ActiviPy Documentation

Release 0.2.dev

Christopher Allan Webber and ActiviPy contributors

July 28, 2016

Contents

Activipy (pronounced "activi-pie") is an ActivityStreams 2.0 toolkit for Python. It helps making your application or website social while keeping things fun.

ActivityStreams is a JSON (and optionally, json-ld) representation of "who's doing what", simple to understand and easy to implement on your own website. ActiviPy makes things extra easy for you by including the full basic ActivityStreams vocabulary, which means that you have the appropriate language to discribe common social network type interactions. This makes it easy to build a world where various servers can speak the same language about common activities. If your application does something more complicated, don't worry, ActivityStreams has an extension mechanism, and Activipy is built to help you take advantage of it. Jump on in!

Contents:

CHAPTER 1

About Activipy

1.1 What is ActivityStreams? How might it help me?

Activipy is a simple library which makes working with ActivityStreams easy. Which is a bit confusing, because ActivityStreams is already dead-easy. Activipy helps us make things a bit more robust.

How easy is ActivityStreams? Here's a valid ActivityStreams document from a mad scientist type person:

```
{"@id": "http://drboss.example/api/objects/123415/",
 "@type": "Create",
 "actor": {
     "@type": "Person",
     "@id": "http://drboss.example/me/",
     "displayName": "Professor BossManager"},
 "to": ["http://employeemine.example/jim/",
        "http://employeemine.example/sarah/"],
 "object": {
     "@type": "Note",
     "@id": "http://drboss.example/rants/power-productivity/",
     "content": "Thanks to my latest invention, productivity is up 1 million percent! MWAHAHA!!"}}
```

Here's another:

```
{"@id": "http://employeemine.example/sarah/blog/new-job-please/"
 "@type": "Note",
 "displayName": "New job, please.",
 "content": "Anyone know where I can get a job not managed by a mad scientist?"}
```

These objects aren't complex, and anyone can read them. They're just JSON, plus some common vocabulary, including a convention that "@id" is the identifier of the object and "@type" is the vocabulary type. That's pretty simple!

And simple is good, because let's face it, most users of most web application APIs are like poor Billy Scripter, a kid who has some scripting language like Ruby or Python or Javascript and some JSON parser in a toolbox and that's about it. Billy Scripter knows how to parse JSON pulled down from some endpoint, and that's about all he knows how to do. Poor Billy Scripter! But it's okay, because ActivityStreams is simple enough that Billy can make it by. And because the ActivityStreams Core serialization specifies that the ActivityStreams Vocabulary is always implied and that those terms must always be available, Billy will always know what a Like object or a Note means. Horray for Billy!

Meanwhile, you get the benefit of a well thought out "subject, predicate, object" type structure (or in other words, "who did what"). The Core Classes provide the basic structure, and that general structure combined with the very minimal rules of ActivityStreams Core (roughly speaking, it's just JSON with the ActivityStreams vocabulary implied, but if you need anything more, you can use json-ld) means that you already have the general structure of all the "social"

type activities that happen on the modern web. Throw in the Extended Classes and you've got all the right language to express what your users are doing too. Awesome.

But hey, what happens when the base ActivityStreams vocabulary just isn't enough? Maybe you're running a social network that's also a game and you need some way to express that your players are beating up goblins (wait, I take that back, don't beat up poor goblins!), or you've got a highly interactive e-commerce site where users can share coupons or whatever. Look, whatever it is, you can express it in ActivityStreams!

Yeah, I said it! Sure, ActivityStreams gives you almost everything you need out of the box, but if you need to get fancy and define your own terms, you can do that too. ActivityStreams is technically a json-ld document with an implied vocabulary, but you can add on new vocabularies too. Need to add Coupon objects, or RpgBeatemup activities? Yeah, you can do it! And for all you RDF fans out there, you can transform ActivityStreams objects into a full-on linked-data graph. Go wild!

But you also don't *have* to. Thanks to the promises made by the ActivityStreams serialization, Billy Scripter can make it by just fine with his Ruby and JSON toolkit. And so can most of the rest of the web!

1.2 How Activipy and ActivityStreams work together

So if ActivityStreams is just JSON (and optionally json-ld), what do you need Activipy for? Good question!

Activipy provides a whole suite of useful tools, including friendly and Pythonic constructors, a flexible and extensible method dispatch system, and much more. We could express the above like this:

```
>>> from activipy import vocab
>>> vocab.Create(
...   "http://drboss.example/api/objects/123415/",
...   actor=vocab.Person(
...     "http://drboss.example/me/",
...     displayName="Professor BossManager"),
...   to=["http://employeemine.example/jim/",
...       "http://employeemine.example/sarah/"],
...   object=vocab.Note(
...     http://drboss.example/rants/power-productivity/",
...     content="Thanks to my latest invention, productivity is up 1 million percent! MWAHAHA!!"))
```

If we were writing our own diary application, we could specify an environment that knows how to post notes to it:

```
>>> from ourjournal import DiaryEnv
>>> dear_diary = DiaryEnv.c.Note(
...     displayName="New job, please.",
...     content="Anyone know where I can get a job not managed by a mad scientist?")
>>> dear_diary.m.post()
```

Activipy provides you with the basic tools you need to map ActivityStreams to the world of your Python application. And if you're just starting out in writing a brand new social network application? You'd better believe Activipy is a good place to start!

