ontobio Documentation

Release 2.8.14

Chris Mungall

Sep 22, 2023

Contents:

Library for working with ontologies and ontology associations.

Provides:

- Transparent access to both local files (obo-json, GAF) and remote services (OntoBee, GO/GOlr, Monarch, Wikidata)
- Powerful graph operations for traversing logical structure of ontologies
- object model for working with ontology metadata elements (synonyms, etc)
- Access to gene product functional annotations in GO
- Access to gene/variant/disease/genotype etc info from Monarch
- Simple basis for building bioinformatics analyses and applications (e.g. enrichment)
- Underpinnings for web service APIs
- Rich command line access for non-programmers (see *Command Line*)
- Examples in *Notebooks*

CHAPTER 1

Compatibility

ontobio requires Python 3.4+.

CHAPTER 2

Contributing

https://github.com/biolink/ontobio

CHAPTER 3

Installation

You can install ontobio with pip:

```
$ pip install ontobio
```

CHAPTER 4

Documentation

4.1 Installation

Ontobio requires Python version 3.4 or higher

Install with pip:

```
pip install ontobio
```

4.1.1 Development Version

The development version can be downloaded from GitHub.

```
git clone https://github.com/biolink/ontobio.git
cd ontobio
pip install -e .[dev,test]
```

4.1.2 With pyvenv

```
cd ontobio
pyvenv venv
source venv/bin/activate
export PYTHONPATH=.:$PYTHONPATH
pip install -r requirements.txt
```

4.2 Quick start

This guide assumes you have already installed ontobio. If not, then follow the steps in the *Installation* section.

4.2.1 Command Line

You can use a lot of the functionality without coding a line of python, via the command line wrappers in the *bin* directory. For example, to search on ontology for matching labels:

```
ogr.py -r mp %cerebellum%
```

See the *Command Line* section for more details.

4.2.2 Notebooks

We provide Jupyter Notebooks to illustrate the functionality of the python library. These can also be used interactively.

See the *Notebooks* section for more details.

4.2.3 Python

This code example shows some of the basics of working with remote ontologies and associations

```
from ontobio.ontol_factory import OntologyFactory
from ontobio.assoc_factory import AssociationSetFactory

## label IDs for convenience
MOUSE = 'NCBITaxon:10090'
NUCLEUS = 'GO:0005634'
TRANSCRIPTION_FACTOR = 'GO:0003700'
PART_OF = 'BFO:0000050'

## Create an ontology object containing all of GO, with relations filtered
ofactory = OntologyFactory()
ont = ofactory.create('go').subontology(relations=['subClassOf', PART_OF])

## Create an AssociationSet object with all mouse GO annotations
afactory = AssociationSetFactory()
aset = afactory.create(ontology=ont,
                       subject_category='gene',
                       object_category='function',
                       taxon=MOUSE)

genes = aset.query([TRANSCRIPTION_FACTOR],[NUCLEUS])
print("Mouse TF genes NOT annotated to nucleus: {}".format(len(genes)))
for g in genes:
    print("  Gene: {} {}".format(g,aset.label(g)))
```

See the notebooks for more examples. For more documentation on specific components, see the rest of these docs, or skip forward to the *API* docs.

4.2.4 Web Services

See the biolink section

4.3 Command Line

A large subset of ontobio functionality is available via a powerful command line interface that can be used by non-programmers.

You will first need to install, see *Installation*

After that, set up your PATH:

```
export PATH $HOME/repos/ontobio/ontobio/bin
ogr -h
```

For many operations you need to be connected to a network

Note: command line interface may change

4.3.1 Live Demo

You can see the tour on asciinema:

- ontology querying tour
- how remote sparql works

4.3.2 Ontologies

The `ogr` command handles ontologies

Connecting to ontologies

Specify an ontology with the `-r` option. this will always be the OBO name, for example `go`, `cl`, `mp`, etc

- `-r go` connect to GO via default method (currently OntoBee-SPARQL)
- `-r obo:go` connect to GO via download and cache of ontology from OBO Library PURL
- `-r /users/my/my-ontologies/go.json` use local download of ontology

See *Inputs* for possible sources to connect to

In the following we assume default method, but the `-r` argument can be substituted.

Basic queries

Show all classes named `neuron`:

```
ogr -r cl neuron
```

Multiple arguments can be provided, e.g.:

```
ogr -r cl neuron hepatocyte erythrocyte
```

Ancestors queries

List all ancestors:

```
ogr -r cl neuron
```

Show ancestors as tree, following only subclass:

```
ogr -r cl -p subClassOf -t tree neuron
```

generates:

```
% GO:0005623 ! cell
 % CL:0000003 ! native cell
  % CL:0000255 ! eukaryotic cell
   % CL:0000548 ! animal cell
    % CL:0002319 ! neural cell
     % CL:0000540 ! neuron *
  % CL:0002371 ! somatic cell
   % CL:0002319 ! neural cell
    % CL:0000540 ! neuron *
```

Descendants of neuron, parts and subtypes

```
ogr -r cl -p subClassOf -p BFO:0000050 -t tree -d d neuron
```

Descendants and ancestors of neuron, parts and subtypes

```
ogr -r cl -p subClassOf -p BFO:0000050 -t tree -d du neuron
```

All ancestors of all classes 2 levels down from subclass-roots within CL:

```
ogr -r cl -P CL -p subClassOf -t tree -d u -L 2
```

Visualization using obographviz

Requires: https://www.npmjs.com/package/obographviz

Add og2dot.js to path

```
ogr -p subClassOf BFO:0000050 -r go -t png   a nucleus
```

