PyOpenWorm Documentation

Release alpha0.5

PyOpenWorm

June 19, 2015

Contents

Our main README is available online on Github. [1] This documentation contains additional materials beyond what is covered there.

Contents:

[1] http://github.com/openworm/PyOpenWorm

CHAPTER 1

PyOpenWorm API

1.1 Basic Classes

1.1.1 Worm

class PyOpenWorm.**Worm**(*scientific_name=False*, ***kwargs*)

Bases: PyOpenWorm.dataObject.DataObject

A representation of the whole worm.

All worms with the same name are considered to be the same object.

Attributes

neuron_network	(ObjectProperty) The neuron network of the worm
muscle	(ObjectProperty) Muscles of the worm

add_reference(*g*, *reference_iri*)

Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

get_neuron_network()

Return the neuron network of the worm.

Example:

```
# Grabs the representation of the neuronal network
>>> net = P.Worm().get_neuron_network()

# Grab a specific neuron
>>> aval = net.aneuron('AVAL')

>>> aval.type()
set([u'interneuron'])

#show how many connections go out of AVAL
>>> aval.connection.count('pre')
77
```

Returns An object to work with the network of the worm

Return type *PyOpenWorm.Network*

get_semantic_net()

Get the underlying semantic network as an RDFLib Graph

Returns A semantic network containing information about the worm

Return type rdflib.ConjunctiveGraph

load()

Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

muscles()

Get all Muscle objects attached to the Worm

Returns a set of all muscles:

Example:

```
>>> muscles = P.Worm().muscles()
>>> len(muscles)
96
```

Returns A set of all muscles

Return type set

retract()

Remove this object from the data store.

save()

Write in-memory data to the database. Derived classes should call this to update the store.

1.1.2 Network

class PyOpenWorm.**Network**(***kwargs*)

Bases: PyOpenWorm.dataObject.DataObject

A network of neurons

Attributes

neuron	Representation of neurons in the network
synapse	Representation of synapses in the network

add_reference(*g*, *reference_iri*)

Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

aneuron(*name*)

Get a neuron by name.

Example:

```
    # Grabs the representation of the neuronal network
    >>> net = P.Worm().get_neuron_network()

    # Grab a specific neuron
    >>> aval = net.aneuron('AVAL')

    >>> aval.type()
    set([u'interneuron'])
```

Parameters **name** – Name of a c. elegans neuron

Returns Neuron corresponding to the name given

Return type *PyOpenWorm.Neuron*

as_networkx()

interneurons()
Get all interneurons

Returns A iterable of all interneurons

Return type iter(Neuron)

load()
Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

motor()
Get all motor

Returns A iterable of all motor neurons

Return type iter(Neuron)

neurons()
Gets the complete set of neurons in this network.

Example:

```
    # Grabs the representation of the neuronal network
    >>> net = P.Worm().get_neuron_network()

    #NOTE: This is a VERY slow operation right now
    >>> len(set(net.neurons()))
    302
    >>> set(net.neurons())
    set(['VB4', 'PDEL', 'HSNL', 'SIBDR', ... 'RIAL', 'MCR', 'LUAL'])
```

retract()
Remove this object from the data store.

save()
Write in-memory data to the database. Derived classes should call this to update the store.

sensory()
Get all sensory neurons

Returns A iterable of all sensory neurons

Return type iter(Neuron)

1.1.3 Connection

class PyOpenWorm.**Connection**(*pre_cell=None*, *post_cell=None*, *number=None*, *syntype=None*, *synclass=None*, ***kwargs*)

Bases: PyOpenWorm.relationship.Relationship

Connection between neurons

Parameters **pre_cell** : string or Neuron, optional

The pre-synaptic cell

post_cell : string or Neuron, optional

The post-synaptic cell

number : int, optional

The weight of the connection

syntype : {'gapJunction', 'send'}, optional

The kind of synaptic connection. 'gapJunction' indicates a gap junction and 'send' a chemical synapse

synclass : string, optional

The kind of Neurotransmitter (if any) sent between *pre_cell* and *post_cell*

add_reference(*g*, *reference_iri*)

Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

load()

Load in data from the database. Derived classes should override this for their own data structures.

load() returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

retract()

Remove this object from the data store.

save()

Write in-memory data to the database. Derived classes should call this to update the store.