Activipy means making your networked application social is easy. And best of all, you can speak a common language with other ActivityStreams speaking applications across the net.

Sound good? *Let's get started!*

CHAPTER 2

An Activipy tutorial

This tutorial is fairly robust and is broken into sections. Each one of them builds upon each other like a ladder, but you can step off at any time. In this tutorial you'll learn:

1. How to use Activipy to express and share basic social networking features.
2. How to *do* cool things with Activipy using the method dispatch system.
3. Go beyond the basics and use extended vocabulary.
4. Why Activipy objects are "immutable" (or are intended to be) and how to use that to your advantage.

You can read as much or as little as you like, but we hope this tutorial is engaging enough where you won't be able to stop reading until the end!

2.1 Sweet beginnings

So say you and your friends are all conspiring to meet up for root beer floats. How could we structure that in Python? First, let's look at the ActivityStreams representation of this note, and then we'll look at how we got there.

What vocabulary do we want to use? Let's look at what comes out of the box:

```
>>> vocab.Create.notes
'Indicates that the actor has created the object.'
>>> vocab.Person.notes
'Represents an individual person.'
>>> vocab.Note.notes
'Represents a short work typically less than a single paragraph in length.'
```

Those sound like the things we mean. Great! It's nice that Activipy includes the notes from the Activity Vocabulary so it's easy for us to keep track of what things mean.

So it turns out we can use these vocabulary definitions as friendly constructors:

```
# gives us the core vocabulary
from activipy import vocab

post_this = vocab.Create(
    "http://tsyesika.co.uk/act/foo-id-here/",
    actor=vocab.Person(
        "http://tsyesika.co.uk/",
        displayName="Jessica Tallon"),
    to=["acct:cwebber@identi.ca",
        "acct:justaguy@rhiaro.co.uk",
```

```
        "acct:ladyaeva@hedgehog.example"],
    object=vocab.Note(
        "http://tsyesika.co.uk/chat/sup-yo/",
        content="Up for some root beer floats?"))
```

Oh, okay, that's pretty easy to read! We can see that we've specified who we are, who we want to send the message to, and the actual message we're posting.

What does our message look like? Let's see:

```
>>> post_this.json()
{"@id": "http://tsyesika.co.uk/act/foo-id-here/",
 "@type": "Create",
 "actor": {
     "@type": "Person",
     "@id": "http://tsyesika.co.uk/",
     "displayName": "Jessica Tallon"},
 "to": ["acct:cwebber@identi.ca",
        "acct:justaguy@rhiaro.co.uk",
        "acct:ladyaeva@hedgehog.example"],
 "object": {
     "@type": "Note",
     "@id": "http://tsyesika.co.uk/chat/sup-yo/",
     "content": "Up for some root beer floats?"}}
```

Oh interesting! That looks pretty similar to the Python constructor version. In fact, we could have built this from the json itself:

```
>>> from activipy import core, vocab
>>> post_this = core.ASObj({
...     "@type": "Create",
...     "@id": "http://tsyesika.co.uk/act/foo-id-here/",
...     "actor": {
...         "@type": "Person",
...         "@id": "http://tsyesika.co.uk/",
...         "displayName": "Jessica Tallon"},
...     "to": ["acct:cwebber@identi.ca",
...            "acct:justaguy@rhiaro.co.uk",
...            "acct:ladyaeva@hedgehog.example"],
...     "object": {
...         "@type": "Note",
...         "@id": "http://tsyesika.co.uk/chat/sup-yo/",
...         "content": "Up for some root beer floats?"}},
...   vocab.BasicEnv)
<ASObj Create>
```

Hm! So it's nice to have "pythonic" constructors, but this json representation isn't so complex... is it worth having a whole library just for this? Let's see what else Activipy gives us.

Activipy gives simple dictionary-style access:

```
>>> post_this["to"]
['acct:cwebber@identi.ca', 'acct:justaguy@rhiaro.co.uk', 'acct:ladyaeva@hedgehog.example']
```

Helpful, but we could have gotten that from running .json() and pulling out the right values! But this is kinda nice:

```
>>> root_beer_note = post_this["object"]
>>> root_beer_note
<ASObj Note "http://tsyesika.co.uk/chat/sup-yo/">
```

Cool, we've extracted the actual object we were going to post, and it came back wrapped in an ASObj object. Of course, we could always get the json version of this if we wanted:

```
>>> root_beer_note.json()
{'@id': 'http://tsyesika.co.uk/chat/sup-yo/',
 '@type': 'Note',
 'content': 'Up for some root beer floats?'}
```

What kind of type is our newly extracted *root_beer_note*? Let's see:

```
>>> root_beer_note.types
['Note']
```

Wait, "types", as in possibly plural? That's right, an ActivityStreams object's "type" is actually a "composite type". It turns out this is useful when handling extensions to the vocabulary, but we'll come back to that later.

Strings are less fun as types than ASTypes, so can we get that back? We sure can:

```
>>> root_beer_note.types_astype
[<ASType Note>]
```

But hey, what's this thing:

```
>>> root_beer_note.types_expanded
['http://www.w3.org/ns/activitystreams#Note']
```

Huh? A URL? This starts to hint at something more complicated... something to do with extensions! But we're getting ahead of ourselves. Extension stuff comes later! Right now we're itching to *do* something with these objects... so what can we do, and how do we do it?