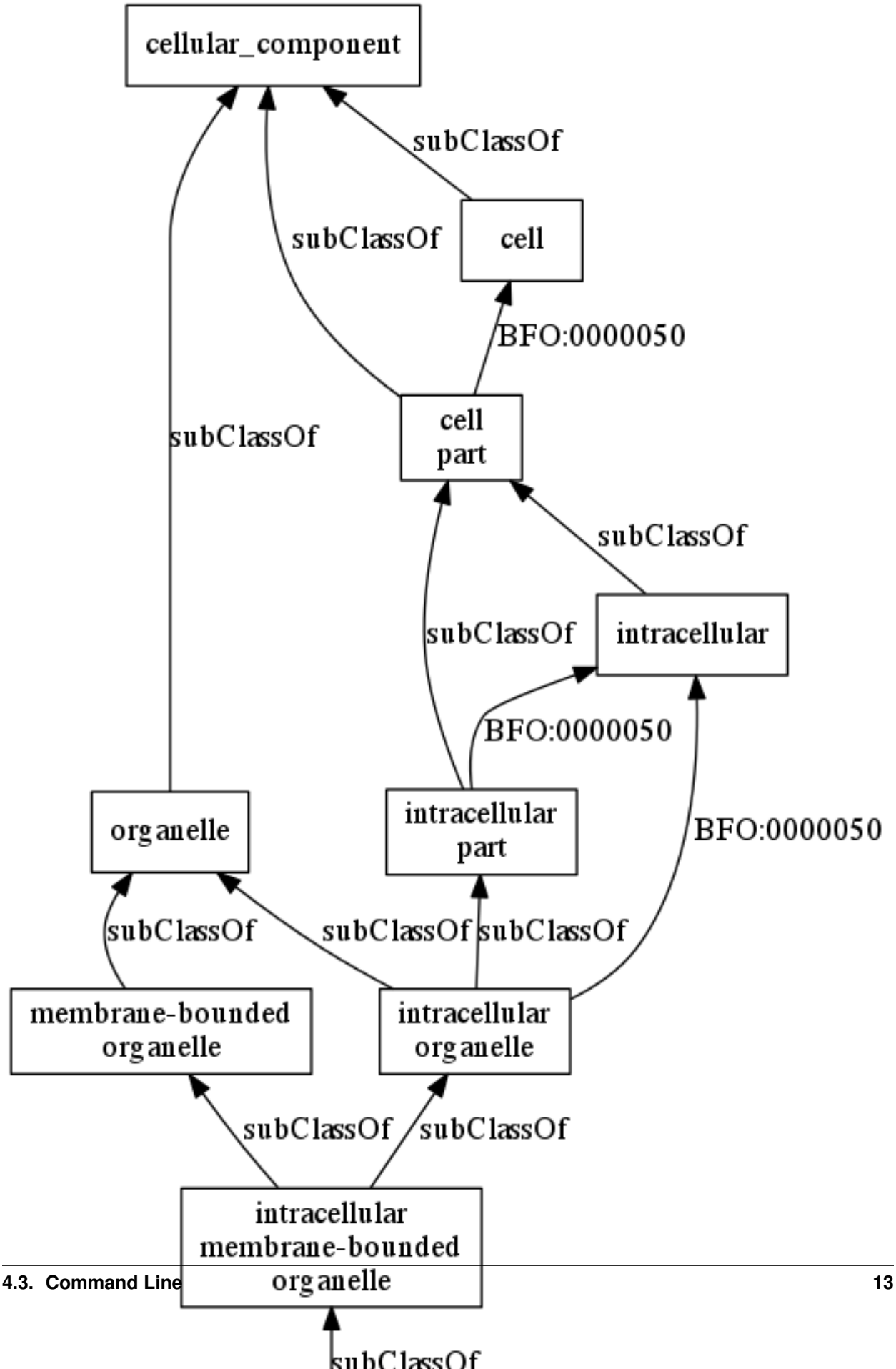
This proceeds by:

1. Using the python ontobio library to extract a networkx subgraph around the specified node
2. Write as obographs-json
3. Calls og2dot.js

Output:

Search

List exact matches to neuron


cellular_component
subClassOf
cell
subClassOf
BFO:0000050
cell part
subClassOf
subClassOf
intracellular
BFO:0000050
organelle
intracellular part
BFO:0000050
subClassOf
subClassOf
subClassOf
membrane-bounded organelle
intracellular organelle
subClassOf
subClassOf
intracellular membrane-bounded organelle
subClassOf

```
ogr -r cl neuron
```

Terms starting with neuron, SQL style

```
ogr -r cl neuron%
```

Terms starting with neuron, regex (equivalent to above)

```
ogr -r cl -s r ^neuron
```

Terms ending with neuron

```
ogr -r cl -s r neuron$
```

Terms containing the string neuron

```
ogr -r cl -s r neuron
```

Note: any of the above can be fed into other renderers, e.g. trees, graphs

E.g. terms containing neuron

```
ogr -r cl %neuron%
```

E.g. terms ending neuron, to tree

```
ogr -r cl %neuron -t tree
```

Properties

Properties (relations) are treated as nodes in the graph, e.g.

```
ogr-tree -d ud -r ro 'develops from'

. RO:0002324 ! developmentally related to
 % RO:0002258 ! developmentally preceded by
  % RO:0002202 ! develops from *
   % RO:0002225 ! develops from part of
   % RO:0002494 ! transformation of
    % RO:0002495 ! immediate transformation of
   % RO:0002207 ! directly develops from
    % RO:0002495 ! immediate transformation of
```

SPARQL integration

SPARQL where clauses can be inserted using `-Q` to pipe the results of a query to generate the initial set of IDs, e.g.:

```
ogr-tree  -r pato -Q "{?x rdfs:subClassOf+ PATO:0000052}"
```

4.3.3 Associations

The `ontobio-assoc` command handles ontologies

Subcommands:

```
subontology          Extract sub-ontology
enrichment           Perform an enrichment test
phenolog             Perform multiple enrichment tests
query                Query based on positive and negative terms
associations         Query for association pairs
intersections        Query intersections
dendrogram           Plot dendrogram from intersections
simmatrix            Plot dendrogram for similarities between subjects
```

Examples

Enrichment analysis, using all genes associated to a GO term as sample (we expect this GO term to be top results)

```
ontobio-assoc -v -r go -T NCBITaxon:9606 -C gene function enrichment -q GO:1903010
```

Plotly:

```
ontobio-assoc -v -r go -T NCBITaxon:10090 -C gene function dendrogram GO:0003700␣
↪GO:0005215 GO:0005634 GO:0005737 GO:0005739 GO:0005694 GO:0005730  GO:0000228␣
↪GO:0000262
```

Show similarity matrix for a set of genes:

```
ontobio-assoc -v -r go -T NCBITaxon:10090 -C gene function simmatrix MGI:1890081␣
↪MGI:97487 MGI:106593 MGI:97250 MGI:2151057 MGI:1347473
```

Basic queries, using file as input:

```
ontobio-assoc -C gene function -T pombe -r go -f tests/resources/truncated-pombase.
↪gaf query -q GO:0005622
```

4.3.4 Parsing assoc files

The `ontobio-parse-assocs.py` command will parse, validate and convert association files (GAF, GPAD, HPOA etc) of all file types and versions.

Top Level Options

`ontobio-parse-assocs.py` mostly uses top level options before subcommands to configure parsing.

- `-r, --resource` is the ontology file, in OBO JSON format
- `-f, --file` input annotation file
- `-F, --format` is the format of the input file. GAF will be the default if not provided
- `--report-md` and `--report-json` are the paths to output the parsing and validation reports to

Use `validate` to produce a report validating the input file, `-f, --file`.

Use `convert` to convert the input annotation file into a GPAD or GAF of any version. A report will still be produced. * `-t, --to` is the format to convert to. `GAF`, `GPAD` are accepted. * `-n, --format-version` is the version. For GAF, 2.1 or 2.2 are accepted with 1.2 as default. For GPAD 1.2 or 2.0 are accepted with 1.2 default.

GO Rules

`ontobio-parse-assocs.py` is capable of running the GO Rules (https://github.com/geneontology/go-site/tree/master/metadata/rules) over each annotation as they are parsed. By default, in this script, annotations are not validated by GO Rules except gorule-0000020, gorule-0000027, and gorule-0000059.

To include a rule in the rule set use the option `-l` or `--rule` followed by an integer representing the rule ID.

For example to include gorule-0000006:

```
ontobio-parse-assocs.py -f my_assoc.gaf --report-md report.md -l 6 validate
```

Use multiple `-l <ID>` to build up a list of rules that will be used to validate the input file:

```
ontobio-parse-assocs.py -f my_assoc.gaf --report-md report.md -l 6 -l 13 validate
```

To turn on all rules at once, use `-l all`:

```
ontobio-parse-assocs.py -f my_assoc.gaf --report-md report.md -l all validate
```

Under the hood, this is all controlled using a parameter, `rule_set` attached to the AssocParserConfig class. This accepts a list of integers or the string `"all"` or `None`. Setting to `None` (the default) will include no rules, and using `"all"` will use all rules.

The parameter passed in is used to create the `assocparser.RuleSet` dataclass.