1.1.4 Cell

class PyOpenWorm.**Cell**(*name=False*, *lineageName=False*, ***kwargs*)

Bases: PyOpenWorm.dataObject.DataObject

A biological cell.

All cells with the same name are considered to be the same object.

Parameters **name** : string

The name of the cell

lineageName : string

The lineageName of the cell Example:

```
>>> c = Cell(name="ADAL")
>>> c.lineageName() # Returns ["AB plapaaaapp"]
```

Attributes

name	(DatatypeProperty) The 'adult' name of the cell typically used by biologists when discussing C. elegans
lineage-Name	(DatatypeProperty) The lineageName of the cell
descrip-tion	(DatatypeProperty) A description of the cell
division-Volume	(DatatypeProperty) When called with no argument, return the volume of the cell at division during development. When called with an argument, set the volume of the cell at division Example:: >>> v = Quantity("600","(um)^3") >>> c = Cell(lineageName="AB plapaaaap") >>> c.divisionVolume(v)

add_reference(*g*, *reference_iri*)

Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

blast()

Return the blast name.

Example:

```
>>> c = Cell(name="ADAL")
>>> c.blast() # Returns "AB"
```

Note that this isn't a Property. It returns the blast extracted from the ''first" lineageName saved.

daughterOf()

Return the parent(s) of the cell in terms of developmental lineage.

Example:

```
>>> c = Cell(lineageName="AB plapaaaap")
>>> c.daughterOf() # Returns [Cell(lineageName="AB plapaaaa")]
```

load()

Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

parentOf()

Return the direct daughters of the cell in terms of developmental lineage.

Example:

```
        >>> c = Cell(lineageName="AB plapaaaap")
        >>> c.parentOf() # Returns [Cell(lineageName="AB plapaaaapp"),Cell(lineageName="AB plapaaaap
```

retract()
Remove this object from the data store.

save()
Write in-memory data to the database. Derived classes should call this to update the store.

1.1.5 Neuron

class PyOpenWorm.**Neuron**(*name=False*, ***kwargs*)
Bases: PyOpenWorm.cell.Cell

A neuron.

See what neurons express some neuropeptide

Example:

```
    # Grabs the representation of the neuronal network
    >>> net = P.Worm().get_neuron_network()

    # Grab a specific neuron
    >>> aval = net.aneuron('AVAL')

    >>> aval.type()
    set([u'interneuron'])

    #show how many connections go out of AVAL
    >>> aval.connection.count('pre')
    77

    >>> aval.name()
    u'AVAL'

    #list all known receptors
    >>> sorted(aval.receptors())
    [u'GGR-3', u'GLR-1', u'GLR-2', u'GLR-4', u'GLR-5', u'NMR-1', u'NMR-2', u'UNC-8']

    #show how many chemical synapses go in and out of AVAL
    >>> aval.Syn_degree()
    90
```

Parameters **name** : string

The name of the neuron.

Attributes

type	(DatatypeProperty) The neuron type (i.e., sensory, interneuron, motor)
receptor	(DatatypeProperty) The receptor types associated with this neuron
innexin	(DatatypeProperty) Innexin types associated with this neuron
neurotransmitter	(DatatypeProperty) Neurotransmitters associated with this neuron
neuropeptide	(DatatypeProperty) Name of the gene corresponding to the neuropeptide produced by this neuron
neighbor	(Property) Get neurons connected to this neuron if called with no arguments, or with arguments, state that neuronName is a neighbor of this Neuron
connection	(Property) Get a set of Connection objects describing chemical synapses or gap junctions between this neuron and others

GJ_degree()
Get the degree of this neuron for gap junction edges only

Returns total number of incoming and outgoing gap junctions

Return type int

Syn_degree()
Get the degree of a this neuron for chemical synapse edges only

Returns total number of incoming and outgoing chemical synapses

Return type int

add_reference(*g*, *reference_iri*)
Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

blast()
Return the blast name.

