works well: monadic combinatoric parsing.** And it's in Python.

What the heck does that mean? "monadic" means we're using monads. "combinatoric" means we can take monad parsers and combine them to make new monad parsers, which is extremely powerful. I call it Pysec. The design is a copy of [Parsec](#) brought to Python. Notice how I said "design"; I didn't bother looking at any of their code; The design described on their web page was good enough guidance for me. But, I'm sure that their implementation is WAY better than mine. If you want to see real monadic parsing, look at Parsec. If you're interested in monadic parsing for Python, keep reading.

```
Here's an example of Pysec for parsing a subset of JSON: from Pysec import Parser, choice, quoted_chars,
group_chars, option_chars, digits, between, pair, spaces, match, quoted_collection # json_choices is a hack
to get around mutual recursion # a json is value is one of text, number, mapping, and collection# text is any
characters between quotes# a number is like the regular expression -?[0-9]+(\.[0-9]+)?# "parser >>
Parser.lift(func)" means "pass the parsed value into func and return a new Parser"# quoted_collection(start,
space, inner, joiner, end)# means "a list of inner separated by joiner surrounded by start and end"# we have
to put a lot of "spaces" in since JSON allows lot of optional whitespace json_choices = []json =
choice(json_choices)text = quoted_chars("'", "'")number = group_chars([option_chars(["-"]), digits,
option_chars([".", digits])] >> Parser.lift(float)joiner = between(spaces, match(","), spaces)mapping_pair =
pair(text, spaces & match(":") & spaces & json)collection = quoted_collection("[", spaces, json, joiner, "]") >>
Parser.lift(list)mapping = quoted_collection("{", spaces, mapping_pair, joiner, "}") >>
Parser.lift(dict)json_choices.extend([text, number, mapping, collection]) print json.parseString('{\"a\" : -1.0, \"b\" :
2.0, \"z\" : {\"c\" : [1.0, [2.0, [3.0]]]})')
```

Like most monadic or functional code, it's pretty dense, so don't feel bad if you go cross-eyed looking at it the first time. Realize that most of the code is building a Parser monad called "json", which parses the test string at the end. I tried to comment each individual part to explain what's going on.

You may be thinking "why would I want to write code like this?". One response is to look at the code: it's the bulk of JSON parsing in 15 lines that look like a grammar definition! Another response I can give is a challenge: go write a parser to parse that string and then compare your code to this code. Which is shorter? Which is easier to read? Which is more elegant? While in college, I had a team project to write a compiler for a subset of Pascal. We were smart enough to use Python, but dumb enough to use Yacc and Flex. I'm sure the parser portion was pretty fast, but it was incredibly painful to get it right. Once we did, we dared not touch it for fear of breaking it. I really wish I had Parse/Pysec back then (ok, Parsec was around back then, but I hadn't even heard of Haskell or monads).

But **monadic combinatoric parsing isn't just about making your code look like a grammar definition.** It makes it possible to combine parsers in incredibly flexible ways. For example, let's say that on a different project, you wrote a simplified CSV parser for numbers like this one: `def line(cell): return sep_end_by(cell, match(","))` `def csv(cell): return sep_end_by(line(cell), match("\n"))` `print csv(number).parseString("1,2,3\n4,5,6")`

And now you realize you'd really like to put whole JSON values in your simplified CSV. In other words, you

want to combine the CVS and JSON parsers. I think that you'll find that doing so really isn't as easy as it sounds. Imagine trying to combine two Yacc grammars. It hurts just thinking about that. Luckily, monadic combinatoric parsers make this incredibly easy: `print csv(json).parseString("{\"a' : 'A'},[1, 2, 3], 'zzz'\n-1.0,2.0,-3.0")`

While this is a slightly contrived example, you must understand that with this technique **you can combine any two parsers** in a similar fashion. I don't know about you, but that really feels right to me. I haven't seen any parsing techniques as elegant as that.

Everything it's perfect of course, especially since making this work in Python is a bit of a hack. Here are some downsides to this approach: It's hard to debug when you do something wrong. It's annoying to import so many things from Pysec. You have to hack around mutual recursion of values in a strict language like python. There's probably a performance hit.

Most of these can be either fixed or worked around, though, so I think long-term monadic parsing is good bet. I'd like to see what you can do with it or to make it better.