4.3.5 GOlr Queries

The `qbiogolr.py` command is for querying a GOlr instance

4.4 Notebooks

4.4.1 Jupyter Notebook Examples

We use Jupyter Notebooks

Browse the examples: Ontobio Jupyter Notebooks

4.4.2 Running Jupyter Locally

Follow the instructions for installing from GitHub in *Installation*. Then start a notebook browser with:

```
PYTHONPATH=.. jupyter notebook
```

4.5 Basic Concepts

4.5.1 Ontologies

We leverage `networkx`

- obographs

- motivation

Class: *Ontology*

```
from ontobio.ontol_factory import OntologyFactory
ont = OntologyFactory().create("go")
[nucleus] = ont.search('nucleus')
ancestors = ont.ancestors(nucleus)
```

Alternatives

Ontobio is aimed primarily at bioinformatics applications, which typically have lightweight ontology requirements: navigation and grouping via graph structures, access to basic metadata such as synonyms.

4.5.2 Associations

The association model is a generalization of the GO association/annotation model. The typical scenario is to link a biological entity (gene, gene product, protein complex, variant or allele, disease, individual organism) to a descriptive ontology class, via a defined relationship type, plus metadata such as provenance and evidence. Note that it can be generalized further also link two entities (e.g. gene-gene, such as homology or relationship) or two ontology classes. In fact the distinction between *entities* and *ontology nodes* is one of convenience.

Categories

TODO

Lightweight vs Complete

For many purposes, it is only necessary to use a very lightweight representation of associations, as a collection of pairwise mappings between *subjects* and *objects*. This can be found in the class *AssociationSet*. An association set can be constructed using a particular set of criteria - e.g. all GO annotations to all zebrafish genes.

For other purposes it is necessary to have a full-blown representation, in which each association is modeled complete with evidence, provenance and so on. **TODO** Link to documentation.

Example Asssociation Set

This example shows a simple set of pairwise associations:

```
from ontobio.assoc_factory import AssociationSetFactory
afactory = AssociationSetFactory()
aset = afactory.create(ontology=ont,
                       subject_category='gene',
                       object_category='function',
                       taxon='NCBITaxon:7955') ## Zebrafish
```

Assocations vs ontology edges

The distinction between an *association* (aka *annotation*) and an ontology edge is primarily one of convenience. For example, it is possible to combine diseases, phenotypes and the associations between them in one graph, with re-

lationship type *has-phenotype* connecting these. Similary, gene could be added to a GO molecular function graph, connecting via *capable-of*.

By stratifying the two sets of entities and using a different data structure to connect these, we make it easier to define and perform certain operations, e.g. enrichment, semantic similarity, machine learning, etc.

But we also provide means of interconverting between these two perspectives (**TODO**).

See also

- GPAD
- OBAN

Class: *AssociationSet*

4.5.3 Identifiers

Ontobio uses CURIEs to identify entities, e.g. OMIM:123, GO:0001850. See *Identifiers* for more information

4.6 Inputs

Ontobio is designed to work with either *local* files or with *remote* information accessed via Services.

Access is generally mediated using a *factory* object. The client requests an ontology via a *handle* to the factory, and the factory will return with the relevant implementation instantiated.

4.6.1 Local JSON ontology files

You can load an ontology from disk (or a URL) that conforms to the obographs JSON standard.

Command line example:

```
ogr.py -r path/to/my/file.json
```

Code example, using an *OntologyFactory*

```
from ontobio.ontol_factory import OntologyFactory
ont = OntologyFactory().create("/path/to/my/file.json")
```

4.6.2 Local OWL and OBO-Format files

Requirement: OWLTools

Command line example:

```
ogr.py -r path/to/my/file.owl
```

Code example, using an *OntologyFactory*

```
from ontobio.ontol_factory import OntologyFactory
ont = OntologyFactory().create("/path/to/my/file.owl")
```

4.6.3 Local SKOS RDF Files

SKOS is an RDF data model for representing thesauri and terminologies.

See the SKOS primer for more details.

Command line example:

```
ogr.py -r path/to/my/skosfile.ttl
```

Code example, using an *OntologyFactory*

```
from ontobio.ontol_factory import OntologyFactory
ont = OntologyFactory().create("skos:/path/to/my/skosfile.ttl")
```

4.6.4 Remote SPARQL ontology access

The default SPARQL service used is the OntoBee one, which provides access to all OBO library ontologies

Warning: May change in future

Command line example:

```
ogr.py -r cl
```

Note that the official OBO library prefix must be used, e.g. `cl`, `go`, `hp`. See http://obofoundry.org/

Code example, using an *OntologyFactory*

```
from ontobio.ontol_factory import OntologyFactory
ont = OntologyFactory().create("cl")
```

4.6.5 Remote SciGraph ontology access

Warning: Experimental

Command line example:

```
ogr.py -r scigraph:ontology
```

Code example, using an *OntologyFactory*

```
from ontobio.ontol_factory import OntologyFactory
ont = OntologyFactory().create("scigraph:ontology")
```

Warning: Since SciGraph contains multiple graphs interwoven together, care must be taken on queries that don't use relationship types, as ancestor/descendant lists may be large

4.6.6 Local GAF or GPAD association files

The `ontobio.AssociationSet` class provides a lightweight way of storing sets of associations.

Code example: parse all associations from a GAF, and filter according to provider:

```
p = GafParser()
assocs = p.parse(open(POMBASE,"r"))
pombase_assocs = [a for a in assocs if a['provided_by'] == 'UniProt']
```

Code example, creating *AssociationSet* objects, using an *AssociationSetFactory*

```
afactory = AssociationSetFactory()
aset = afactory.create_from_file(file=args.assocfile,ontology=ont)
```

4.6.7 Remote association access via GOlr

GOlr is the name given to the Solr instance used by the Gene Ontology and Planteome projects. This has been generalized for use with the Monarch Initiative project.

GOlr provides fast access and faceted search on top of *Associations* (see the *Basic Concepts* section for more on the concept of associations). Ontobio provides both a transparent facade over GOlr, and also direct access to advanced queries.

By default an *eager* loading strategy is used: given a set of query criteria (minimally, subject and object *categories* plus a taxon, but optionally including evidence etc), all asserted pairwise associations are loaded into an association set. E.g.

```
aset = afactory.create(ontology=ont,
                       subject_category='gene',
                       object_category='function',
                       taxon=MOUSE)
```

Additionally, this is cached so future calls will not invoke the service overhead.

For performing advanced analytic queries over the complete GOlr database, see the *GolrAssociationQuery* class. **TODO** provide examples.

4.6.8 Remote association access via wikidata

TODO

4.6.9 Use of caching

When using remote services to access ontology or association set objects, caching is automatically used to avoid repeated access. Currently an *eager* strategy is used, in which large blocks are fetched in advance, though in future *lazy* strategies are optionally employed.

4.6.10 To be implemented

- Remote access to SciGraph/Neo4J
- Remote access to Chado databases
- Remote access to Knowledge Beacons

4.7 Outputs and Visualization

See the GraphRenderer module

4.7.1 Graphviz Output

Dependency: obographviz

4.7.2 Matplotlib Output

TODO

4.7.3 plotly

TODO

4.7.4 JSON output

TODO

4.7.5 OBO-Format output

4.8 Identifiers

4.8.1 URIs, prefixes and CURIEs

4.8.2 Mapping

4.9 Ontology-Based Analyses

Warning: In the future the analysis methods may migrate from the *AssociationSet* class to dedicated analysis engine classes.