Example:

```
>>> c = Cell(name="ADAL")
>>> c.blast() # Returns "AB"
```

Note that this isn't a Property. It returns the blast extracted from the ''first" lineageName saved.

daughterOf()
Return the parent(s) of the cell in terms of developmental lineage.

Example:

```
>>> c = Cell(lineageName="AB plapaaaap")
>>> c.daughterOf() # Returns [Cell(lineageName="AB plapaaaa")]
```

get_incidents(*type=0*)
Get neurons which synapse at this neuron

load()
Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

parentOf()
Return the direct daughters of the cell in terms of developmental lineage.

Example:

```
>>> c = Cell(lineageName="AB plapaaaap")
>>> c.parentOf() # Returns [Cell(lineageName="AB plapaaaapp"),Cell(lineageName="AB plapaaaap
```

retract()
Remove this object from the data store.

save()
Write in-memory data to the database. Derived classes should call this to update the store.

1.1.6 Muscle

class PyOpenWorm.**Muscle**(*name=False*, ***kwargs*)
Bases: PyOpenWorm.cell.Cell

A single muscle cell.

See what neurons innervate a muscle:

Example:

```
>>> mdr21 = P.Muscle('MDR21')
>>> innervates_mdr21 = mdr21.innervatedBy()
>>> len(innervates_mdr21)
4
```

Attributes

neurons	(ObjectProperty) Neurons synapsing with this muscle
receptors	(DatatypeProperty) Get a list of receptors for this muscle if called with no arguments, or state that this muscle has the given receptor type if called with an argument

add_reference(*g*, *reference_iri*)
Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

blast()
Return the blast name.

Example:

```
>>> c = Cell(name="ADAL")
>>> c.blast() # Returns "AB"
```

Note that this isn't a Property. It returns the blast extracted from the ''first" lineageName saved.

daughterOf()
Return the parent(s) of the cell in terms of developmental lineage.

Example:

```
>>> c = Cell(lineageName="AB plapaaaap")
>>> c.daughterOf() # Returns [Cell(lineageName="AB plapaaaa")]
```

load()
Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

parentOf()
Return the direct daughters of the cell in terms of developmental lineage.

Example:

```
>>> c = Cell(lineageName="AB plapaaaap")
>>> c.parentOf() # Returns [Cell(lineageName="AB plapaaaapp"),Cell(lineageName="AB plapaaaap
```

retract()
Remove this object from the data store.

save()
Write in-memory data to the database. Derived classes should call this to update the store.

1.1.7 Channel

class PyOpenWorm.**Channel**(*subfamily=False*, ***kwargs*)
Bases: PyOpenWorm.dataObject.DataObject

An ion channel.

Channels are identified by subtype name.

Parameters **subfamily** : string

The subfamily to which the ion channel belongs

Attributes

subfamily	(DatatypeProperty) The subfamily to which the ion channel belongs
Models	(Property) Get experimental models of this ion channel

add_reference(*g*, *reference_iri*)
Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

load()
Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

retract()
Remove this object from the data store.

save()
Write in-memory data to the database. Derived classes should call this to update the store.

1.1.8 Evidence

class PyOpenWorm.**Evidence**(*conf=False*, ***source*)
Bases: PyOpenWorm.dataObject.DataObject

A representation of some document which provides evidence like scholarly references, for other objects.

Possible keys include:

```
pmid,pubmed: a pubmed id or url (e.g., 24098140)
wbid,wormbase: a wormbase id or url (e.g., WBPaper00044287)
doi: a Digitial Object id or url (e.g., s00454-010-9273-0)
```