2.2 Methods for our madness

2.2.1 New environments, new friends

Before we hop right into methods, a quick refresher. Remember when we said we could have built our post_this object like this?

```
>>> from activipy import core, vocab
>>> post_this = core.ASObj({
...     # ... json stuff here
...   },
...   vocab.BasicEnv)
<ASObj Create>
```

Wait, what's that BasicEnv thing hanging off the end? That's pretty curious. What does it do?

Since we're passing into the object, it's a good (and correct) guess to assume that the ASObj instance has access to it:

```
>>> post_this.env
<activipy.core.Environment at 0x7fecf206f5c0>
```

The environment helps us in a few ways. For one thing, it contains a set of vocabulary that our environment "knows" about. In fact, we could even do constructors that implicitly pass in the environment directly from the environment itself, using the vocabulary it's been informed of:

```
>>> env = vocab.BasicEnv
>>> root_beer_note = env.c.Note(
...   "http://tsyesika.co.uk/chat/sup-yo/",
```

```
...   content="Up for some root beer floats?")
>>> root_beer_note
<ASObj.Note>
```

Well that's pretty neat! It looks like the *Environment.c* accessor is a friendly way to access vocabulary classes. Cool! So you can already guess at one purpose for environments: if your application is working with an extended vocabulary, it's possible for Activipy to "know" about your vocabulary while determining types, etc. The BasicEnv is, as you would expect, the default and most minimal environment, containing the core vocabulary and nothing else. For many applications, this is all you need. If your application needs additional terminology, we will cover this later in the manual, but for now, we will only concern ourselves with the core vocabulary.

Even if we stick with the core vocabulary, we may wish to use a different environment than BasicEnv. Why? Well we keep saying that we want to *do* something with our applications. Aside from mapping vocabulary, *Environment* objects can contain a mapping of methods!

So, we want to try something... what would be a good demo? How about storing things! Sounds good to me! In our case we're going to simply serialize ActivityStreams objects to json and dump them in and out of a minimalist key-value dbm database. (Note: this will be a lot more efficient if you install the Python bindings for gdbm.)

Our dbm demo module contains a JsonDBM wrapper which conveniently serializes/deserializes to/from json when pulling things in/out of the database. Let's give it a spin so we know what we're working with:

```
>>> from activipy.demos import dbm
>>> db = dbm.JsonDBM.open("/tmp/test.db")
>>> db["foo"] = {"cat": "meow", "dog": "woof"}
>>> db["foo"]
{'cat': 'meow', 'dog': 'woof'}
>>> "foo" in db
True
>>> del db["foo"]
>>> "foo" in db
False
```

Okay, so that's a pretty easy to use key-value store! We could clearly dump our ASObj objects to json and manually save them into here. It would be nice if there was a "save" method that could do that for us though. How could such a save method be made available?

2.2.2 Save one for me, please

Ah, here's a use for Environments! You see, an *Environment* object not only contains information about vocabulary, it contains information about methods as well. As it turns out, we have a handy environment ready for you to play with which knows how to work with a *JsonDBM* wrapped database. Let's try it!

```
>>> env = dbm.DbmEnv
>>> note = env.c.Note("http://example.org/notes/cookie-time/",
...                   content="I really want a cookie!")
>>> note.m.save(db)
>>> db["http://example.org/notes/cookie-time"]
{'@id': 'http://example.org/notes/cookie-time/',
 '@type': 'Note',
 'content': 'I really want a cookie!'}
```

Hey, it worked! That sure was handy... we got a .save() method attached right to our Note! How about a .delete()?

```
>>> "http://example.org/notes/cookie-time" in db
True
>>> note.m.delete(db)
```

```
>>> "http://example.org/notes/cookie-time" in db
False
```

How convenient!

You may notice that we don't call *note.save()* or *note.delete()*; instead, we call *note.m.save()* and *note.m.delete()*! That's because the *.m* attribute is a proxy object to all the methods the *ASObj.env* knows about (in this case, DbmEnv):

```
>>> dbm.DbmEnv.methods
{(<MethodId save>, <ASType Object>): <function activipy.demos.dbm.dbm_save>,
 (<MethodId delete>, <ASType Object>): <function activipy.demos.dbm.dbm_delete>}
```

In fact, we could have used DbmEnv.m instead:

```
>>> dbm.DvmEnv.m.save(note, db)
>>> # is the same as
>>> note.m.save(db)
```

But that's way more verbose! Why not just use *note.m.save(db)* instead? So convenient!

What happens if we save a more complicated, nested note to the db? Remember our root beer float friend?

```
>>> post_this.json()
{"@id": "http://tsyesika.co.uk/act/foo-id-here/",
 "@type": "Create",
 "actor": {
     "@type": "Person",
     "@id": "http://tsyesika.co.uk/",
     "displayName": "Jessica Tallon"},
 "to": ["acct:cwebber@identi.ca",
        "acct:justaguy@rhiaro.co.uk",
        "acct:ladyaeva@hedgehog.example"],
 "object": {
     "@type": "Note",
     "@id": "http://tsyesika.co.uk/chat/sup-yo/",
     "content": "Up for some root beer floats?"}}
```