4.9.1 Enrichment

See the Notebook example

OntoBio allows for generalized gene set enrichment: given a set of annotations that map genes to descriptor terms, and an input set of genes, and a background set, find what terms are enriched in the input set compared to the background.

With OntoBio, enrichment tests work for any annotation corpus, not necessarily just gene-oriented. For example, disease-phenotype. However, care must be taken with underlying assumptions with non-gene sets.

The very first thing you need to do before an enrichment analysis is fetch both an *Ontology* object and an *AsssociationSet* object. This could be a mix of local files or remote service/database. See *Inputs* for details.

Assume that we are using a remote ontology and local GAF:

```
from ontobio import OntologyFactory
from ontobio import AssociationSetFactory
ofactory = OntologyFactory()
afactory = AssociationSetFactory()
ont = ofactory.create('go')
aset = afactory.create_from_gaf('my.gaf', ontology=ont)
```

Assume also that we have a set of sample and background gene IDs, the test is:

```
enr = aset.enrichment_test(subjects=gene_ids, background=background_gene_ids,␣
↪threshold=0.00005, labels=True)
```

This returns a list of dicts (**TODO** - decide if we want to make this an object and follow a standard class model)

NOTE the input gene IDs *must* be the same ones used in the AssociationSet. If you load from a GAF, this is the IDs that are formed by combining col1 and col2, separated by a ":". E.g. UniProtKB:P123456

What if you have different IDs? Or what if you just have a list of gene symbols? In this case you will need to *map* these names or IDs, the subject of the next section.

Reproducibility

For reproducible analyses, use a **versioned PURL** for the ontology

Command line wrapper

You can use the *ontobio-assoc* command to run enrichment analyses. Some examples:

Create a gene set for all genes in "regulation of bone development" (GO:1903010). Find other terms for which this is enriched (in human)

```
# find all mouse genes that have 'abnormal synaptic transmission' phenotype
# (using remote sparql service for MP, and default (Monarch) for associations
ontobio-assoc.py -v -r mp -T NCBITaxon:10090 -C gene phenotype query -q MP:0003635 >␣
↪genes.txt

# get IDs
cut -f1 -d ' ' genes.txt > genes.ids

# enrichment, using GO
ontobio-assoc.py  -r go -T NCBITaxon:10090 -C gene function enrichment -s genes.ids

# resulting GO terms are not very surprising...
2.48e-12 GO:0045202 synapse
2.87e-11 GO:0044456 synapse part
3.66e-08 GO:0007270 neuron-neuron synaptic transmission
3.95e-08 GO:0098793 presynapse
1.65e-07 GO:0099537 trans-synaptic signaling
1.65e-07 GO:0007268 chemical synaptic transmission
```

Further reading

For API docs, see enrichment_test in AssociationSet model

4.9.2 Identifier Mapping

TODO

4.9.3 Semantic Similarity

TODO

To follow progress, see this PR

4.9.4 Slimming

TODO

4.9.5 Graph Reduction

TODO

4.9.6 Lexical Analyses

See the lexmap API docs

You can also use the command line:

```
ontobio-lexmap.py ont1.json ont2.json > mappings.tsv
```

The inputs can be any kind of handle - a local ontology file or a remote ontology accessed via services.

For example, this will work:

    ontobio-lexmap.py mp hp wbphenotype > mappings.tsv

See *Inputs* for more details.

For examples of lexical mapping pipelines, see:

- https://github.com/cmungall/sweet-obo-alignment
- ‘<https://github.com/monarch-initiative/monarch-disease-ontology/tree/master/src/icd10>_

These have examples of customizing configuration using a yaml file.

4.10 Advanced Ontology Use

TODO

4.11 GO Rules Engine

GO Rules are data quality validation checks for Gene Ontology annotation data. All GO Rules are defined here and represent what valid Annotation Data should look like.

In Ontobio, when we parse GPAD or GAF annotations using *ontobio.io.gafparser.GafParser* or ontobio.io.gpadparser.GpadParser we can validate each annotation line on each rule defined in *ontobio.io.qc*.

Any line that fails a rule will have a message made in ontobio.io.assocparser.Report.

The GO Rules engine is defined in *ontobio.io.qc* and is where new rules should be implemented.

4.11.1 Rules Definition

A GO Rule implementation works by implementing a function that encodes the logic of the defined GO Rule defined in a rule markdown in the rule definitions

In code, a Rule consists of an ID, title, fail_mode, and optionally rule tags.

- The ID is the Curie style rule ID, like GORULE:0000013 (referring to GORULE:0000013)
- The title should be more or less direct from the rule definition in go-site. For example in GORULE:0000006 the title is "IEP and HEP usage is restricted to terms from the Biological Process ontology" and that should be used here.
- fail_mode comes from the rule's SOP.md. Annotations that fail a GO Rule that have a HARD fail_mode will be filtered and SOFT will be kept, but with a warning message.
- Tags should be copied over from the rule definition as well. For example GORULE:0000058 has a tag "context-import". This is used to signal extra information about rules and how they should be run. In the GoRule definition, there is a *_is_run_from_context* which detects if a rule should be run given the context in the ontobio.io.assocparser.AssocParserConfig rule_contexts.

A rule class will provide its own definition of test() which should perform the logic of the rule, returning a TestResult. In the majority of cases, the helper method _result(passes: bool) should be used which will perform some default behavior given True for passing and False for failing the given rule.

4.11.2 How to Write a New Rule Implementation

1. Create a new class subclassing GoRule, typically named after the rule ID number.

```
class GoRule02(GoRule):

    def __init__(self):
        pass
```

2. Write an __init__ calling the super GoRule init, defining the relavent values for your new rule.

```
class GoRule02
    def __init__(self):
        super().__init__("GORULE:0000002", "No 'NOT' annotations to 'protein binding ;
↪ GO:0005515'", FailMode.SOFT)
        # Note title in second argument copied from gorule-0000002 definition
```

3. Override test() implementing the logic of your rule. The annotation is the incoming annotation as a GoAssociation, the config holds important metadata about the current running instance and has resources like the ontology. Note that all identifiers that can be are proper CURIEs, defined by the *ontobio.model.association.Curie*, so must be wrapped in str to compare against a string.

```
def test(self, annotation: association.GoAssociation, config: assocparser.
↪AssocParserConfig, group=None) -> TestResult:
    """
    Fake rule that passes only annotations to the GO Term GO:0003674 molecular␣
↪function
    """
    return self._result(str(annotation.object.id) == "GO:0003674")
```

4. Add new Rule Instance to the `GoRules` enum. This is how you register a rule with the runner system, so it gets run automatically by ontobio.
5. Write Tests for your rule in tests/test_qc.py

Implmentation Notes

Rules can generally use the `self._result(bool)` helper function instead of producing a TestResult manually. True for Passing, False for Failing. This method will take care of the fail mode, messages, etc, automatically.

For slightly more control, use the `result(bool, FailMode)` function to create the correct `ResultType`.

Rules that perform repairs on incoming `GoAssociations` can be done by instead subclassing `RepairRule`.

In general, when testing an annotation, the GoAssociation instance is passed along to each rule implementation. In a RepairRule the result will contain the updated annotation. So the runner will grab this updated annotation, passing it along to the next rule down the line. In this way annotations under test may accumulate repairs across the length of the rules.

As a matter of policy, if a rule requires a resource, the `test` implmentation should test that the `AssocParserConfig` has that resource defined, and automatically pass the rule if it is not preseent. In the future, we could instead have a "skip" state that encapsulates this.