I know how much you all like a working example, so here's some code that you can just cut, paste, and run (in Python 2.5; older versions of Python may require some tweaking). Please ignore the top half. It's mostly stuff I've covered in other articles, like [Immutable Records](#) and [monad base classes](#). The meat starts where it says "Parser Monad".

```
I'd like to talk about it in more detail, but I'll save that for another article. For now, please play around with it and see what you think. ##### Base Libraries included here for convenience ##### def
Record(*props): class cls(RecordBase): pass cls.setProps(props) return cls class RecordBase(tuple):
PROPS = () def __new__(cls, *values): if cls.prepare != RecordBase.prepare: values = cls.prepare(*values)
return cls.fromValues(values) @classmethod def fromValues(cls, values): return tuple.__new__(cls, values)
def __repr__(self): return self.__class__.__name__ + tuple.__repr__(self) ## overridable @classmethod def
prepare(cls, *args): return args ## setting up getters and setters @classmethod def setProps(cls, props): for
index, prop in enumerate(props): cls.setProp(index, prop) cls.PROPS = props @classmethod def setProp(cls,
index, prop): getter_name = prop setter_name = "set" + prop[0].upper() + prop[1:] setattr(cls, getter_name,
cls.makeGetter(index, prop)) setattr(cls, setter_name, cls.makeSetter(index, prop)) @classmethod def
makeGetter(cls, index, prop): return property(fget = lambda self : self[index]) @classmethod def
makeSetter(cls, index, prop): def setter(self, value): values = (value if current_index == index else
current_value for current_index, current_value in enumerate(self)) return self.fromValues(values) return setter
class ByteStream(Record("bytes", "index")): @classmethod def prepare(cls, bytes, index = 0): return (bytes,
index) def get(self, count): start = self.index end = start + count bytes = self.bytes[start : end] return bytes,
(self.setIndex(end) if bytes else self) def make_decorator(func, *dec_args): def decorator(undecorated): def
decorated(*args, **kargs): return func(undecorated, args, kargs, *dec_args) decorated.__name__ =
undecorated.__name__ return decorated decorator.__name__ = func.__name__ return decorator decorator
= make_decorator class Monad: ## Must be overridden def bind(self, func): raise NotImplementedError
@classmethod def unit(cls, val): raise NotImplementedError @classmethod def lift(cls, func): return (lambda
val : cls.unit(func(val))) ## useful defaults that should probably NOT be overridden def __rshift__(self,
bindee): return self.bind(bindee) def __and__(self, monad): return self.shove(monad) ## could be overridden
if useful or if more efficient def shove(self, monad): return self.bind(lambda _ : monad) class
StateChanger(Record("changer", "bindees"), Monad): @classmethod def prepare(cls, changer, bindees = ()):
return (changer, bindees) # binding can be slow since it happens at bind time rather than at run time def
bind(self, bindee): return self.setBindees(self.bindees + (bindee,)) def __call__(self, state): return
self.run(state) def run(self, state0): value, state = self.changer(state0) if callable(self.changer) else
self.changer state = state0 if state is None else state for bindee in self.bindees: value, state =
bindee(value).run(state) return (value, state) @classmethod def unit(cls, value): return cls((value, None))
##### Parser Monad ##### class ParserState(Record("stream", "position")): @classmethod def
prepare(cls, stream, position = 0): return (stream, position) def read(self, count): collection, stream =
self.stream.get(count) return collection, self.fromValues((stream, self.position + count)) class
Parser(StateChanger): def parseString(self, bytes): return self.parseStream(ByteStream(bytes)) def
parseStream(self, stream): state = ParserState(stream) value, state = self.run(state) return value class
ParseFailed(Exception): def __init__(self, message, state): self.message = message self.state = state
Exception.__init__(self, message) @decoratordef parser(func, func_args, func_kargs): def changer(state):
```

```

return func(state, *func_args, **func_kargs) changer.__name__ = func.__name__ return Parser(changer)
##### combinatoric functions ##### @parserdef tokens(state0, count, process): tokens, state1 =
state0.read(count) passed, value = process(tokens) if passed: return (value, state1) else: raise
ParseFailed(value, state0) def read(count): return tokens(count, lambda values : (True, values)) @parserdef
skip(state0, parser): value, state1 = parser(state0) return (None, state1) @parserdef option(state,
default_value, parser): try: return parser(state) except ParseFailed, failure: if failure.state == state: return
(default_value, state) else: raise @parserdef choice(state, parsers): for parser in parsers: try: return
parser(state) except ParseFailed, failure: if failure.state != state: raise failure raise ParseFailed("no choices
were found", state) @parserdef match(state0, expected): actual, state1 = read(len(expected))(state0) if actual
== expected: return actual, state1 else: raise ParseFailed("expected %r" % (expected,), state0) def
between(before, inner, after): return before & inner >> (lambda value : after & Parser.unit(value)) def
quoted(before, inner, after): return between(match(before), inner, match(after)) def quoted_collection(start,
space, inner, joiner, end): return quoted(start, space & sep_end_by(inner, joiner), end) @parserdef
many(state, parser, min_count = 0): values = [] try: while True: value, state = parser(state)
values.append(value) except ParseFailed: if len(values) > (lambda value1 : parser2 >> (lambda value2 :
Parser.unit((value1, value2)))) @parserdef skip_many(state, parser): try: while True: value, state =
parser(state) except ParseFailed: return (None, state) def skip_before(before, parser): return skip(before) &
parser @parserdef skip_after(state0, parser, after): value, state1 = parser(state0) _, state2 = after(state1)
return value, state2 @parserdef option_many(state0, first, repeated, min_count = 0): try: first_value, state1 =
first(state0) except ParseFailed: if min_count > 0: raise else: return [], state0 else: values, state2 =
many(repeated, min_count-1)(state1) values.insert(0, first_value) return values, state2 # parser separated
and ended by sepdef end_by(parser, sep_parser, min_count = 0): return many(skip_after(parser,
sep_parser), min_count) # parser separated by sepdef sep_by(parser, sep_parser, min_count = 0): return
option_many(parser, skip_before(sep_parser, parser), min_count) # parser separated and optionally ended
by sepdef sep_end_by(parser, sep_parser, min_count = 0): return skip_after(sep_by(parser, sep_parser,
min_count), option(None, sep_parser)) ##### char-specific parsing ##### def satisfy(name, passes):
return tokens(1, lambda char : (True, char) if passes(char) else (False, "not " + name)) def one_of(chars):
char_set = frozenset(chars) return satisfy("one of %r" % chars, lambda char : char in char_set) def
none_of(chars): char_set = frozenset(chars) return satisfy("not one of %r" % chars, lambda char : char and
char not in char_set) def maybe_match_parser(parser): return match(parser) if isinstance(parser, str) else
parser def maybe_match_parsers(parsers): return tuple(maybe_match_parser(parser) for parser in parsers)
def many_chars(parser, min_count = 0): return join_chars(many(parser, min_count)) def
option_chars(parsers): return option("", group_chars(parsers)) def group_chars(parsers): return
join_chars(group(maybe_match_parsers(parsers))) #return join_chars(group(parsers)) def join_chars(parser):
return parser >> Parser.lift("".join) def while_one_of(chars, min_count = 0): return many_chars(one_of(chars),
min_count) def until_one_of(chars, min_count = 0): return many_chars(none_of(chars), min_count) def
char_range(begin, end): return "".join(chr(num) for num in xrange(ord(begin), ord(end))) def
quoted_chars(start, end): assert len(end) == 1, "end string must be exactly 1 character" return quoted(start,
many_chars(none_of(end)), end) digit = one_of(char_range("0", "9")) digits = many_chars(digit, min_count =
1) space = one_of("\ \t\r\n") spaces = skip_many(space) ##### simplified JSON
##### #from Pysec import Parser, choice, quoted_chars, group_chars, option_chars,
digits, between, pair, spaces, match, quoted_collection, sep_end_by #HACK: json_choices is used to get
around mutual recursion #a json is value is one of text, number, mapping, and collection, which we define
later json_choices = [] json = choice(json_choices) #text is any characters between quotestext =
quoted_chars("'", "'") #sort of like the regular expression -?[0-9]+(\.[0-9]+)? #in case you're unfamiliar with
monads, "parser >> Parser.lift(func)" means "pass the parsed value into func but give me a new Parser
back" number = group_chars([option_chars(["-"]), digits, option_chars([".", digits])]) >> Parser.lift(float)
#quoted_collection(start, space, inner, joiner, end) means "a list of inner separated by joiner surrounded by
start and end" #also, we have to put a lot of spaces in there since JSON allows lot of optional
whitespace joiner = between(spaces, match(", "), spaces) mapping_pair = pair(text, spaces & match(":") &
spaces & json) collection = quoted_collection("[", spaces, json, joiner, "]") >> Parser.lift(list) mapping =
quoted_collection("{", spaces, mapping_pair, joiner, "}") >> Parser.lift(dict) #HACK: finish the work around
mutual recursion json_choices.extend([text, number, mapping, collection]) ##### simplified CSV