So we now remember that when post_this was set up, it used the general purpose environment. This means that there is no *post_this.m.save()* method for us to call, because that method is not set up in the BasicEnv environment. We can't even use *BasicEnv.m.save()*, because Activipy safeguards against this:

```
>>> dbm.DbmEnv.m.save(post_this, db)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/home/cwebber/devel/activipy/activipy/core.py", line 464, in method_dispatcher
  method = self.asobj_get_method(asobj, method_id)
File "/home/cwebber/devel/activipy/activipy/core.py", line 550, in asobj_get_method
  "ASObj attempted to call method with an Environment "
activipy.core.EnvironmentMismatch: ASObj attempted to call method with an Environment it was not bou
```

This makes sense, because different environments provide different vocabularies and handle different methods, and subtle bugs could creep in if we permitted this. Normally this is no problem, one application will in general only make use of a single *Environment* tuned to that application. Even here, it is easy to correct... let's just recast *post_this* to our new environment:

```
>>> post_this = core.ASObj(post_this.json(), dbm.DbmEnv)
```

Now we can save away::

```
>>> post_this.m.save(db)
>>> db["http://tsyesika.co.uk/act/foo-id-here/"]
{"@id": "http://tsyesika.co.uk/act/foo-id-here/",
 "@type": "Create",
 "actor": {
     "@type": "Person",
     "@id": "http://tsyesika.co.uk/",
     "displayName": "Jessica Tallon"},
 "to": ["acct:cwebber@identi.ca",
        "acct:justaguy@rhiaro.co.uk",
        "acct:ladyaeva@hedgehog.example"],
 "object": {
     "@type": "Note",
     "@id": "http://tsyesika.co.uk/chat/sup-yo/",
     "content": "Up for some root beer floats?"}}
```

Hooray, our note is in the database! That's really nice.

2.2.3 Save a few more

But wait, is this really how we want? Notice that this activity contains two nested ActivityStreams objects: *actor* and *object*! Wouldn't it be nice if the *.save()* method was able to be smart about this and "normalize" the data for us, saving the child ActivityStreams objects as their own database references, and pulling them out as needed?

Let's think about this for a moment. We know that the root activity that we're posting here is of the type *Create*. We could look at the ActivityStreams Vocabulary document to find out the inheritance chain, but we don't even have to... Activipy can help us out here:

```
>>> post_this.types_inheritance
[<ASType Create>, <ASType Activity>, <ASType Object>]
```

Looking at this, we know that *Create* is a type of *Activity*, which is itself a type of *Object*. Looking at the vocabulary document, it's clear to us that the *actor* and *object* fields inherit from Activity.

It seems fine to save a general *Object* type as-is as we already are, and indeed, you may have noticed that the save method was operating precisely on this ASType:

```
>>> dbm.DbmEnv.methods
{(<MethodId save>, <ASType Object>): <function activipy.demos.dbm.dbm_save>,
 (<MethodId delete>, <ASType Object>): <function activipy.demos.dbm.dbm_delete>}
```

So, since a *Create is* an Object, of course the basic save happens here. But it's even more an *Activity* than a mere *Object*, and if we think about it, hey! Pretty much on any *Activity* ASType (whether it's a *Create* or a *Delete* or a *Like*...) it would be really nice to normalize the *actor* and *object* fields. Is there a way to specify that we'd like to treat Activity objects a bit differently? Indeed, there is! As you've already guessed, if our *Environment* had a separate method that did something different for *save* on *Activity*, that would be really helpful. And it turns out, we've already supplied you with such an environment:

```
>>> dbm.DbmNormalizedEnv.methods
{(<MethodId save>, <ASType Object>): <function activipy.demos.dbm.dbm_save>,
 (<MethodId save>, <ASType Activity>): <function activipy.demos.dbm.dbm_activity_normalized_save>,
 (<MethodId denormalize>, <ASType Object>): <function activipy.demos.dbm.dbm_denormalize_object>,
 (<MethodId denormalize>, <ASType Activity>): <function activipy.demos.dbm.dbm_denormalize_activity>,
 (<MethodId delete>, <ASType Object>): <function activipy.demos.dbm.dbm_delete>}
```

Neat, this does indeed provide us with a separate method for Activity. Let's switch to using the *DbmNormalizedEnv* instead and cast *post_this* to use it (again, you wouldn't normally need to do this in an application that uses just one environment):

```
>>> env = dbm.DbmNormalizedEnv
>>> post_this = core.ASObj(post_this.json(), dbm.DbmNormalizedEnv)
```

Now what happens if we save the object?

```
>>> post_this.m.save(db)
>>> db["http://tsyesika.co.uk/act/foo-id-here/"]
{'@id': 'http://tsyesika.co.uk/act/foo-id-here/',
 '@type': 'Create',
 'actor': 'http://tsyesika.co.uk/',
 'object': 'http://tsyesika.co.uk/chat/sup-yo/',
 'to': ['acct:cwebber@identi.ca',
        'acct:justaguy@rhiaro.co.uk',
        'acct:ladyaeva@hedgehog.example']}
>>> db["http://tsyesika.co.uk/"]
{'@id': 'http://tsyesika.co.uk/',
 '@type': 'Person',
 'displayName': 'Jessica Tallon'}
>>> db["http://tsyesika.co.uk/chat/sup-yo/"]
{'@id': 'http://tsyesika.co.uk/chat/sup-yo/',
 '@type': 'Note',
 'content': 'Up for some root beer floats?'}
```

Awesome... that is *exactly* what we were hoping for!