Also, each rule implementation should complete as fast as possible, and not delay. Any long computation should be cached - so at least only the first run of a rule will be slow. See rules where we compute sublcass closures, like `ontobio.io.qc.GoRule07`.

4.12 API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

4.12.1 API

Ontology Access

Factory

The OntologyFactory class provides a means of creating an ontology object backed by either local files or remote services. See *Inputs* for more details.

class ontobio.ontol_factory.**OntologyFactory**(*handle=None*)

Implements a factory for generating `Ontology` objects.

You should use a factory object rather than initializing *Ontology* directly. See *Inputs* for more details.

initializes based on an ontology name

Parameters **handle** (*str*) – see *create*

create(*handle=None*, *handle_type=None*, ***args*)
Creates an ontology based on a handle

Handle is one of the following

- *FILENAME.json* : creates an ontology from an obographs json file
- *obo:ONTID* : E.g. obo:pato - creates an ontology from obolibrary PURL (requires owltools)
- *ONTID* : E.g. 'pato' - creates an ontology from a remote SPARQL query

Parameters **handle** (*str*) – specifies how to retrieve the ontology info

Ontology Object Model

class ontobio.ontol.**Ontology**(*handle=None*, *id=None*, *graph=None*, *xref_graph=None*, *meta=None*, *payload=None*, *graphdoc=None*)
An object that represents a basic graph-oriented view over an ontology.

The ontology may be represented in memory, or it may be located remotely. See subclasses for details.

The default implementation is an in-memory wrapper onto the python networkx library

initializes based on an ontology name.

Note: do not call this directly, use OntologyFactory instead

add_node(*id*, *label=None*, *type='CLASS'*, *meta=None*)
Add a new node to the ontology

add_parent(*id*, *pid*, *relation='subClassOf'*)
Add a new edge to the ontology

add_synonym(*syn*)
Adds a synonym for a node

add_text_definition(*textdef*)
Add a new text definition to the ontology

add_to_subset(*id*, *s*)
Adds a node to a subset

add_xref(*id*, *xref*)
Adds an xref to the xref graph

all_obsoletes()
Returns all obsolete nodes

all_synonyms(*include_label=False*)
Retrieves all synonyms

Parameters **include_label** (*bool*) – If True, include label/names as Synonym objects

Returns *Synonym* objects

Return type list[*Synonym*]

ancestors(*node*, *relations=None*, *reflexive=False*)
Return all ancestors of specified node.

The default implementation is to use networkx, but some implementations of the Ontology class may use a database or service backed implementation, for large graphs.

Parameters

- **node** (*str*) – identifier for node in ontology
- **reflexive** (*bool*) – if true, return query node in graph
- **relations** (*list*) – relation (object property) IDs used to filter

Returns ancestor node IDs

Return type list[str]

child_parent_relations(*subj*, *obj*, *graph=None*)

Get all relationship type ids between a subject and a parent.

Typically only one relation ID returned, but in some cases there may be more than one

Parameters

- **subj** (*string*) – Child (subject) id
- **obj** (*string*) – Parent (object) id

Returns

Return type list

children(*node*, *relations=None*)

Return all direct children of specified node.

Wraps networkx by default.

Parameters

- **node** (*string*) – identifier for node in ontology
- **relations** (*list of strings*) – list of relation (object property) IDs used to filter

create_slim_mapping(*subset=None*, *subset_nodes=None*, *relations=None*, *disable_checks=False*)

Create a dictionary that maps between all nodes in an ontology to a subset

Parameters

- **ont** (*Ontology*) – Complete ontology to be mapped. Assumed pre-filtered for relationship types
- **subset** (*str*) – Name of subset to map to, e.g. goslim_generic
- **nodes** (*list*) – If no named subset provided, subset is passed in as list of node ids
- **relations** (*list*) – List of relations to filter on
- **disable_checks** (*bool*) – Unless this is set, this will prevent a mapping being generated with non-standard relations. The motivation here is that the ontology graph may include relations that it is inappropriate to propagate gene products over, e.g. transports, has-part

Returns maps all nodes in ont to one or more non-redundant nodes in subset

Return type dict

Raises `ValueError` – if the subset is empty

descendants(*node*, *relations=None*, *reflexive=False*)

Returns all descendants of specified node.

The default implementation is to use networkx, but some implementations of the Ontology class may use a database or service backed implementation, for large graphs.

Parameters

- **node** (*str*) – identifier for node in ontology
- **reflexive** (*bool*) – if true, return query node in graph
- **relations** (*list*) – relation (object property) IDs used to filter

Returns descendant node IDs

Return type list[str]

equiv_graph()

Returns bidirectional networkx graph of all equivalency relations

Return type graph

extract_subset(*subset*, *contract=True*)
Return all nodes in a subset.

We assume the oboInOwl encoding of subsets, and subset IDs are IRIs, or IR fragments

filter_redundant(*ids*)
Return all non-redundant ids from a list

get_filtered_graph(*relations=None*, *prefix=None*)
Returns a networkx graph for the whole ontology, for a subset of relations

Only implemented for eager methods.

Implementation notes: currently this is not cached

Parameters

- **relations** (–) – list of object property IDs, e.g. subClassOf, BFO:0000050. If empty, uses all.
- **prefix** (–) – if specified, create a subgraph using only classes with this prefix, e.g. ENVO, PATO, GO

Returns A networkx MultiDiGraph object representing the filtered ontology

Return type nx.MultiDiGraph

get_graph()
Return a networkx graph for the whole ontology.

Note: Only implemented for *eager* implementations

Returns A networkx MultiDiGraph object representing the complete ontology

Return type nx.MultiDiGraph

get_level(*level*, *relations=None*, ***args*)
Get all nodes at a particular level

Parameters **relations** (*list[str]*) – list of relations used to filter

get_property_chain_axioms(*nid*)
Retrieves property chain axioms for a class id

Parameters **nid** (*str*) – Node identifier for relation to be queried

Returns

Return type PropertyChainAxiom

get_roots(*relations=None*, *prefix=None*)
Get all nodes that lack parents

Parameters

- **relations** (*list[str]*) – list of relations used to filter
- **prefix** (*str*) – E.g. GO. Exclude nodes that lack this prefix when testing parentage

has_node(*id*)
True if id identifies a node in the ontology graph

inline_xref_graph()
Copy contents of xref_graph to inlined meta object for each node

is_obsolete(*nid*)
True if node is obsolete

Parameters **nid** (*str*) – Node identifier for entity to be queried

label(*nid*, *id_if_null=False*)
Fetches label for a node

Parameters

- **nid** (*str*) – Node identifier for entity to be queried
- **id_if_null** (*bool*) – If True and node has no label return id as label

Returns

Return type str

logical_definitions(*nid*)
Retrieves logical definitions for a class id

Parameters **nid** (*str*) – Node identifier for entity to be queried

Returns

Return type *LogicalDefinition*

merge(*ontologies*)
Merges specified ontology into current ontology

node(*id*)
Return a node with a given ID. If the node with the ID exists the Node object is returned, otherwise None is returned.

Wraps networkx by default

node_type(*id*)
If stated, either CLASS, PROPERTY or INDIVIDUAL

nodes()
Return all nodes in ontology

Wraps networkx by default

parent_index(*relations=None*)
Returns a mapping of nodes to all direct parents

Parameters

- **relations** (*list[str]*) – list of relations used to filter
- **Returns** –

- **list** – list of lists [[CLASS_1, PARENT_1,1, ..., PARENT_1,N], [CLASS_2, PARENT_2,1, PARENT_2,2, ...] ...]

parents(*node*, *relations=None*)
Return all direct 'parents' of specified node.