```

```
##### def line(cell): return sep_end_by(cell, match(",")) def csv(cell): return
sep_end_by(line(cell), match("\n")) ##### testing ##### print
json.parseString("{ 'a' : -1.0, 'b' : 2.0, 'z' : { 'c' : [1.0, [2.0, [3.0]]] } }") print
csv(number).parseString("1,2,3\n4,5,6") print csv(json.parseString("{ 'a' : 'A'}, [1, 2, 3], 'zzz'\n-1.0,2.0,-3.0"))
```

<http://www.valuedlessons.com/2008/02/pysec-monadic-combinatoric-parsing-in.html>

The Easy Way to Print Flash Cards (with Python and ImageMagick)

Here's a valued lessons that I learned the long and hard way: how to easily print flash cards. Don't waste any time doing it the hard way like I did. Follow my easy route and save yourself some time.

The other day, my wife had a very simple computer need: print and cut pictures into little uniform cards, like flash cards with pictures. It sounds like a simple problem, but there's a simple way and an easy way, and I'd like to share with you the easy way.

The simple way is what she did: use some software like [Scribus](#) (in her case) or Microsoft Word (if she were almost anyone else), copy the pictures in, manually align them, and print. That works great ... until you try and do 700 pictures (about 80 pages of 9 pictures per page). Scribus ate all of the computer's memory, the hard drive thrashed like mad, everything slowed down, and my poor wife spent hours putting it all together. I doubt Word would have fared much better.

But it still wasn't over. When she tried to print, the printer printed out one page with "error" on it. So, we exported to a PDF, and tried printing that: "error" again. So, we tried to print one page at a time: after about 5 minutes, it printed. All I could figure was that something bad was going on between the computer and the printer and involving sending these images uncompressed.

At this point, I decided that it was time to call it quits on manual work and use automated work: write a program to do this for me. I'm not about to manually tell 78 pages to individually print. So, with some quick bash command line work, I told it to print each page by itself. That worked for a few pages, but some were still too big and we got the dreaded "error".

Finally, I decided to really roll up my sleeves and write a program like we should have in the first place: in go pictures, out comes a PDF, and you print the PDF. Additionally, I wanted like to control the DPI of the images used to control the data flow to the printer. I did so, made the PDF, printed it and (after a while, still), it printed. Hurray! Even better, now if we wanted to change pictures or change sizes, we'll be able to do it with very little work.

So here's the program. Put some pictures in a directory, run the program, and print the pdf of "flash cards" or "image thumbnails" or "reading lessons" or whatever you want to call them. You could even make this into an Automator task in Mac OS X. It uses [ImageMagick](#), though, so make sure you have that installed first. from subprocess import call MONTAGE = ["montage", "-bordercolor", "white"] CONVERT = ["convert", "-bordercolor", "white"] def montage_files_of_paths(paths, name, columns = 3, rows = 3, width = 10, height = 8, xborder = .33, yborder = .33, density = 200, frame = 5, spacing = 10, intermediate_ext = "png", final_ext = "pdf"): width, height, xborder, yborder = (int(val * density) for val in (width, height, xborder, yborder)) thumbnail_width = int((width - (xborder * 2)) / columns) thumbnail_height = int((height - (yborder * 2)) / rows) call(MONTAGE + ["-tile", "%rx%r" % (rows, columns), "-geometry", "%rx%r+%r+%r" % (thumbnail_width, thumbnail_height, spacing, spacing), "-frame", str(frame), "-density", str(density)] + [path + "[0]" for path in paths] # [0] to ignore animated GIFS + [name + "." + intermediate_ext]) call(CONVERT + ["-border", "%rx%r" % (xborder, yborder), "-density", str(density), "-annotate", "0x0+%r+%r" % (xborder, yborder), name, name + "*" + intermediate_ext, name + "." + final_ext]) import sys if __name__ == "__main__": if len(sys.argv)

<http://www.valuedlessons.com/2008/02/easy-way-to-print-flash-cards-with.html>

Garlic Programmers for Silver Code?