Parameters **doi** : string

A Digital Object Identifier (DOI) that provides evidence, optional

pmid : string

A PubMed ID (PMID) that point to a paper that provides evidence, optional

wormbaseid : string

An ID from WormBase that points to a record that provides evidence, optional

author : string

The author of the evidence

title : string

The title of the evidence

year : string or int

The date (e.g., publication date) of the evidence

uri : string

A URL that points to evidence

Attributes

asserts	(ObjectProperty (value_type=DataObject)) When used with an argument, state that this Evidence asserts that the relationship is true. Example:: import bibtex bt = bibtex.parse("my.bib") n1 = Neuron("AVAL") n2 = Neuron("DA3") c = Connection(pre=n1,post=n2,class="synapse") e = Evidence(bibtex=bt['white86']) e.asserts(c) Other methods return objects which asserts accepts. Example:: n1 = Neuron("AVAL") r = n1.neighbor("DA3") e = Evidence(bibtex=bt['white86']) e.asserts(r) When used without arguments, returns a sequence of statements asserted by this evidence Example:: import bibtex bt = bibtex.parse("my.bib") n1 = Neuron("AVAL") n2 = Neuron("DA3") c = Connection(pre=n1,post=n2,class="synapse") e = Evidence(bibtex=bt['white86']) e.asserts(c) list(e.asserts()) # Returns a list [..., d, ...] such that d==c
doi	(DatatypeProperty) A Digital Object Identifier (DOI) that provides evidence, optional
pmid	(DatatypeProperty) A PubMed ID (PMID) that point to a paper that provides evidence, optional
wormbaseid	(DatatypeProperty) An ID from WormBase that points to a record that provides evidence, optional
author	(DatatypeProperty) The author of the evidence
title	(DatatypeProperty) The title of the evidence
year	(DatatypeProperty) The date (e.g., publication date) of the evidence
uri	(DatatypeProperty) A URL that points to evidence

add_data(*k*, *v*)
Add a field

Parameters **k** : string

Field name

v : string

Field value

add_reference(*g*, *reference_iri*)
Add a citation to a set of statements in the database

Parameters **triples** – A set of triples to annotate

load()
Load in data from the database. Derived classes should override this for their own data structures.

`load()` returns an iterable object which yields DataObjects which have the same class as the object and have, for the Properties set, the same values

Parameters **self** – An object which limits the set of objects which can be returned. Should have the configuration necessary to do the query

retract()
Remove this object from the data store.

save()
Write in-memory data to the database. Derived classes should call this to update the store.

CHAPTER 2

For Users

2.1 Requirements for data storage in OpenWorm

Our OpenWorm database captures facts about C. *elegans*. The database stores data for generating model files and together with annotations describing the origins of the data. Below are a set of recommendations for implementation of the database organized around an RDF model.

2.1.1 Interface

Access is through a Python library which communicates with the database. This library serves the function of providing an object oriented view on the database that can be accessed through the Python scripts commonly used in the project. The *draft api* is described separately.

2.1.2 Data modelling

Biophysical and anatomical data are included in the database. A sketch of some features of the data model is below. Also included in our model are the relationships between these types. Given our choice of data types, we do not model the individual interactions between cells as entities in the database. Rather these are described by generic predicates in an RDF triple. For instance, neuron A synapsing with muscle cell B would give a statement (A, synapsesWith, B), but A synapsing with neuron C would also have (A, synapsesWith, C). Data which belong to the specific relationship between two nodes is attached to an rdf:Statement object which points to the statement. This choice is intended to easy querying and extension later on.

Nervous system

In the worm's nervous system, we capture a few important data types (listed *below*). These correspond primarily to the anatomical structures and chemicals which are necessary for the worm to record external and internal stimuli and activate its body in response to those stimuli.

Data types

A non-exhaustive list of neurological data types in our C. elegans database:

- receptor types identified in the nerve cell
- neurons
- ion channels

- neurotransmitters
- muscle receptors

Development

Caenorhabditis elegans has very stable cell division patterns in the absence of mutations. This means that we can capture divisions in our database as static 'daughter_of' relationships. The theory of differentiation codes additionally gives an algorithmic description to the growth patterns of the worm which describes signals transmitted between developing cells. In order to test this theory we would like to leverage existing photographic data indicating the volume of cells at the time of their division as this relates to the differentiation code stored by the cell. Progress on this issue is documented on Github.

Aging

Concurrently with development, we would like to begin modeling the effects of aging on the worm. Aging typically manifests in physiological changes due to transcription errors or cell death. These physiological changes can be represented abstractly as parameters to the function of biological entities. See Github for further discussion.