Note that in the context of ontobio, 'parent' means any node that is traversed in a single hop along an edge from a subject to object. For example, if the ontology has an edge "finger part-of some hand", then "hand" is the parent of finger. This can sometimes be counter-intutitive, for example, if the ontology contains has-part axioms. If the ontology has an edge "X receptor activity has-part some X binding", then "X binding" is the 'parent' of "X receptor activity" over a has-part edge.

Wraps networkx by default.

Parameters

- **node** (*string*) – identifier for node in ontology
- **relations** (*list of strings*) – list of relation (object property) IDs used to filter

prefix(*nid*)
Return prefix for a node

prefix_fragment(*nid*)
Return prefix and fragment/localid for a node

prefixes()
list all prefixes used

relations_used()
Return list of all relations used to connect edges

replaced_by(*nid*, *strict=True*)
Returns value of 'replaced by' (IAO_0100001) property for obsolete nodes

Parameters

- **nid** (*str*) – Node identifier for entity to be queried
- **strict** (*bool*) – If true, raise error if cardinality>1. If false, return list if cardinality>1

Returns

Return type None if no value set, otherwise returns node id (or list if multiple values, see strict setting)

resolve_names(*names*, *synonyms=False*, ***args*)
returns a list of identifiers based on an input list of labels and identifiers.

Parameters

- **names** (*list*) – search terms. '%' treated as wildcard
- **synonyms** (*bool*) – if true, search on synonyms in addition to labels
- **is_regex** (*bool*) – if true, treats each name as a regular expression
- **is_partial_match** (*bool*) – if true, treats each name as a regular expression .*name.*

search(*searchterm*, ***args*)
Simple search. Returns list of IDs.

Parameters

- **searchterm** (*list*) – search term. '%' treated as wildcard

- **synonyms** (*bool*) – if true, search on synonyms in addition to labels
- **is_regex** (*bool*) – if true, treats each name as a regular expression
- **is_partial_match** (*bool*) – if true, treats each name as a regular expression .*name.*

Returns match node IDs

Return type list

sorted_nodes()
Returns all nodes in ontology, after topological sort

subgraph(*nodes=None*)
Return an induced subgraph

By default this wraps networkx subgraph, but this may be overridden in specific implementations

subontology(*nodes=None*, *minimal=False*, *relations=None*)
Return a new ontology that is an extract of this one

Parameters

- **nodes** (–) – list of node IDs to include in subontology. If None, all are used
- **relations** (–) – list of relation IDs to include in subontology. If None, all are used

subsets(*nid*, *contract=True*)
Retrieves subset ids for a class or ontology object

synonyms(*nid*, *include_label=False*)
Retrieves synonym objects for a class

Parameters

- **nid** (*str*) – Node identifier for entity to be queried
- **include_label** (*bool*) – If True, include label/names as Synonym objects

Returns *Synonym* objects

Return type list[*Synonym*]

text_definition(*nid*)
Retrieves logical definitions for a class or relation id

Parameters **nid** (*str*) – Node identifier for entity to be queried

Returns

Return type TextDefinition

traverse_nodes(*qids*, *up=True*, *down=False*, ***args*)
Traverse (optionally) up and (optionally) down from an input set of nodes

Parameters

- **qids** (*list[str]*) – list of seed node IDs to start from
- **up** (*bool*) – if True, include ancestors
- **down** (*bool*) – if True, include descendants
- **relations** (*list[str]*) – list of relations used to filter

Returns nodes reachable from qids

Return type list[str]

xrefs(*nid*, *bidirectional=False*, *prefix=None*)
Fetches xrefs for a node

Parameters

- **nid** (*str*) – Node identifier for entity to be queried
- **bidirection** (*bool*) – If True, include nodes xreffed to nid

Returns

Return type list[str]

class ontobio.ontol.**Synonym**(*class_id*, *val=None*, *pred='hasRelatedSynonym'*, *lextype=None*, *xrefs=None*, *ontology=None*, *confidence=1.0*, *synonymType=None*)
Represents a synonym using the OBO model

Parameters

- **class_id** (–) – the class that is being defined
- **val** (–) – the synonym itself
- **pred** (–) – oboInOwl predicate used to model scope. One of: has{Exact,Narrow,Related,Broad}Synonym - may also be 'label'
- **lextype** (–) – From an open ended set of types
- **xrefs** (–) – Provenance or cross-references to same usage

as_dict()
Returns Synonym as obograph dict

class ontobio.ontol.**LogicalDefinition**(*class_id*, *genus_ids*, *restrictions*)
A simple OWL logical definition conforming to the pattern:

```
class_id = (genus_id_1 AND ... genus_id_n) AND (P_1 some FILLER_1) AND ... (P_m
↪some FILLER_m)
```

See obographs docs for more details

Parameters

- **class_id** (*string*) – the class that is being defined
- **genus_ids** (*list*) – a list of named classes (typically length 1)
- **restrictions** (*list*) – a list of (PROPERTY_ID, FILLER_CLASS_ID) tuples

Assocation Access

Factory

class ontobio.assoc_factory.**AssociationSetFactory**
Factory for creating AssociationSets

Currently support for golr (GO and Monarch) is provided but other stores possible

initializes based on an ontology name

create(*ontology=None*, *subject_category=None*, *object_category=None*, *evidence=None*, *taxon=None*, *relation=None*, *file=None*, *fmt=None*, *skim=True*)
creates an AssociationSet

Currently, this uses an eager binding to a *ontobio.golr* instance. All compact associations for the particular combination of parameters are fetched.

Parameters

- **ontology** (an *Ontology* object) –
- **subject_category** (*string representing category of subjects (e.g. gene, disease, variant)*) –
- **object_category** (*string representing category of objects (e.g. function, phenotype, disease)*) –
- **taxon** (*string holding NCBITaxon:nnnn ID*) –

create_from_assocs(*assocs*, ***args*)
Creates from a list of association objects

create_from_file(*file=None*, *fmt='gaf'*, *skim=True*, ***args*)
Creates from a file. If fmt is set to None then the file suffixes will be used to choose a parser.

Parameters

- **file** (*str or file*) – input file or filename
- **fmt** (*str*) – name of format e.g. gaf

create_from_gaf(*file*, ***args*)
Creates from a GAF file

create_from_phenopacket(*file*)
Creates from a phenopacket file

create_from_remote_file(*group*, *snapshot=True*, ***args*)
Creates from remote GAF

create_from_simple_json(*file*)
Creates from a simple json rendering

create_from_tuples(*tuples*, ***args*)
Creates from a list of (subj,subj_name,obj) tuples

Assocation Object Model

class ontobio.assocmodel.**AssociationSet**(*ontology=None*, *association_map=None*, *subject_label_map=None*, *meta=None*)
An object that represents a collection of associations

NOTE: the intention is that this class can be subclassed to provide either high-efficiency implementations, or implementations backed by services or external stores. The default implementation is in-memory.