I just read Larry O'Brien's article about [there being no silver programmers](#). He makes the case that although there may be great variance in programmer productivity, "the significant thing is not that some professional programmers are awesome, it's that some professional programmers suck". I admire Larry O'Brien greatly, and I think there is much wisdom in what he says here.

But I think he's missing something.

I have noticed a trend in my own productivity: **my productivity varies greatly with the code base I'm working on**. This variance may be greater than that between individual programmers. While many are rightly concerned with individual and team productivity, no one ever talks about code base productivity. If we do, where does it lead us?

A few years ago, I was working for a small medical software company. One project was a new venture with a new code base. Though new, the code was already difficult to work with. Its design forced the developer to duplicate work. In my first meeting about the project, I was told we had to do lots of "fat fingering". I didn't even know what that meant. It turned out to mean typing almost the exact same thing over and over. I was horrified and ultimately wrote a code generation tool to automate the work. Though creating and maintaining it was extra work in itself, without it, **doing many common tasks in this code easily took 3x longer than necessary**.

After that project was "shelved", I was moved to the company's main product, which was older and had a much more "mature" code base. Actually, half was new-style (similar to what I had been working on) and half was old-style. The old-style code was even worse to work with. It was so bad that I was almost giddy whenever I could work on the new-style code. I'm sure you know what kind of code I'm talking about. **Doing anything in this code easily took 3x even longer**.

Combined, that's 9x slower. That seems too high. Is it even possible? How can a code base reduce productivity so much? Here are a few possibilities I thought of; I'm sure you can think of more: Bad code leads to more code, and more code is slower to work with Bad code is hard to understand Bad code is dangerous to change Bad code leads to duplication of effort Bad code leads to a slow change-compile-test cycle, leading to wasted time and loss of focus Bad code from third party libraries can drive you crazy by crashing, malfunctioning randomly, or having little documentation Bad code saps energy, interest, and morale

A bad code base makes any programmer less productive. Have you ever noticed that when you work from scratch with no code base to weigh you down, you can be much more productive? Imagine that as your baseline productivity. Now think of how long it takes you to accomplish something similar on a project that you're working on right now. How does it compare? I bet you're slower on the existing project.

But wait just a minute. Imagine if my list of the effects of bad code were reversed. What if, instead, it read: Good code leads to less code, and less code is easier to work with Good code is easy to understand Good code is safe to change Good code avoids duplication of effort Good code leads to a quick change-compile-test cycle Good code uses libraries which remove the need for "dirty work" and let you focus on the important problems Good code is fun to work with!

A good code base makes any programmer more productive. Have you ever worked on a project that was so wonderful that you could crank out magnificent results almost effortlessly? Perhaps [it has a great DSL embedded in it](#) or has [powerful infrastructure](#).

Even with conservative estimates, **if bad code slows us down 5x and good code speeds us up just 2x faster, that's easily a 10x difference**.

Maybe silver programmers don't exist. But is there "silver code"? What if there are programmers who are more adept at writing silver code? If they can make a code base that's just 2x easier to work with long-term, then that makes everyone that works on that code 2x more productive, possibly forever. If such programmers exist, then they must be extremely valuable, not so much because of their own productivity, but because of the effect they will have on the productivity of others in the future. Perhaps rather than "silver bullet" programmers that kill werewolves, these are just "garlic programmers" that ward off vampires.

Given that productivity can vary so much with quality of existing code, **perhaps it's worth looking for "garlic programmers" who will write code that will keep the rest of us productive long-term**. I leave it as an exercise for the reader to figure out how to measure a programmer's garlic-ness :).

List Monad in Ruby and Python

Recently, I wrote about ways to add something like Haskell's `do` syntax to programming languages like [python](#) and [ruby](#). The python trick used bidirectional generators and the ruby trick used `callcc`. I have since been notified that there is one serious problem with both of them: the `bindee` (the function given to `bind`) can only be called once. For many monads, this limitation is acceptable, but for some, it means you can't use that nice syntax.

Take the List monad, for example, in ruby: `class ListMonad` It looks great, but `list1` should be `ListMonad([11, 21, 31, 12, 22, 32, 13, 23, 33])`, but we get `ListMonad([11])` instead.

Traditional binding works fine: `list2 = [1, 2, 3].bindb do |x| [10, 20, 30].bindb do |y| x + y end`
`list2` is `ListMonad([11, 21, 31, 12, 22, 32, 13, 23, 33])`. So, we know our monad is fine, but our `with_monad` doesn't work.

```
We have the same problem in python: class ListMonad(Monad): def __init__(self, vals): self.vals = vals def
bind(self, bindee): return self.__class__(concat((self.maybeUnit(bindee(val)) for val in self.vals)))
@classmethod def unit(cls, val): return cls([val]) def __repr__(self): return "ListMonad(%s)" % self.vals def
__iter__(self): return iter(self.vals) def concat(lists): return [val for lst in lists for val in lst] @do(ListMonad)def
with_list_monad(): x = yield [1, 2, 3] y = yield [10, 20, 30] mreturn(x + y) list1 = with_list_monad() def
with_list_monad_binding(): return \ ListMonad([ 1, 2, 3]) >> (lambda x: ListMonad([10, 20, 30]) >> (lambda y:
x + y)) list2 = with_list_monad_binding()
```

Again, it looks good, but `list1` is `ListMonad([11, None, None, None, None])` while `list2` is `ListMonad([11, 21, 31, 12, 22, 32, 13, 23, 33])`. The result of `list1` sure is bizarre.