2.1.3 Information assurance

Reasoning and Data integrity

To make full use of RDF storage it's recommended to leverage reasoning over our stored data. Encoding rules for the worm requires a good knowledge of both C. elegans and the database schema. More research needs to be done on this going forward. Preliminarily, SPIN, a constraint notation system based on SPARQL looks like a good candidate for *specifying* rules, but an inference engine for *enforcing* the rules still needs to be found.

Input validation

Input validation is to be handled through the interface library referenced *above*. In general, incorrect entry of biological names will result in an error being reported identifying the offending entry and providing a acceptable entries where appropriate. No direct access to the underlying data store will be provided.

Provenance

Tracking the origins of facts stated in the database demands a method of annotating statements in our database. Providing citations for facts must be as simple as providing a global identifier (e.g., URI, DOI) or a local identifier (e.g., Bibtex identifier, Pubmed ID). A technique called RDF reification allows us to annotate arbitrary facts in our database with additional information. This technique allows for the addition of structured citation data to facts in the database as well as annotations for tracking responsibility for uploads to the database. Further details for the attachment of evidence using this technique are given in the *draft api*.

In line with current practices for communication through the source code management platform, Github, we would like to track responsibility for new uploads to the database. Two methods are proposed for tracking this information: RDF named graphs and RDF reification. Tracking information must include, at least, a time-stamp on the update and linking of the submitted data to the uploader's unique identifier (e.g., email address). Named graphs have the advantage of wide support for the use of tracking uploads. The choice between these depends largely the support of the chosen data store for named graphs.

Access control

Write access to data in the project has been inconsistent between various data sources in the project. Going forward, write access to OpenWorm databases should be restricted to authenticated users to forestall the possibility of malicious tampering.

One way to accomplish this would be to leverage GitHub's fork and pull model with the data as well as the code. This would require two things:

- Instead of remote hosting of data, data is local to each copy of the library

within a local database - A serialization method dumps a new copy of the data out to a flat file enabling all users of the library to contribute their modifications to the data back to the PyOpenWorm project via GitHub.

A follow on to #2 is that the serialization method would need to preserve the ordering of data elements and write in some plain text format so that a simple diff on GitHub would be able to illuminate changes that were made.

2.1.4 Miscellaneous

Versioning

Experimental methods are constantly improving in biological research. These improvements may require updating the data we reference or store internally. However, in making updates we must not immediately expunge older content, breaking links created by internal and external agents. Ideally we would have a means of deprecating old data and specifying replacements. On the level of single resources, this is a trivial mapping which may be done transparently to all readers. For a more significant change, altering the schema, human intervention may be required to update external readers.

2.1.5 Why RDF?

RDF offers advantages in resilience to schema additions and increased flexibility in integrating data from disparate sources. [1] These qualities can be valued by comparison to relational database systems. Typically, schema changes in a relational database require extensive work for applications using it. [2] In the author's experience, RDF databases offer more freedom in restructuring. Also, for data integration, SPARQL, the standard language for querying over RDF has Federated queries which allow for nearly painless integration of external SPARQL endpoints with existing queries.

FuXi

FuXi is implemented as a semantic reasoning layer in PyOpenWorm. In other words, it will be used to automatically infer (and set) properties from other properties in the worm database. This means that redundant information (ex: explicitly stating that each object is of class "dataType") and subclass relationships (ex: that every object of type "Neuron" is also of type "Cell"), as well as other relationships, can be generated by the firing of FuXi's rule engine, without being hand-coded.

Aside from the time it saves in coding, FuXi may allow for a smaller footprint in the cloud, as many relationships within the database could be inferred *after* download.

The advantage of local storage of the database that goes along with each copy of the library is that the data will have the version number of the library. This means that data can be 'deprecated' along with a deprecated version of the library. This also will prevent changes made to a volatile database that break downstream code that uses the library.

[1] http://answers.semanticweb.com/questions/19183/advantages-of-rdf-over-relational-databases
[2] http://research.microsoft.com/pubs/118211/andy%20maule%20-%20thesis.pdf

2.2 Adding Data to *YOUR* OpenWorm Database

So, you've got some biological data about the worm and you'd like to save it in PyOpenWorm, but you don't know how it's done?