NOTE: in general you do not need to call this yourself. See assoc_factory

initializes an association set, which minimally consists of:

- an ontology (e.g. GO, HP)
- a map between subjects (e.g genes) and sets/lists of term IDs

annotations(*subject_id*)
Returns a list of classes used to describe a subject

@Deprecated: use objects_for_subject

as_dataframe (*fillna=True*, *subjects=None*)
Return association set as pandas DataFrame

Each row is a subject (e.g. gene) Each column is the inferred class used to describe the subject

associations (*subject*, *object=None*)
Given a subject-object pair (e.g. gene id to ontology class id), return all association objects that match.

enrichment_test (*subjects=None*, *background=None*, *hypotheses=None*, *threshold=0.05*, *labels=False*, *direction='greater'*)
Performs term enrichment analysis.

Parameters

- **subjects** (*string list*) – Sample set. Typically a gene ID list. These are assumed to have associations
- **background** (*string list*) – Background set. If not set, uses full set of known subject IDs in the association set
- **threshold** (*float*) – p values above this are filtered out
- **labels** (*boolean*) – if true, labels for enriched classes are included in result objects
- **direction** (*'greater', 'less' or 'two-sided'*) – default is greater - i.e. enrichment test. Use 'less' for depletion test.

index ()
Creates indexes based on inferred terms.

You do not need to call this yourself; called on initialization

inferred_types (*subj*)
Returns: set of reflexive inferred types for a subject.

E.g. if a gene is directly associated with terms A and B, and these terms have ancestors C, D and E then the set returned will be {A,B,C,D,E}

Parameters – **ID string** (*subj*) –

Returns: set of class IDs

static intersectionlist_to_matrix (*ilist*, *xterms*, *yterms*)
WILL BE DEPRECATED

Replace with method to return pandas dataframe

jaccard_similarity (*s1*, *s2*)
Calculate jaccard index of inferred associations of two subjects

|ancs(s1) ∧ ancs(s2)| — |ancs(s1) ∨ ancs(s2)|

label (*id*)
return label for a subject id

Will make use of both the ontology and the association set

objects_for_subject (*subject_id*)
Returns a list of classes used to describe a subject

query (*terms=None*, *negated_terms=None*)
Basic boolean query, using inference.

Parameters

- **terms** (–) – list

  list of class ids. Returns the set of subjects that have at least one inferred annotation to each of the specified classes.

- **negated_terms** (–) – list

  list of class ids. Filters the set of subjects so that there are no inferred annotations to any of the specified classes

query_associations(*subjects=None*, *infer_subjects=True*, *include_xrefs=True*)
Query for a set of associations.

Note: only a minimal association model is stored, so all results are returned as (subject_id,class_id) tuples

Parameters

- **subjects** – list

  list of subjects (e.g. genes, diseases) used to query associations. Any association to one of these subjects or a descendant of these subjects (assuming infer_subjects=True) are returned.

- **infer_subjects** – boolean (default true)

  See above

- **include_xrefs** – boolean (default true)

  If true, then expand inferred subject set to include all xrefs of those subjects.

Example: if a high level disease node (e.g. DOID:14330 Parkinson disease) is specified, then the default behavior (infer_subjects=True, include_xrefs=True) and the ontology includes DO, results will include associations from both descendant DOID classes, and all xrefs (e.g. OMIM)

query_intersections(*x_terms=None*, *y_terms=None*, *symmetric=False*)
Query for intersections of terms in two lists

Return a list of intersection result objects with keys:

- x : term from x
- y : term from y
- c : count of intersection
- j : jaccard score

similarity_matrix(*x_subjects=None*, *y_subjects=None*, *symmetric=False*)
Query for similarity matrix between groups of subjects

Return a list of intersection result objects with keys:

- x : term from x
- y : term from y
- c : count of intersection
- j : jaccard score

subontology(*minimal=False*)
Generates a sub-ontology based on associations

termset_ancestors(*terms*)
reflexive ancestors

Parameters – a set or list of class IDs(*terms*)–

Returns: set of class IDs

TODO - detailed association modeling

Association File Parsers

class ontobio.io.gafparser.**GafParser**(*config=None*, *group='unknown'*, *dataset='unknown'*, *bio_entities=None*)

Parser for GO GAF format

config : a AssocParserConfig object

association_generator(*file*, *skipheader=False*, *outfile=None*) → Dict[KT, VT]

Returns a generator that yields successive associations from file

Yields *association*

map_to_subset(*file*, *outfile=None*, *ontology=None*, *subset=None*, *class_map=None*, *relations=None*)

Map a file to a subset, writing out results

You can pass either a subset name (e.g. goslim_generic) or a dictionary with ready-made mappings

Parameters

- **file** (*file*) – Name or file object for input assoc file
- **outfile** (*file*) – Name or file object for output (mapped) assoc file; writes to stdout if not set
- **subset** (*str*) – Optional name of subset to map to, e.g. goslim_generic
- **class_map** (*dict*) – Mapping between asserted class ids and ids to map to. Many to many
- **ontology** (*Ontology*) – Ontology to extract subset from

parse(*file*, *skipheader=False*, *outfile=None*)

Parse a line-oriented association file into a list of association dict objects

Note the returned list is of dict objects. TODO: These will later be specified using marshmallow and it should be possible to generate objects

Parameters

- **file** (*file or string*) – The file is parsed into association objects. Can be a http URL, filename or *file-like-object*, for input assoc file
- **outfile** (*file*) – Optional output file in which processed lines are written. This a file or *file-like-object*

Returns Associations generated from the file

Return type list

parse_line(*line*)

Parses a single line of a GAF

Return a tuple *(processed_line, associations)*. Typically there will be a single association, but in some cases there may be none (invalid line) or multiple (disjunctive clause in annotation extensions)

Note: most applications will only need to call this directly if they require fine-grained control of parsing. For most purposes, **:method:'parse_file'** can be used over the whole file

Parameters **line** (*str*) – A single tab-seperated line from a GAF file

skim(*file*)
Lightweight parse of a file into tuples.

Note this discards metadata such as evidence.

Return a list of tuples (subject_id, subject_label, object_id)

upgrade_empty_qualifier(*assoc: ontobio.model.association.GoAssociation*) → ontobio.model.association.GoAssociation
From https://github.com/geneontology/go-site/issues/1558

For GAF 2.1 we will apply an algorithm to find a best fit relation if the qualifier column is empty. If the qualifiers field is empty, then:

If the GO Term is exactly GO:008150 Biological Process, then the qualifier should be *involved_in* If the GO Term is exactly GO:0008372 Cellular Component, then the qualifer should be *is_active_in* If the GO Term is a Molecular Function, then the new qualifier should be *enables* If the GO Term is a Biological Process, then the new qualifier should be *acts_upstream_or_within Otherwise for Cellular Component, if it's subclass of anatomical structure, than use 'located_in*

and if it's a protein-containing complexes, use *part_of*

Parameters **assoc** – GoAssociation

Returns the possibly upgraded GoAssociation

Go Rules

class ontobio.io.qc.**FailMode**
An enumeration.

class ontobio.io.qc.**GoRules**
An enumeration.

class ontobio.io.qc.**GoRulesResults**(*all_results*, *annotation*)
Create new instance of GoRulesResults(all_results, annotation)

all_results
Alias for field number 0

annotation
Alias for field number 1

class ontobio.io.qc.**RepairState**
An enumeration.

ontobio.io.qc.**ResultType**
alias of ontobio.io.qc.Result

class ontobio.io.qc.**TestResult**(*result_type: ontobio.io.qc.Result*, *message: str*, *result*)
Represents the result of a single association.GoAssociation being validated on some rule

Create a new TestResult

Parameters

- **result_type** (*ResultType*) – enum of PASS, WARNING, ERROR. Both WARNINGs and ERRORs are reported, but ERROR will filter the offending GoAssociation
- **message** (*str*) – Description of the failure of GoAssociation to pass a rule. This is usually just the rule title

- **result** – [description] True if the GoAssociation passes, False if not. If it's repaired, this is the updated, repaired, GoAssociation

ontobio.io.qc.**repair_result**(*repair_state: ontobio.io.qc.RepairState*, *fail_mode: ontobio.io.qc.FailMode*) → ontobio.io.qc.Result

Returns ResultType.PASS if the repair_state is OKAY, and WARNING if REPAIRED.