Luckily, I have found a solution for ruby, and a bad hack that could be construed as a solution for python if you ignore some issues.

In Ruby, the trick is to monkey with the `rbind` continuation so that at the end of `with_monad` it returns control to the point where the continuation is called. In other words, we need to store a second continuation at the point right after the first continuation is called. Confused yet? Continuations will melt your brain, and this took a little while for me to get right: `def with_monad_ext(monad_class, &block) finishers = [] rbind_ext = lambda do
|monad| begin checked_callcc do |outer_cont| monad.bindb do |val| callcc do |inner_cont|
finishers.push(inner_cont) outer_cont.call(val) end end end rescue ContinuationUnused => unused raise
MonadEscape.new(unused.result) end end val = begin monad_class.maybeUnit(block.call(rbind_ext)) rescue
MonadEscape => escape escape.monad end finisher = finishers.pop() if finisher finisher.call(val) else val
end end list3 = with_monad_ext ListMonad do |rbind| x = rbind.call [1,2,3] y = rbind.call [10,20,30] x + y end
Success! list3 is correctly ListMonad([11, 21, 31, 12, 22, 32, 13, 23, 33]). The only real problem is that the normal rbind won't work. We have to make a special one and give it to the block as an argument. This wouldn't be that bad, except that Ruby has this silly limitation so we have to say rbind.call rather than just rbind, which makes it less fun to type. I named it with_monad_ext because I don't think it will be needed as often as with_monad, and with_monad is more convenient.`

Now, back to python. The problem is harder. It's rooted in the fact that for a given iterator, `itr.send()` is not reentrant. But, if we could copy the iterator, that would give us a possible solution. I did a little googling and found that python isn't going to have `copy.copy(itr)` anytime soon because [no one cares about them](#). Some people [have made some progress](#), but it segfaults on my computer (64-bit Linux, in case you're wondering). So, I came up with this terrible little hack which makes it possible to "copy" iterators: `class CopyableIterator:
def __init__(self, generator, log = ()): self.generator = generator self.log = list(log) #hmmm... if the logs were
immutable, we wouldn't have to do this self.iterator = None def getIterator(self): if self.iterator is None:
self.iterator = self.generator() for value in self.log: self.iterator.send(value) return self.iterator def send(self,
value): iterator = self.getIterator() self.log.append(value) return iterator.send(value) def next(self): return
self.send(None) def copy(self): return self.__class__(self.generator, self.log)`

```
That let's us define @do_ext: @decorator_with_argsdef do_ext(func, func_args, func_kargs, Monad):
@handle_monadic_throws(Monad) def run_maybe_iterator(): itr = func(*func_args, **func_kargs) if
isinstance(itr, types.GeneratorType): @handle_monadic_throws(Monad) def send(itr, val): try: # here's the
real magic monad = Monad.maybeNew(itr.send(val)) return monad.bind(lambda val : send(itr.copy(), val))
except StopIteration: return Monad.unit(None) return send(CopyableIterator(lambda : func(*func_args,
**func_kargs)), None) else: #not really a generator if itr is None: return Monad.unit(None) else: return itr
return run_maybe_iterator() @do_ext(ListMonad)def with_list_monad_ext(): x = yield [1, 2, 3] y = yield [10,
```

```
20, 30] mreturn(x + y) list3 = with_list_monad_ext()
```

And list3 is ListMonad([11, 21, 31, 12, 22, 32, 13, 23]). Success again! There is a downside, though. First, all of the calculations are done for all of the combinations. This is especially bad if you do this:

```
@do_ext(ListMonad)def with_list_monad_ext(): print "foo" x = yield [1, 2, 3] y = yield [10, 20, 30] mreturn(x + y)
```

Because now you'll print "foo" 9 times instead of 1.

So there you have it: you can do the List monad with do syntax in python and ruby, but you'll have to decide whether the trade-offs are worth it.

<http://www.valuedlessons.com/2008/01/recently-i-wrote-about-ways-to-add.html>

Monads in Ruby (with nice syntax!)

My last [article](#) was about how you can do monads in python with nice syntax. Now I'd like to present nice monad syntax in Ruby. I'm not explaining what monads are or how you can use them. For now, I'm expecting you to be familiar with monads. If you aren't, go read a [nice article](#) on them.

Imagine you have a Monad base class like this:

```
class Monad def bind(func) raise "not implemented" end def self.unit(val) raise "not implemented" end # bind to block def bindb(&func) bind(func) endend
```

As an aside, Please excuse there being a "bind" and a "bindb". It's silly that Ruby differentiates between a block and Proc like that. I also think it's silly that I have to keep typing "lambda {|val| func(val)}" to turn a method into a Proc and "proc.call(val)" to call a Proc. In python, methods, functions, and lambdas are all the same thing, and it's a lot easier to code with. In this sense, python is way better. But Ruby has really slick syntax for passing in the block, and python's lambda restrictions are annoying. Why can't we have the best of both? I guess then I'd need Scala or Perl6. End aside.

