Locate Me Documentation

Release hadrian

Luis Pedro Coelho and Shannon Quinn

Sep 27, 2017

Contents

Waldo tells what everyone already knows.

We have a manuscript in preparation. If you use this in a publication, please let us know so we can (to the best of our knowledge), give you the right citation.

You can access the waldo webservice at http://murphylab.web.cmu.edu/services/waldo/home

CHAPTER 1

Example

:: import waldo.uniprot.retrieve from waldo.go import id_to_term

name = 'ACTB_HUMAN' gos = waldo.uniprot.retrieve.retrieve_go_annotations(name)

for g in gos: print id_to_term(g)

This prints out:

```
axon
ortical cytoskeleton
ytoskeleton
ytosol
xtracellular vesicular exosome
LL5-L complex
uA4 histone acetyltransferase complex
ostsynaptic density
ibonucleoprotein complex
```

CHAPTER 2

Contents

Installing Waldo

Getting the code

If you use `pip` or `easy_install`, you should be able to install waldo with:

```
pip install waldo
easy_install waldo
```

If you prefer to install from source, you can get either **'the released version <>'__** or the bleeding edge.

Once you download the code, you should be able to install it with:

```
python setup.py install
```

Dependencies

- python
- lxml
- sqlalchemy
- bottle

bottle is only needed if wish to run the web application (i.e., if you only using the local programming API, then you can skip this step).

Under a debian or Ubuntu system, the following commands will install all needed packages:

```
sudo apt-get install python-lxml
sudo apt-get install python-sqlalchemy
sudo apt-get install python-bottle
```

Downloading and building the database

```
update-waldo --user --unsafe --verbose
```

The `--user` flag installs the database just for this user (but does not require super-user [root] permissions).

The `--unsafe` flag makes the process much faster. However, it also means that if the process is interrupted, it will need to be started from scratch.

The `--verbose` flag, as expected, make the whole process more verbose.

Waldo Tutorial

We assume you have already installed waldo and its databases. We will show you how to use it as a library.

Our task will be to find out more information about 3 mouse proteins, listed using MGI (Mouse Genome Informatics) gene IDs:

1. Actn1
2. Cdc42
3. Fah

If we want to look up the GO locations within MGI, we could simply do:

```
import waldo.mgi
annotations = waldo.mgi.retrieve_go_annotations('Actn1')
print annotations
```

Prints out:

```
[u'GO:0000139',
 u'GO:0005622',
 u'GO:0005622',
 u'GO:0005623',
 u'GO:0005737',
 u'GO:0005737',
 u'GO:0005737',
 u'GO:0005856',
 u'GO:0005886',
 u'GO:0005886',
 u'GO:0016020',
 u'GO:0030141',
 u'GO:0030175',
 u'GO:0030496',
 u'GO:0042995',
 u'GO:0043005',
 u'GO:0043025',
 u'GO:0045177',
 u'GO:0051233',
 u'GO:0071944',
 u'GO:0072686']
```

These are Gene Ontology IDs, but they are hard to understand, we can get the English version with `waldo.go.id_to_term`:

```
from waldo.go import id_to_term
print map(id_to_term, annotations)
```

Now, you see:

```
[u'Golgi membrane',
 u'intracellular',
 u'intracellular',
 u'cell',
 u'cytoplasm',
 u'cytoplasm',
 u'cytoplasm',
 u'cytoskeleton',
 u'plasma membrane',
 u'plasma membrane',
 u'membrane',
 u'secretory granule',
 u'filopodium',
 u'midbody',
 u'cell projection',
 u'neuron projection',
 u'neuronal cell body',
 u'apical part of cell',
 u'spindle midzone',
 u'cell periphery',
 u'mitotic spindle']
```

To get information on all the genes we had above, we can now just use standard Python constructs:

```
genes = ['Actn1', 'Cdc42', 'Fah']
annotations = {}
for g in genes:
    annotations[g] = waldo.mgi.retrieve_go_annotations(g)
```

Other Databases

First we need to deal with **identifiers**. Proteins can be identified in many ways. Mapping identifiers is itself often a big problem.

Waldo uses **ENSEMBL identifiers** as the common identifiers. It knows how to convert identifiers from the other databases to Ensembl and back. Above, we listed MGI symbols, so they worked with MGI look up. Now, we will convert them to see what Uniprot has to say.

First, we get the ensembl gene ID:

```
from waldo import translate
for g in genes:
    print translate(g, 'mgi:symbol', 'ensembl:gene_id')
```

This prints out (see the identifier section to learn about the identifiers that Waldo knows about):

```
ENSMUSG00000015143
ENSMUSG00000006699
ENSMUSG00000030630
```

To get a uniprot name, we need two steps:

```
for g in genes:
    e = translate(g, 'mgi:symbol', 'ensembl:gene_id')
    uname = translate(e, 'ensembl:gene_id', 'uniprot:name')
    print uname
```

To get:

```
ACTN1_MOUSE
CDC42_MOUSE
FAAA_MOUSE
```

We now just look these up using the Uniprot module:

```
import waldo.uniprot

for g in genes:
    e = translate(g, 'mgi:symbol', 'ensembl:gene_id')
    uname = translate(e, 'ensembl:gene_id', 'uniprot:name')
    print waldo.uniprot.retrieve_go_annotations(uname)
```

Voilà!