You've come to the right place!

A few biological entities (e.g., Cell, Neuron, Muscle, Worm) are pre-coded into PyOpenWorm. The full list is available in the *API*. If these entities already cover your use-case, then all you need to do is add values for the appropriate fields and save them. If you have data already loaded into your database, then you can load objects from it:

```
n = Neuron()
n.receptor('UNC-13')
for x in n.load():
    do_something_with_unc13_neuron(n)
```

If you need additional entities it's easy to create them. Documentation for this is provided *here*.

Typically, you'll want to attach the data that you insert to entities already in the database. This allows you to recover objects in a hierarchical fashion from the database later. `Worm`, for instance has a property, `neuron_network`, which points to the `Network` which should contain all neural cells and synaptic connections. To initialize the hiearchy you would do something like:

```
w = Worm('C. briggsae') # The name is optional and currently defaults to 'C. elegans'
nn = Network()          # make a neuron network
w.neuron_network(nn)    # attach to the worm the neuron network
n = Neuron()            # make an unnamed neuron
n.receptor('UNC-13')    # state that the neuron has a UNC-13 type receptor
nn.neuron(n)            # attach to the neuron network
w.save()                # save all of the data attached to the worm
```

It is possible to create objects without attaching them to anything and they can still be referenced by calling load on an instance of the object's class as in `n.load()` above. This also points out another fact: you don't have to set up the hierarchy for each insert in order for the objects to be linked to existing entities. If you have previously set up connections to an entity (e.g., `Worm('C. briggsae')`), assuming you *only* have one such entity, you can refer to things attached to it without respecifying the hierarchy for each script. The database packaged with PyOpenWorm should have only one Worm and one Network.

Remember that once you've set up all of the data, you must save the objects. For now, this requires keeping track of top-level objects – objects which aren't values of some other property – and calling `save()` on each of them individually. This isn't too difficult to achieve.

Future capabilities:

- Adding propositional logic to support making statements about all entities

matching some conditions without needing to `load()` and `save()` them from the database. * Statements like:

```
  w = Worm()
  w.neuron_network.neuron.receptor('UNC-13')
  l = list(w.load()) # Get a list of worms with neurons expressing 'UNC-13'

currently, to do the equivalent, you must work backwards, finding all neurons
with UNC-13 receptors, then getting all networks with those neurons, then
getting all worms with those networks::

  worms = set()
  n = Neuron()
  n.receptor('UNC-13')
  for ns in n.load():
```

```
      nn = Network()
      nn.neuron(ns)
      for z in nn.load():
          w = Worm()
          w.neuron_network(z)
          worms.add(w)
  l = list(worms)

It's not difficult logic, but it's 8 extra lines of code for a conceptually
very simple query.
```

- Also, queries like:

```
    l = list(Worm('C. briggsae').neuron_network.neuron.receptor()) # get a list
    #of all receptors expressed in neurons of C. briggsae
```

Again, not difficult to write out, but in this case it actually gives a much longer query time because additional values are queried in a `load()` call that are never returned.

We'd also like operators for composing many such strings so:

```
    Worm('C. briggsae').neuron_network.neuron.get('receptor', 'innexin') # list
    #of (receptor, innexin) values for each neuron
```

would be possible with one query and thus not requiring parsing and iterating over neurons twice–it's all done in a single, simple query.

2.3 Making data objects

To make new objects like `Neuron` or `Worm`, for the most part, you just need to make a Python class. Say, for example, that I want to record some information about drug reactions in C. elegans. I make `Drug` and `Experiment` classes to describe C. elegans reactions:

```
from PyOpenWorm import (DataObject,
                        DatatypeProperty,
                        ObjectProperty,
                        Worm,
                        Evidence,
                        connect)

class Drug(DataObject):
    # We set up properties in __init__
    def __init__(self,drug_name=False,*args,**kwargs):
        # pass arguments to DataObject
        DataObject.__init__(self,*args,**kwargs)
        Drug.DatatypeProperty('name', owner=self)
        if drug_name:
            self.name(drug_name)

class Experiment(DataObject):
    def __init__(self,*args,**kwargs):
        # pass arguments to DataObject
        DataObject.__init__(self,*args,**kwargs)
        Experiment.ObjectProperty('drug', value_type=Drug, owner=self)
        Experiment.ObjectProperty('subject', value_type=Worm, owner=self)
        Experiment.DatatypeProperty('route_of_entry', owner=self)
        Experiment.DatatypeProperty('reaction', owner=self)
```

```
connect()
# Set up with the RDF translation machinery
Experiment.register()
Drug.register()
```