Let's implement the Either monad, which I call it Failable:

```
class Failable Now we can write some code that safely handles divide by zero without using exceptions (actually, exceptions essentially are the Either monad, but never mind that for now): def fdiv(a, b) if b == 0 failure("cannot divide by zero") else success(a / b) endend def fdiv_with_binding(first_divisor) fdiv(2.0, first_divisor).bindb do |val1| fdiv(3.0, 1.0) .bindb do |val2| fdiv(val1, val2) .bindb do |val3| success(val3) end end endend puts fdiv_with_binding(1.0)puts fdiv_with_binding(0.0)
```

Which prints:

```
Success(0.6666666666666667)Failure(cannot divide by zero)
```

But it's not very pretty. Luckily, ruby has callcc, which makes it very easy to define a function which I'll call "rbind" which can do this:

```
def fdiv_with_rbinding(first_divisor) with_monad Failable do val1 = rbind fdiv(2.0, first_divisor) val2 = rbind fdiv(3.0, 1.0) val3 = rbind fdiv(val1, val2) val3 endend def with_monad(monad_class, & block) begin val = block.call() if val.class == monad_class val else monad_class.unit(val) end rescue MonadEscape => escape escape.monad endend def rbind(monad) begin checked_callcc {|cc| monad.bind(cc)} rescue ContinuationUnused => unused raise MonadEscape.new(unused.result) endend class MonadEscape Ruby's block syntax makes it very nice to say "with_monad Monad do", which I like. But I don't really like seeing "= rbind". I'd really like it if we could override "
```

<http://www.valuedlessons.com/2008/01/monads-in-ruby-with-nice-syntax.html>

Monads in Python (with nice syntax!)

Update: I've been informed that I didn't clearly explain what monads are and what the code examples are supposed to do. Sorry. I guess I assumed too much preexposure to monads. If you're not familiar with them, there are already so many tutorials on monads that I'll simply direct you to two of my favorites: [spacesuits](#) and [wikipedia](#). I've also added more explanations of what the code snippets are supposed to do. I hope that helps.

Update 2: By popular demand, I'm including some code that you can actually run :). See the bottom of the article.

Recently, I used monads in production code for a soon-to-be-publicly-released application. Although many think they are strange, esoteric, and perhaps useless, monads were the best way to solve the problem. My code is not in Haskell; it's in Python. I'm not doing anything weird like IO in a purely functional way; I'm just parsing a file.

The crazy, unexpected conclusion I came to is that you can and should use monads in your code in almost any programming language. There are two parts to this: "can" and "should". I think I'll save "should" for another article. Right now, I'm excited to show you how you can.

As a preview for "should", please consider that you may be using monads already without knowing it. [LINQ in C# is a monad \(pdf\)](#), so if you've ever used it, you've used a monad. The C# guys used monads for queries for the same reason I'm using them for parsing: they're the right tool for the job. But unlike them, I can't change the syntax of the programming language.

The biggest challenge with using monads in a "normal" programming language is that monads involve lots of closures. This is exactly the same problem you run into with [CPS](#), which isn't surprising since a monad's "bind" operator is CPS and since continuations can be implemented with monads. By the way, if your programming language doesn't have closures (meaning you are stuck programming in C, C++, or Java), then monads are probably out of the question. Assuming you have closures and use them with monads directly, you end up with code like the following. It's python using the [Maybe monad](#) to handle divide by zero errors. I'm using ">>" (`__rshift__`) overloaded to mean "bind". `def mdiv(a, b): if b == 0: return Nothing else: return Something(a / b) def with_maybe(): return \ mdiv(2.0, 2.0) >> (lambda val1 : mdiv(3.0, 0.0) >> (lambda val2 : mdiv(val1, val2) >> (lambda val3 : Something(val3))))`

That's not very pretty. We need a way to clean that up. How we can do so depends on the programming language. Haskell has "do" syntax built-in, which makes monadic code look like an imperative language or even a list comprehension. Ruby and Scheme have `call/cc` which makes it trivial to wrap a bind call with a continuation to make any monadic code look "normal". C# has LINQ, which is practically Haskell's do notation with funny names.

But I'm using python. What does python have? Luckily for me, in python 2.5 they added bidirectional generators, and I found a way to use them to make something like "do" notation. Now we can write the above code like this (`@do` and `mreturn` are defined later): `@do(Maybe)def with_maybe(first_divisor): val1 = yield mdiv(2.0, 2.0) val2 = yield mdiv(3.0, 0.0) val3 = yield mdiv(val1, val2) mreturn(val3)`

I even copied the names "do" and "return" from Haskell, although I had to spell "return" as "mreturn". All you really have to remember is that "yield" means "bind" and that the end result is a monad. There are limitations to this technique, but it's working very well for me so far. I've implemented the Maybe monad, the Error monad, the StateChanger monad, and the Continuation monad (which will require another article to explain).

I particularly like the continuation monad because it allows me to write `callcc`, which lets me do threadless actors (message passing) in python: `from collections import deque class Mailbox: def __init__(self): self.messages = deque() self.handlers = deque() def send(self, message): if self.handlers: handler = self.handlers.popleft() handler(message)() else: self.messages.append(message) def receive(self): return callcc(self.react) @do(ContinuationMonad) def react(self, handler): if self.messages: message = self.messages.popleft() yield handler(message) else: self.handlers.append(handler) done(ContinuationMonad.zero()) @do(ContinuationMonad)def insert(mb, values): for val in values: mb.send(val) @do(ContinuationMonad)def multiply(mbin, mbout, factor): while True: val = (yield mbin.receive()) mbout.send(val * factor) @do(ContinuationMonad)def print_all(mb): while True: print (yield mb.receive()) original = Mailbox()multiplied = Mailbox() print_all(multiplied)multiply(original, multiplied, 2)insert(original, [1, 2, 3])()`