2.2.4 There and back again

Just to bring things full circle, here's a method that demonstrates pulling an object out of the database:

```
>>> def dbm_fetch(id, db, env):
...     return core.ASObj(db[id], env)
...
>>> normalized_post = dbm_fetch("http://tsyesika.co.uk/act/foo-id-here/",
...                             db, dbm.DbmNormalizedEnv)
>>> normalized_post
<ASObj Create "http://tsyesika.co.uk/act/foo-id-here/">
>>> normalized_post.json()
{'@id': 'http://tsyesika.co.uk/act/foo-id-here/',
 '@type': 'Create',
 'actor': 'http://tsyesika.co.uk/',
 'object': 'http://tsyesika.co.uk/chat/sup-yo/',
 'to': ['acct:cwebber@identi.ca',
        'acct:justaguy@rhiaro.co.uk',
        'acct:ladyaeva@hedgehog.example']}
```

We could make use of the environment's denormalize method:

```
>>> normalized_post.m.denormalize(db)
<ASObj Create "http://tsyesika.co.uk/act/foo-id-here/">
>>> normalized_post.m.denormalize(db).json()
{'@id': 'http://tsyesika.co.uk/act/foo-id-here/',
 '@type': 'Create',
 'actor': {'@id': 'http://tsyesika.co.uk/',
           '@type': 'Person',
           'displayName': 'Jessica Tallon'},
 'object': {'@id': 'http://tsyesika.co.uk/chat/sup-yo/',
            '@type': 'Note',
            'content': 'Up for some root beer floats?'},
```

```
 'to': ['acct:cwebber@identi.ca',
        'acct:justaguy@rhiaro.co.uk',
        'acct:ladyaeva@hedgehog.example']}
```

Hey look, it's our original post back, with the *actor* and *object* filled in! This time, they were extracted from their own entries' key-value pairs in the database. Neat!

And finally, we could simplify this whole thing, and write a method to pull data out of the database in a denormalized fashion, making use of our environment's denormalize methods:

```
>>> def dbm_fetch_denormalized(id, db, env):
...     return env.m.denormalize(
...         dbm_fetch(id, db, env), db)
...
>>> denormalized_post = dbm_fetch_denormalized(
...     "http://tsyesika.co.uk/act/foo-id-here/",
...     db, dbm.DbmNormalizedEnv)
>>> denormalized_post
<ASObj Create "http://tsyesika.co.uk/act/foo-id-here/"
>>> denormalized_post.json()
{'@id': 'http://tsyesika.co.uk/act/foo-id-here/',
 '@type': 'Create',
 'actor': {'@id': 'http://tsyesika.co.uk/',
           '@type': 'Person',
           'displayName': 'Jessica Tallon'},
 'object': {'@id': 'http://tsyesika.co.uk/chat/sup-yo/',
            '@type': 'Note',
            'content': 'Up for some root beer floats?'},
 'to': ['acct:cwebber@identi.ca',
        'acct:justaguy@rhiaro.co.uk',
        'acct:ladyaeva@hedgehog.example']}
```

Whew, what a round trip!

2.2.5 A word to the enwisened

This whole process above of calling the appropriate methods for the appropriate type (or in our case, ASType) is called "method dispatch". You may have noticed that we do things fairly differently from most Python libraries, which usually use Python's native classes as an inheritance chain, something like this:

```
class Object(ASClass):
    class_id = "http://www.w3.org/ns/activitystreams#Object"
    # bla bla
    def save(self, db):
         # save thing here
         pass

class Activity(Object):
    class_id = "http://www.w3.org/ns/activitystreams#Activity"
    # more bla bla
    def save(self, db):
         # save a bit differently
         pass

class Create(Activity):
    class_id = "http://www.w3.org/ns/activitystreams#Create"

# also define Note, etc here
```

We aren't doing that... we're using this intermediate *Environment* thing instead, and ASObj instances are all just instances of ASObj. Why? Why not just use Python's normal class hierarchy? Why have an *Environment* at all?

There are a few reasons:

1. ActivityStreams technically has "composite types"... an "@type" can actually have *multiple* values set here, and the functionality provided by the ASObj will be a union of those types. Because of this, Python's classes really don't work at all to track inheritance. Luckily, there are other benefits of going with an *Environment*....
2. Different applications need to do different things. It's useful to have a general way of handling method dispatch that appropriately pays respect to the inheritance system of the ActivityStreams base vocabulary, and it's nice to make that as Pythonic as possible, but you might never save ActivityStreams objects to a DBM store (very few production applications would). You very well may store objects to an SQL database, or some object store, or who knows what. You may wish to use Activipy for a desktop client or a server application, and those might do very different things. What methods you specify are up to you, but Environments are built in such a way that sharing methods between them, picking and choosing the ones useful to you, and defining entirely new methods is easy.
3. The core vocabulary is good enough for most social web applications, but not for all. You may well need to define entirely new vocabulary, and Activipy allows you to do this. Allowing a user to define their own Environment means that this is not difficult to do, and how to transfer to those to the appropriate ASType representations (and then to know how to operate upon them) is very possible.
4. We didn't get into it here, but fancier method handling can also be done than just the traditional "dispatch hands off to a single procedure" approach. You can also set up a method with a handler which maps or folds over the methods provided through the method mapping and inheritence chain. (Cool, but advanced, stuff! You can imagine how this might be helpful for writing a validator, or etc.)