I can then make a Drug object for moon rocks and describe an experiment by Aperture Labs:

```
d = Drug('moon rocks')
e = Experiment()
w = Worm("C. elegans")
ev = Evidence(author="Aperture Labs")
e.subject(w)
e.drug(d)
e.route_of_entry('ingestion')
e.reaction('no reaction')
ev.asserts(e)
```

and save it:

```
ev.save()
```

For simple objects, this is all we have to do.

You can also add properties to an object after it has been created by calling either ObjectProperty or DatatypeProperty on the object as is done in `__init__`:

```
d = Drug('moon rocks')
Drug.DatatypeProperty('granularity', owner=self)
d.granularity('ground up')
```

Properties added in this fashion will not propagate to any other objects, but they will be saved along with the object they are attached to.

2.4 Sharing Data with other users

Sharing is key to PyOpenWorm. This document covers the appropriate way to share changes with other PyOpenWorm users.

The shared PyOpenWorm database is stored in a Git repository distinct from the PyOpenWorm source code. Currently the database is stored in a Github repository here .

When you create a database normally, it will be stored in a format which is opaque to humans. In order to share your database you have two options: You can share the scripts which are used to create your database or you can share a human-readable serialization of the database. The second option is better since it doesn't require re-running your script to use the generated data, but it is best to share both.

For sharing the serialization, you should first clone the repository linked above, read the current serialization into your database (see *below* for an example of how you would do this), and then write out the serialization:

```
import PyOpenWorm as P
P.connect('path/to/your/config/file')
P.config()['rdf.graph'].serialize('out.n3', format='n3')
P.disconnect()
```

Commit, your changes to the git repository, push to a fork of the repository on Github and submit a pull request on the main repository. If for some reason your are unwilling or unable to create a Github account, post to the OpenWorm-discuss mailing list with a patch on the main repository with your changes and someone will have a look, possibly ask for adjustments or justification for your addition, and ultimately merge the changes for you.

To read the database back in you would do something like:

```
import PyOpenWorm as P
P.connect('path/to/your/config/file')
P.config()['rdf.graph'].parse('out.n3', format='n3')
P.disconnect()
```

Scripts are also added to the repository on Github to the `scripts` subdirectory.

CHAPTER 3

For Developers

3.1 Adding documentation

Documentation for PyOpenWorm is housed in two locations:

1. In the top-level project directory as `INSTALL.md` and `README.md`.
2. As a Sphinx project under the `docs` directory

To add a page about useful facts concerning C. elegans to the documentation, include an entry in the list under `toctree` in `docs/index.rst` like:

```
worm-facts
```

and create the file `worm-facts.rst` under the `docs` directory and add a line:

```
.. _worm-facts:
```

to the top of your file, remembering to leave an empty line before adding all of your wonderful worm facts.

You can get a preview of what your documentation will look like when it is published by running `sphinx-build` on the docs directory:

```
sphinx-build -w sphinx-errors docs build_destination
```

The docs will be compiled to html which you can view by pointing your web browser at `build_destination/index.html`. If you want to view the documentation locally with the ReadTheDocs theme you'll need to download and install it.

3.1.1 API Documentation

API documentation is generated by the Sphinx autodoc extension. The format should be easy to pick up on, but a reference is available here. Just add a docstring to your function/class/method and add an `automodule` line to `PyOpenWorm/__init__.py` and your class should appear among the other documented classes.