A few months ago, I wrote a similar implementation of threadless actors in python. It used generators in a

similar way, but it was 10 times as much code. I was shocked at how short this implementation ended up being. You might think that it's because the continuation monad implementation is big. Nope. It's just as short (Monad defined later, and Record defined [here](#)):

```
class ContinuationMonad(Record("run"), Monad):
    def __call__(self, cont = lambda a : a):
        return self.run(cont)
    def bind(self, bindee):
        return ContinuationMonad(lambda cont : self.run(lambda val : bindee(val).run(cont)))
    @classmethod
    def unit(cls, val):
        return cls(lambda cont : cont(val))
    @classmethod
    def zero(cls):
        return cls(lambda cont : None)
    def callcc(usecc):
        return ContinuationMonad(lambda cont : usecc(lambda val : ContinuationMonad(lambda _ : cont(val))).run(cont))
```

So, you can use monads with elegant syntax in any language that has closures and any of the following: do syntax (Haskell, C#)call/cc (Scheme, Ruby)bidirectional generators (Python 2.5, a future JavaScript?)coroutines (Lua, Io)

The only thing I haven't shown you is the implementation of Monad, @do, mreturn, and done. It has a few nasty details related to using generators and decorators in python, but here's the gist of it:

```
class Monad:
    def bind(self, func):
        raise NotImplementedError
    def __rshift__(self, bindee):
        return self.bind(bindee)
    def __add__(self, bindee_without_arg):
        return self.bind(lambda _ : bindee_without_arg())
    @decorator_with_args
    def do(func, func_args, func_kargs, Monad):
        itr = func(*func_args, **func_kargs)
        def send(val):
            try:
                monad = itr.send(val)
                return monad.bind(send)
            except MonadReturn, ret:
                return Monad.unit(ret.value)
            except Done, done:
                return done.monad
        return send(None)
    def mreturn(val):
        raise MonadReturn(val)
    def done(val):
        raise Done(val)
```

That's it. If you've made it all the way to the end of this long article, I hope you've found inspiration for using monads in your own applications, especially if you are coding in python. If so, here's some code that you can run:

```
import types
##### Base Monad and @do syntax#####
class Monad:
    def bind(self, func):
        raise NotImplementedError
    def __rshift__(self, bindee):
        return self.bind(bindee)
    def __add__(self, bindee_without_arg):
        return self.bind(lambda _ : bindee_without_arg())
    def make_decorator(func, *dec_args):
        def decorator(undecorated):
            def decorated(*args, **kargs):
                return func(undecorated, args, kargs, *dec_args)
            decorated.__name__ = undecorated.__name__
            return decorated
        decorator.__name__ = func.__name__
        return decorator
    def make_decorator_with_args(func):
        def decorator_with_args(*dec_args):
            return make_decorator(func, *dec_args)
        return decorator_with_args
    decorator = make_decorator
    decorator_with_args = make_decorator_with_args
    @decorator_with_args
    def do(func, func_args, func_kargs, Monad):
        @handle_monadic_throws(Monad)
        def run_maybe_iterator():
            itr = func(*func_args, **func_kargs)
            if isinstance(itr, types.GeneratorType):
                @handle_monadic_throws(Monad)
                def send(val):
                    try:
                        # here's the real magic
                        monad = itr.send(val)
                        return monad.bind(send)
                    except StopIteration:
                        return Monad.unit(None)
                    return send(None)
                else:
                    #not really a generator if itr is None:
                    return Monad.unit(None)
                else:
                    return itr
            return run_maybe_iterator()
        @decorator_with_args
        def handle_monadic_throws(func, func_args, func_kargs, Monad):
            try:
                return func(*func_args, **func_kargs)
            except MonadReturn, ret:
                return Monad.unit(ret.value)
            except Done, done:
                assert isinstance(done.monad, Monad)
                return done.monad
        class MonadReturn(Exception):
            def __init__(self, value):
                self.value = value
        Exception.__init__(self, value)
        class Done(Exception):
            def __init__(self, monad):
                self.monad = monad
        Exception.__init__(self, monad)
        def mreturn(val):
            raise MonadReturn(val)
        def done(val):
            raise Done(val)
        def fid(val):
            return val
        ##### Failable Monad #####
        class Failable(Monad):
            def __init__(self, value, success):
                self.value = value
                self.success = success
            def __repr__(self):
                if self.success:
                    return "Success(%r)" % (self.value,)
                else:
                    return "Failure(%r)" % (self.value,)
            def bind(self, bindee):
                if self.success:
                    return bindee(self.value)
                else:
                    return self
            @classmethod
            def unit(cls, val):
                return cls(val, True)
        class Success(Failable):
            def __init__(self, value):
                Failable.__init__(self, value, True)
        class Failure(Failable):
            def __init__(self, value):
                Failable.__init__(self, value, False)
        def failable_monad_example():
            def fdiv(a, b):
                if b == 0:
                    return Failure("cannot divide by zero")
                else:
                    return Success(a / b)
            @do(Failable)
            def with_failable(first_divisor):
                val1 = yield fdiv(2.0, first_divisor)
                val2 = yield fdiv(3.0, 1.0)
                val3 = yield fdiv(val1, val2)
            mreturn(val3)
            print with_failable(0.0)
            print with_failable(1.0)
        ##### StateChanger Monad #####
        class StateChanger(Monad):
            def __init__(self, run):
                self.run = run
            def bind(self, bindee):
                run0 = self.run
                def run1(state0):
                    (result, state1) = run0(state0)
                    return bindee(result).run(state1)
                return StateChanger(run1)
            @classmethod
            def unit(cls, val):
                return cls(lambda state : (val, state))
            def get_state(view = fid):
                return change_state(fid, view)
            def change_state(changer, view = fid):
                def make_new_state(old_state):
                    new_state = changer(old_state)
                    viewed_state = view(old_state)
                    return (viewed_state, new_state)
                return StateChanger(make_new_state)
            def state_changer_monad_example():
                @do(StateChanger)
                def
```

```

dict_state_copy(key1, key2): val = yield dict_state_get(key1) yield dict_state_set(key2, val) mreturn(val)
@do(StateChanger) def dict_state_get(key, default = None): dct = yield get_state() val = dct.get(key, default)
mreturn(val) @do(StateChanger) def dict_state_set(key, val): def dict_set(dct, key, val): dct[key] = val return
dct new_state = yield change_state(lambda dct: dict_set(dct, key, val)) mreturn(val) @do(StateChanger) def
with_dict_state(): val2 = yield dict_state_set("a", 2) yield dict_state_copy("a", "b") state = yield get_state()
mreturn(val2) print with_dict_state().run({}) # (2, {"a" : 2, "b" : 2}) ##### Continuation Monad #####
class ContinuationMonad(Monad): def __init__(self, run): self.run = run def __call__(self, cont = fid): return
self.run(cont) def bind(self, bindee): return ContinuationMonad(lambda cont : self.run(lambda val :
bindee(val).run(cont))) @classmethod def unit(cls, val): return cls(lambda cont : cont(val)) @classmethod def
zero(cls): return cls(lambda cont : None) def callcc(usecc): return ContinuationMonad(lambda cont :
usecc(lambda val : ContinuationMonad(lambda _ : cont(val))).run(cont)) def continuation_monad_example():
from collections import deque class Mailbox: def __init__(self): self.messages = deque() self.handlers =
deque() def send(self, message): if self.handlers: handler = self.handlers.popleft() handler(message)() else:
self.messages.append(message) def receive(self): return callcc(self.react) @do(ContinuationMonad) def
react(self, handler): if self.messages: message = self.messages.popleft() yield handler(message) else:
self.handlers.append(handler) done(ContinuationMonad.zero()) @do(ContinuationMonad) def insert(mb,
values): for val in values: mb.send(val) @do(ContinuationMonad) def multiply(mbin, mbout, factor): while
True: val = (yield mbin.receive()) mbout.send(val * factor) @do(ContinuationMonad) def print_all(mb): while
True: print (yield mb.receive()) original = Mailbox() multiplied = Mailbox() print_all(multiplied)()
multiply(original, multiplied, 2)() insert(original, [1, 2, 3])() if __name__ == "__main__":
failable_monad_examle() state_changer_monad_example() continuation_monad_example()