There's more to say on these subjects, but hopefully this section helped put some of this into perspective. Hopefully the rest of this will become clear shortly, including how to expand our vocabulary without tripping over each others' definitions between applications.

2.3 Expanding our vocabulary

2.3.1 Setting the stage

The default ActivityStreams vocabulary is fairly comprehensive, and for most social networking applications, probably has everything you need.

But what happens if it doesn't? In this section we'll explore adding vocabulary for an imaginary social network application that's a "check-in" application (somewhere between Foursquare and Groupon in design). Users check in on their phones or other mobile devices when they arrive somewhere, and their frequency of attendance is somewhat gameified. Frequent attendance increases a "royalty" status, and some stores or other consumer outlets may reward users with one-time use coupons for frequent attendance.

This is an intereresting application, but the default vocabulary doesn't give us everything we need. We'd like to define new types like "CheckIn", but if someone comes up with a hospital application that also defines CheckIn, we don't want our applications to get confused while talking to each other. How can we do this?

Luckily, Activipy has you covered... read on!

2.3.2 Some new terms

Let's start out easy, and worry about the details later.

So let's say that we're a user of a CheckUp vocabulary using service. We have a nice little demo for this, so let's import that, and we'll start using the CheckUpEnv:

```
>>> from activipy.demos import checkup     # contains vocab and environment
>>> from activipy import vocab             # we'll use some of these too
>>> env = checkup.CheckUpEnv               # for convenience of tutorial
```

We'll also already assume that we've got a user setup in this system. Even though we're working with new vocabulary, the Core vocabulary is also set up in the CheckUpEnv:

```
>>> me = env.c.Person(
...   "http://social.example/u/sugartooth/",
...   displayName="Sarah Sugartooth")
```

We're also going to set up an imaginary connection to the CheckUp server we're using, just for demonstration purposes:

```
.. TODO: Provide an actual FakeConnection() thing eventually
```

```
>>> conn = FakeConnection()
```

So we know there's a CheckIn vocabulary available through CheckUpEnv... let's say we just arrived at the "Sweet Expressions" ice cream parlor. Time to check in!

```
>>> check_in = env.c.CheckIn(
...     actor=me,
...     location=env.c.Place(
...         "http://sweetexpressions.example/",
...          displayName="Sweet Expressions"))
>>> check_in
<ASObj CheckIn>
```

What does that check_in object look like in json form?

```
>>> check_in.json()
{'@context': 'http://checkup.example/context.jld',
 '@type': 'CheckIn',
 'actor': {'@id': 'http://social.example/u/sugartooth/',
           '@type': 'Person',
           'displayName': 'Sarah Sugartooth'},
 'location': {'@id': 'http://sweetexpressions.example/',
              '@type': 'Place',
              'displayName': 'Sweet Expressions'}}
```

Huh, so this is kind of interesting. The *@type* looks nice and simple as "CheckIn", but there's also this *@context* thing. We won't worry too much about what that is yet, but a brief preview is given by checking the types:

```
>>> check_in.types_astype
[<ASType CheckIn>]
>>> check_in.types_expanded
['http://checkup.example/ns#CheckIn']
```

Hm, so this is pretty cool! Something in that @context has helped clarify exactly what "CheckIn" we're talking about. We'll get into this more later, but if in the future there was ever a DoctorVisit vocabulary, we'd never mistake "http://checkup.example/ns#CheckIn" for a "http://doctoroffice.example/terms#CheckIn". Our doctor might be fairly confused if we sent her a note telling her that we're out to get ice cream, but now we can be sure that that mistake won't happen. We'll get into this more later, but that's already good to know!

Okay, so we've made the *check_in* object, but we haven't *done* anything with it yet. We have this connection to our CheckUp service, why not post it there!

```
>>> check_in.m.post(conn)
```

Well that was easy... what can we do now? How about check out our inbox?

```
>>> inbox_contents = me.m.inbox(conn)
<ASObj Collection>
>>> inbox_contents.json()
{'@context': 'http://checkup.example/context.jld',
 '@type': 'Collection',
 'items': [
     {'@type': ['Coupon', 'Note'],
      'content': 'Thanks for 40 visits to Sweet Expressions!',
      'recipient': {'@id': 'http://social.example/u/sugartooth/',
                    '@type': 'Person',
                    'displayName': 'Sarah Sugartooth'},
      'redeem_uri': 'http://sweetexpressions.example/coupon/9ae37630/',
      'vendor': {'@id': 'http://sweetexpressions.example/',
                 '@type': 'Place',
                 'displayName': 'Sweet Expressions'}}
     {'@type': 'http://checkup.example/ns#RoyalStatus',
      'displayName': "Sarah Sugartooth's been upgraded to Queen status!",
      'recipient': {'@id': 'http://social.example/u/sugartooth/',
                    '@type': 'Person',
                    'displayName': 'Sarah Sugartooth'},
      'status': 'Queen',
      'vendor': {'@id': 'http://sweetexpressions.example/',
                 '@type': 'Place',
                 'displayName': 'Sweet Expressions'}}]}
```

Huh... that's interesting, so this is a collection... it has two items in it. We could pull out that first item individually and take a look at it in detail:

```
>>> coupon = inbox_contents["items"][0]
>>> coupon
<ASObj Coupon, Note>
```