Identifier Translation

One of waldo's features is the ability to translate between different gene identifiers.

It knows about the following identifier types:

- `embl:cds` *EMBL CDS*
- `ensembl:peptide_id` *ENSEMBL Peptide ID*
- `ensembl:gene_id` *ENSEMBL Gene ID*
- `ensembl:transcript_id` *ENSEMBL Transcript ID*
- `mgi:id` *MGI ID*
- `mgi:symbol` *MGI Symbol*
- `mgi:name` *MGI Name*
- `refseq:accession` *RefSeq Accession*
- `uniprot:name` *Uniprot Name*
- `uniprot:accession` *Uniprot Accession*
- `locate:id` *Locate ID*
- `hpa:id` *Human Protein Atlas ID*

The strings like `ensembl:gene_id` are the ones used in the code.

Here is a simple example of how to translate Uniprot accessions to Uniprot names:

```
from waldo import translate
accessions = [
    'P60709',
    'P07437',
    'Q9BQE3',
```

```
    'Q9NY65',
]
for a in accessions:
    n = translate(a, 'uniprot:accession', 'uniprot:name')
    print('{} -> {}'.format(a, n))
```

Prints out:

```
P60709 -> ACTB_HUMAN
P07437 -> TBB5_HUMAN
Q9BQE3 -> TBA1C_HUMAN
Q9NY65 -> TBA8_HUMAN
```

Waldo Architecture

Each datasource (Uniprot, MGI, LOCATE) has its own independent module, but each of the datasource modules adheres to a common interface.

Each module should: -load information from its corresponding flat files into the local relational database -perform translations from the datasource-specific protein identifiers to the global ensembl gene IDs, enabling easy searching across all datasources -be able to retrieve all GO terms corresponding to a given ensembl gene ID / datasource-specific identifier

The web front-end can then call these methods for all the available datasources when a query is issued.

Datasource files are stored in the "data/" folder, and are referenced independently in each module. As the number of modules grows, this may be redesigned. However, these are only needed when the data is loaded into the local relational database, which should happen very infrequently.

Install & Update

In order to periodically update the datasources, new versions of the files need to be pulled down from the servers. Executing the following command will perform this task:

```
./bin/update-waldo --user --unsafe --verbose
```

Note: this might take a while, e.g. Uniprot takes about an hour at the time of writing.

Local Database Structure

For every data source (LOCATE, Uniprot, MGI, etc), there were decisions made regarding what data to extract and store locally and what information was left in the downloaded files (datasources), possibly for future use. This is an explanation of the rationale.

LOCATE

LOCATE contains significant information in its datasources. Here is a brief list of the information available:

-Protein (organism, name, function, sequence, location) -Transcript (isoforms) -Experimental data (images, colocalization images, location) -External database annotations (locations from external sources) -Literature evidence (research citations, location) -Subcellular location predictions (location, method of prediction) -Motifs (name, position, type,

status) -Memos (memos, methods, scores) -External References (identifiers into external data sources) -Topology (methods).

Of these categories, we currently only thoroughly capture information pertaining to the protein, transcript, external annotations, literature, predictions, and external references, as these are the only topics that are most directly associated with subcellular location.

The experimental images, topology, and motifs could be very useful for future work involving subcellular location prediction.

Uniprot

Currently the information we capture from Uniprot involves accession numbers (which map directly to Uniprot entries), GO annotations, comments (particularly pertaining to subcellular location), and direct references. There is a significant amount of other information, particularly in the form of external database references that is not being used (as only those directly referencing Ensembl IDs are currently used). This could be useful later to cross-reference against other data authorities.

MGI

MGI's gene annotations consists of a tab-delimited file where unique entry identifiers may be repeated throughout the datasource in order to express a one-to-many relationship with the other values of the additional columns. The only information from the gene annotations file that is currently used is the unique MGI identifier, the ensembl ID (cross-referenced from MGI's MRK_Ensembl file), and the GO location terms.

Much of MGI's data is split up across many different downloadable files, as opposed to huge single files similar to the other data sources. Literature evidence, for example, is also not currently used (as the PubMed IDs for each MGI entry are in yet a third file), but there are plans to incorporate this information very soon.

Full API Documentation

WALDO : WHERE PROTEINS ARE

This is a Python library to collect information about proteins from online databases (Uniprot, MGI, ...) and expose it as an easy to use Python library.

`waldo.`**`translate`**(*name*, *input_namespace*, *output_namespace*, *session=None*)

name = translate(name, input_namespace, output_namespace, session={backend.create_session()})

Translate from one namespace to another.

name [str] input name

input_namespace [str] namespace to translate from (must be a known namespace)

output_namespace [str] namespace to translate to (must be a known namespace)

session [SQLAlchemy sesion object] SQLAlchemy session to use (default: call backend.create_session())

name [str or None] result of translation or None if not found.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

T

W