```

<http://www.valuedlessons.com/2008/01/monads-in-python-with-nice-syntax.html>

Immutable Data in Python (Record or Named Tuple)

A valued lesson that I have learned the hard way is that mutable data can get nasty, and that it's really nice to have immutable data structures. I've been writing a lot of python recently, and after a while I realized that I was writing a lot of this: `class SomeDataStructure: def __init__(self, arg1, arg2): self.prop1 = arg1 self.prop2 = arg2`

Not only was this repetitive, but I found that little mutability bugs were cropping up on me. It's just too easy to trip over them when the default is mutability, as it is in python. In fact, immutable data structures aren't even a built-in option in python.

Finally the pain of using mutable data structures grew too large. I looked for a way and couldn't find anything, so I created my own. It supports getters, setters, inheritance, default values, altering several values at once, and I think the syntax is nice. I've been using it for everything the last few months and it's been great. It's helped with code repetition, concurrency, and serialization.

I hope you can learn from my valued lesson and not repeat my mistakes. Here's how you use it. `class Person(Record("name", "age")): pass class OldPerson(Person): @classmethod def prepare(cls, name, age = None): return (name, age) peter = Person("Peter", 26) wes = Person("Wes", 28) grandpa = OldPerson("Bubba") wes2 = wes.setAge(29) wes3 = wes.alter(name = "Grandpa", age = 57) print peter, grandpa, wes.name, wes2.age`

Here is the code. This is actually a simplified version of the original that I rewrote on my own time. I use a more complete/complex version in the code I'm writing for my employer. `def Record(*props): class cls(RecordBase): pass cls.setProps(props) return cls class RecordBase(tuple): PROPS = () def __new__(cls, *values): if cls.prepare != RecordBase.prepare: values = cls.prepare(*values) return cls.fromValues(values) @classmethod def fromValues(cls, values): return tuple.__new__(cls, values) def __repr__(self): return self.__class__.__name__ + tuple.__repr__(self) ## overridable @classmethod def prepare(cls, *args): return args ## setting up getters and setters @classmethod def setProps(cls, props): for index, prop in enumerate(props): cls.setProp(index, prop) cls.PROPS = props @classmethod def setProp(cls, index, prop): getter_name = prop setter_name = "set" + prop[0].upper() + prop[1:] setattr(cls, getter_name, cls.makeGetter(index, prop)) setattr(cls, setter_name, cls.makeSetter(index, prop)) @classmethod def makeGetter(cls, index, prop): return property(fget = lambda self : self[index]) @classmethod def makeSetter(cls, index, prop): def setter(self, value): values = (value if current_index == index else current_value for current_index, current_value in enumerate(self)) return self.fromValues(values) return setter`

For comparison, I've seen similar ideas at

<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/500261> and

<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/303439> but I don't like their implementation or design as much, especially since they lack proper setters.

I've also read that future versions of Python will have NamedTuples, which is something I wish it had already.

<http://www.valuedlessons.com/2007/12/immutable-data-in-python-record-or.html>