Huh, that's interesting, it says it's a Coupon *and* a Note? It sure is:

```
>>> coupon.json()
{'@context': 'http://checkup.example/context.jld',
 '@type': ['Coupon', 'Note'],
 'content': 'Thanks for visiting Sweet Expressions!',
 'recipient': {'@id': 'http://social.example/u/sugartooth/',
               '@type': 'Person',
               'displayName': 'Sarah Sugartooth'},
 'redeem_uri': 'http://sweetexpressions.example/coupon/9ae37630/',
 'vendor': {'@id': 'http://sweetexpressions.example/',
            '@type': 'Place',
            'displayName': 'Sweet Expressions'}}
>>> coupon.types_astype
[<ASType Coupon>, <ASType Note>]
>>> coupon.types_expanded
['http://checkup.example/ns#Coupon', 'http://www.w3.org/ns/activitystreams#Note']
```

So first of all, this makes sense. All a Coupon is to our system is something fairly functional, something by which a vendor can deliver a *redeem_uri* (which is a one-time-use URI to redeem a coupon for something special) to a recipient. But our friends at Sweet Expressions wanted to include a little thank-you along with the coupon, so this object acts as a composite type of both of these.

As a user, for the most part, the details of handling composite types are mostly taken care of for you by Activipy. But as a side note, it's interesting to look at what this means for the inheritence chain:

```
# inheritence chain for the Coupon ASType
>>> checkup.Coupon.inheritance_chain
[<ASType Coupon>, <ASType Object>]
# inheritence chain for the Coupon ASType
>>> vocab.Note.inheritance_chain
[<ASType Note>, <ASType Content>, <ASType Object>]
# The actual types for our coupon object
>>> coupon.types_astype
[<ASType Coupon>, <ASType Note>]
# And the inheritence chain built by the composite type
>>> coupon.types_inheritance
[<ASType Coupon>, <ASType Note>, <ASType Content>, <ASType Object>]
```

You don't necessarily need to make use of composite types in your system, but we can now see one possible reason you might want to: some of your activities might be a union of actions, and composite types are a great way to express them. (As will be covered in the "advanced tutorial", you can use the mapping/folding method features to maximize composite types with your method handling, too.)

There's another reason that we might see composite types used... let's look at our outbox from our connection:

```
>>> me.m.outbox(conn).json()
{'@context': 'http://checkup.example/context.jld',
 '@type': 'Collection',
 'items': [
     {'@type': ['CheckIn', 'Arrive'],
      'actor': {'@id': 'http://social.example/u/sugartooth/',
                '@type': 'Person',
                'displayName': 'Sarah Sugartooth'},
      'location': {'@id': 'http://sweetexpressions.example/',
                   '@type': 'Place',
                   'displayName': 'Sweet Expressions'}}]}
```

Huh... that's interesting. So we've retreived from the connection our recent activities, but whatever server is on the other end is showing the CheckIn we just did as a composite type of both *CheckIn* and *Arrive*! This seems kind of stange, because *CheckIn* is just a subclass of *Arrive* anyway:

```
>>> checkup.CheckIn.inheritance_chain
... [<ASType CheckIn>,
...  <ASType Arrive>,
...  <ASType IntransitiveActivity>,
...  <ASType Activity>,
...  <ASType Object>]
```

So if a *CheckIn* is technically part of an *Arrive*, why might our server do this? Well, not all servers might have the *CheckIn* vocabulary (or know that it's a parent to *Arrive*), so since *CheckIn* can technically be displayed as an *Arrive*, so if a recipient of this activity knows how to display an *Arrive* but not a *Checkin*, they could do that.

So anyway! Now you've seen how to work with new types. Now you're maybe starting to wonder, what's the theoretical model of all this under the hood? (What, you weren't wondering that before? Okay, now you are!) Let's take a look!

2.3.3 Expanding into json-ld

Remember when we did this?

```
>>> root_beer_note.types_expanded
['http://www.w3.org/ns/activitystreams#Note']
>>> coupon.types_expanded
['http://checkup.example/ns#Coupon', 'http://www.w3.org/ns/activitystreams#Note']
```

This starts to make more sense when we think about naming conflicts... if you send me a message about "running a mile", and I send you a message about "running a program", those are obviously two very different definitions of "running", and it might create a lot of problems if they become confused. There should be an unambiguous way to represent things, and that's exactly where json-ld comes in. In json-ld, json objects can be "expanded" to an unambiguous format, and then "compacted" to the right definitions for our own local server, so we'll never get confused between two different definitions of "running" again. Here's a brief hint towards that right now:

```
>>> post_this.expanded()
[{'@id': 'http://tsyesika.co.uk/act/foo-id-here/',
  '@type': ['http://www.w3.org/ns/activitystreams#Create'],
  'http://www.w3.org/ns/activitystreams#actor': [
      {'@id': 'http://tsyesika.co.uk/',
       '@type': ['http://www.w3.org/ns/activitystreams#Person'],
       'http://www.w3.org/ns/activitystreams#displayName': [
           {'@value': 'Jessica Tallon'}]}],
  'http://www.w3.org/ns/activitystreams#object': [
      {'@id': 'http://tsyesika.co.uk/chat/sup-yo/',
       '@type': ['http://www.w3.org/ns/activitystreams#Note'],
       'http://www.w3.org/ns/activitystreams#content': [
           {'@value': 'Up for some root beer floats?'}]}],
  'http://www.w3.org/ns/activitystreams#to': [
      {'@id': 'acct:cwebber@identi.ca'},
      {'@id': 'acct:justaguy@rhiaro.co.uk'},
      {'@id': 'acct:ladyaeva@hedgehog.example'}]}]
```

That might look a bit complicated, but normally you wouldn't work in an expanded document, you'd compact to your local context. If this seems confusing, you don't really need to worry about; Activipy uses json-ld under the hood but you usually won't need to interact with it. One nice feature though is that ActivityStreams 2.0 documents have an "implied context" of the core ActivityStreams vocabulary. This means that a "Note" will always mean the ActivityStreams version of a Note, even if you don't do any fancy context things and are using just plain old json. Even when you get into extension land, Activipy makes things so that you can think as in terms of pythonic constructors rather than json-ld, so your code will look like simple Python, just like at the very beginning of our tutorial.