3.1.2 Substitutions

Project-wide substitutions can be (conservatively!) added to allow for easily changing a value over all of the documentation. Currently defined substitutions can be found in `conf.py` in the `rst_epilog` setting. More about substitutions

3.1.3 Conventions

If you'd like to add a convention, list it here and start using it.

Currently there are no real conventions to follow for documentation style, but additions to the docs will be subject to style and content review by project maintainers.

3.2 RDF semantics for PyOpenWorm

In the context of PyOpenWorm, biological objects are classes of, for instance, anatomical features of a worm. That is to say, statements made about C. elegans are not about a specific worm, but are stated about the entire class of worms. The semantics of a `property SimpleProperty/value value` triple are that if any value is set, then without any additional statements being made, an instance of the object has been observed to have the value at some point in time, somewhere, under some set of conditions. In other words, the statement is an existential quantification over the associated object(class).

The purpose of the identifiers for Properties is to allow statements to be made about them directly. An example:

```
<http://openworm.org/entities/Entity/1> <http://openworm.org/entities/Entity/interactsWith> <http://o
<http://openworm.org/entities/Entity_interactsWith/2> <http://openworm.org/entities/SimpleProperty/va

<http://openworm.org/entities/Entity/4> <http://openworm.org/entities/Entity/modulates> <http://openw
<http://openworm.org/entities/Entity_modulates/5> <http://openworm.org/entities/SimpleProperty/value>
```

3.3 RDF structure for PyOpenWorm

For most use cases, it is (hopefully) not necessary to write custom queries over the RDF graph in order to work with PyOpenWorm. However, if it does become necessary, it will be helpful to have an understanding of the structure of the RDF graph. Thus, a summary is given below.

For all `DataObjects` which are not `Properties`, there is an identifier of the form

```
<http://openworm.org/entities/Object_type/md5sum>
```

stored in the graph. This identifier will be associated with type data:

```
<http://openworm.org/entities/Object_type/md5sum> rdf:type <http://openworm.org/entities/Object_type>
<http://openworm.org/entities/Object_type/md5sum> rdf:type <http://openworm.org/entities/parent_of_Ob
<http://openworm.org/entities/Object_type/md5sum> rdf:type <http://openworm.org/entities/parent_of_pa
...
```

Properties have a slightly different form. They also have an identifier, which for `SimpleProperties` will look like this:

```
<http://openworm.org/entities/OwnerType_propertyName/md5sum>
```

`OwnerType` is the type of the Property's owner and `propertyName` is the name by which the property is accessed from an object of the owner's type. Other Properties will not necessarily have this form, but all of the standard Properties are implemented in terms of SimpleProperties and have no direct representation in the graph. For other Properties it is necessary to refer to their documentation or to examine the triples released by the Property of interest.

A DataObject's identifier is connected to a property in a triple like:

```
<http://openworm.org/entities/OwnerType/md5sum> <http://openworm.org/entities/OwnerType/propertyName>
```

and the property is connected to its values like:

```
<http://openworm.org/entities/OwnerType_propertyName/md5sum> <http://openworm.org/entities/SimpleProp
```

The following API calls do not yet exist, but would be excellent next functions to implement

3.4 Population()

A collection of cells. Constructor creates an empty population.

3.4.1 Population.filterCells(filters : ListOf(PairOf(unboundMethod, methodArgument))) : Population

Allows for groups of cells to be created based on shared properties including neurotransmitter, anatomical location or region, cell type.

Example:

```
p = Worm.cells()
p1 = p.filterCells([(Cell.lineageName, "AB")]) # A population of cells with AB as the blast cell
```

3.5 NeuroML()

A utility for generating NeuroML files from other objects. The semantics described *above* do not apply here.

3.5.1 NeuroML.generate(object : {Network, Neuron, IonChannel}, type : {0,1,2}) : neuroml.NeuroMLDocument

Get a NeuroML object that represents the given object. The `type` determines what content is included in the NeuroML object:

- 0=full morphology+biophysics
- 1=cell body only+biophysics
- 2=full morphology only

3.5.2 NeuroML.write(document : neuroml.NeuroMLDocument, filename : String)

Write out a NeuroMLDocument

CHAPTER 4

Issues

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

W