So if you nearly never need to work with this super-extensible version of things, what's the point of us showing you it? Well the interesting here is, since that's the unambiguous "expanded" version, we can now understand how we can get information from another source and clearly understand its meaning.

We can also now begin to understand what "compacted" means: the simple JSON representations we've been showing for most of this post!

So we can take data from the outside world, expand it into an unambiguous format, and then compact it down to the terminology we actually know. Once compacted, it's in a format that's so simple even poor Billy Scripter with his json and ruby toolbox can use it. And since we've compacted it to a context *we* know, we know that an *Activity* is an Activity and we'll never confuse "run a program" with someone else's "run a mile" again. Horray!

But... the *CheckIn* / *Coupon* examples above included this *@context* key. What is that thing? Let's take a look.

2.3.4 It's all contextual

As a refresher, some of our early examples looked like this:

```
>>> root_beer_note
{'@id': 'http://tsyesika.co.uk/chat/sup-yo/',
 '@type': 'Note',
 'content': 'Up for some root beer floats?'}
>>> root_beer_note.types_expanded
['http://www.w3.org/ns/activitystreams#Note']
```

No *@context* there. And yet just recently we saw this:

```
>>> check_in.json()
{'@context': 'http://checkup.example/context.jld',
 '@type': 'CheckIn',
 'actor': {'@id': 'http://social.example/u/sugartooth/',
           '@type': 'Person',
           'displayName': 'Sarah Sugartooth'},
 'location': {'@id': 'http://sweetexpressions.example/',
              '@type': 'Place',
              'displayName': 'Sweet Expressions'}}
>>> check_in.types_expanded
['http://checkup.example/ns#CheckIn']
```

Okay, that does have an *@context*! Well, we're working with extensions, so it's obvious that this maps our vocabulary to the unambiguous definitions we saw in the expanded version.

So that's kind of cool, we can imagine that *'http://checkup.example/context.jld'* somehow maps *'CheckIn'* -> *'http://checkup.example/ns#CheckIn'* in our second example. But wait, how did *'Note'* get mapped to *'http://www.w3.org/ns/activitystreams#Note'* in our first example? We didn't specify any context at all! This is because ActivityStreams has an "implied context" of its own vocabulary at *'http://www.w3.org/TR/activitystreams-core/activitystreams2-context.jsonld'*, so that vocabulary mapping context is there without us even having to specify it.

You might have noticed a small amount of trickery: *http://checkup.example/context.jld* doesn't exist! You caught us, this is just a demonstration, so we overrode the *default_loader* in the environment to pretend that it knew what was at that URL already. Tricky!

It turns out we could have rewritten the *CheckIn* example with the contents of what we were "pretending" was at *http://checkup.example/context.jld*, and it would have worked during expansion just as well:

```
{'@context': {'CheckIn': {'@id': 'http://checkup.example/ns#CheckIn',
                          '@type': '@id'},
              'Coupon': {'@id': 'http://checkup.example/ns#Coupon',
                         '@type': '@id'},
              'RoyalStatus': {'@id': 'http://checkup.example/ns#RoyalStatus',
                              '@type': '@id'}},
 '@type': 'CheckIn',
 'actor': {'@id': 'http://social.example/u/sugartooth/',
           '@type': 'Person',
           'displayName': 'Sarah Sugartooth'},
 'location': {'@id': 'http://sweetexpressions.example/',
              '@type': 'Place',
              'displayName': 'Sweet Expressions'}}
```

Wow, that's *quite* verbose. But that's basically dumping what would have been at *http://checkup.example/context.jld* inline.

It turns out there's another kind of interesting way to specify terms in an *@context*, which is namespacing:

```
{'@context': {'CheckUp': 'http://checkup.example/ns#'},
 '@type': 'CheckUp:CheckIn',
 'actor': {'@id': 'http://social.example/u/sugartooth/',
```

```
           '@type': 'Person',
           'displayName': 'Sarah Sugartooth'},
 'location': {'@id': 'http://sweetexpressions.example/',
              '@type': 'Place',
              'displayName': 'Sweet Expressions'}}
```

If you'd like to play with these, we have environments set up for them at *activipy.demos.checkup* under the varibles *CheckUpVerboseEnv* and *CheckUpNSEnv*.

2.3.5 Consume the world

So now you have a solid theoretical understanding of how information can be unambiguously represented in Activipy. But what if we want to consume activities from the outside world?

TODO: Finish this section!

2.4 The more we change, the more we stay the same

TODO: Fill in this section on the immutable properties of Activipy

CHAPTER 3

Indices and tables

- genindex
- modindex
- search