This is used by RepairRule implementations.

Parameters

- **repair_state** (RepairState) – If the GoAssocition was repaired during a rule, then this should be RepairState.REPAIRED, otherwise RepairState.OKAY
- **fail_mode** (FailMode) – [description]

Returns [description]

Return type ResultType

ontobio.io.qc.**result**(*passes: bool*, *fail_mode: ontobio.io.qc.FailMode*) → ontobio.io.qc.Result

Send True for passes, and this returns the PASS ResultType, and if False, then depending on the fail mode it returns either WARNING or ERROR ResultType.

GoAssociation internal Model

This contains the data model for parsing annotations from GAF and GPAD.

The idea is to make it easy to parse text lines of any source into a GoAssociation object and then give the GoAssociation object the ability to convert itself into GPAD or GAF of any version. Or any other format that is required.

class ontobio.model.association.**ConjunctiveSet**(*elements: List[T]*)

This respresents a comma separated list of objects which can be turned into strings.

This is used for the with/from and extensions fields in the GoAssociation.

The field *elements* can be a list of Curie or ExtensionUnit. Curie for with/from, and ExtensionUnit for extensions field.

display(*conjunct_to_str=<function ConjunctiveSet.<lambda>>*) → str

Convert this ConjunctiveSet to a string separateted by commas.

This calls *conjunct_to_str* (which defaults to *str*) on each element before joining. To use a different string representation of each element, pass in a different function. This functionality is used to differentiate between GPAD 1.2 and GPAD 2.0, where relations are written differently per version.

classmethod list_to_str(*conjunctions: List[T]*, *conjunct_to_str=<function ConjunctiveSet.<lambda>>*) → str

List should be a list of ConjunctiveSet Given [ConjunctiveSet, ConjunctiveSet], this will call ConjunctiveSet.display() using the *conjunct_to_str* function (which defaults to *str*) and join them with a pipe.

To have elements of the ConjunctiveSet displayed differently, use a different *conjunct_to_str* function. This functionality is used to differentiate between GPAD 1.2 and GPAD 2.0, where relations are written differently per version.

classmethod str_to_conjunctions(*entity: str*, *conjunct_element_builder: Union[C, ontobio.model.association.Error] = <function ConjunctiveSet.<lambda>>*) → Union[List[C], ontobio.model.association.Error]

Takes a field that conforms to the pipe (|) and comma (,) separator type. The parsed version is a list of pipe separated values which are themselves a comma separated list.

If the elements inside the comma separated list should not just be strings, but be converted into a value of a type, *conjunct_element_builder* can be provided which should take a string and return a parsed value or an instance of an Error type (defined above).

If there is an error in producing the values of the conjunctions, then this function will return early with the error.

This function will return a List of ConjunctiveSet

class ontobio.model.association.**Curie**(*namespace: str*, *identity: str*)
Object representing a Compact URI, with a namespace identifier along with an ID, like GO:1234567.

Use *from_str* to parse a string like "GO:1234567" into a Curie. The result should be checked for errors with *is_error*

class ontobio.model.association.**Date**(*year*, *month*, *day*, *time*)
Create new instance of Date(year, month, day, time)

day
Alias for field number 2

month
Alias for field number 1

time
Alias for field number 3

year
Alias for field number 0

class ontobio.model.association.**Error**(*info: str*, *entity: str = ''*)

class ontobio.model.association.**Evidence**(*type: ontobio.model.association.Curie, has_supporting_reference: List[ontobio.model.association.Curie], with_support_from: List[ontobio.model.association.ConjunctiveSet]*)

class ontobio.model.association.**ExtensionUnit**(*relation: ontobio.model.association.Curie*, *term: ontobio.model.association.Curie*)
An ExtensionUnit is a single element of the extensions field of GAF or GPAD. This consists of a relation and a term.

Create an ExtensionUnit with *from_str* or *from_curie_str*. If there is an error in parsing then *Error* is returned. Results from these functions should be checked for *Error*.

The string representation will depend on the format, and so the *display* method should be used. By default this will write the relation using the label with undercores (example: part_of) as defined in ontobio.rdfgen.relations.py. To write the relation as a CURIE (as in gpad 2.0), set parameter *use_rel_label* to *True*.

display(*use_rel_label=False*)
Turns the ExtensionUnit into a string. By default this uses the ontobio.rdfgen.relations module to lookup the relation label. To use the CURIE instead, pass *use_rel_label=True*.

classmethod from_curie_str(*entity: str*) → Union
Attempts to parse string entity as an ExtensionUnit If the *relation(term)* is not formatted correctly, an Error is returned. *relation* is a Curie, and so is any errors in formatting are delegated to Curie.from_str()

classmethod from_str(*entity: str*) → Union
Attempts to parse string entity as an ExtensionUnit If the *relation(term)* is not formatted correctly, an Error is returned. If the *relation* cannot be found in the *relations* dictionary then an error is also returned.

class ontobio.model.association.**GoAssociation**(*source_line: Optional[str], subject: ontobio.model.association.Subject, relation: ontobio.model.association.Curie, object: ontobio.model.association.Term, negated: bool, qualifiers: List[ontobio.model.association.Curie], aspect: Optional[NewType.<locals>.new_type], interacting_taxon: Optional[ontobio.model.association.Curie], evidence: ontobio.model.association.Evidence, subject_extensions: List[ontobio.model.association.ExtensionUnit], object_extensions: List[ontobio.model.association.ConjunctiveSet], provided_by: NewType.<locals>.new_type, date: ontobio.model.association.Date, properties: List[Tuple[str, str]]*)

The internal model used by the parsers and qc Rules engine that all annotations are parsed into.

If an annotation textual line cannot be parsed into a GoAssociation then it is not a well formed line.

This class provides several methods to convert this GoAssociation into other representations, like GAF and GPAD of each version, as well as the old style dictionary Association that this class replaced (for compatibility if needed).

Each parser has its own function or functions that converts an annotation line into a GoAssociation, and this is the first phase of parsing. In general, GoAssociations are only created by the parsers.

to_gaf_2_1_tsv() → List[T]
Converts the GoAssociation into a "TSV" columnar GAF 2.1 row as a list of strings.

to_gaf_2_2_tsv() → List[T]
Converts the GoAssociation into a "TSV" columnar GAF 2.2 row as a list of strings.

to_gpad_1_2_tsv() → List[T]
Converts the GoAssociation into a "TSV" columnar GPAD 1.2 row as a list of strings.

to_gpad_2_0_tsv() → List[T]
Converts the GoAssociation into a "TSV" columnar GAF 2.0 row as a list of strings.

to_hash_assoc() → dict
Converts the GoAssociation into the old style dictionary association for backwards compatibility

class ontobio.model.association.**Header**(*souce_line: Union[str, NoneType]*)

class ontobio.model.association.**Subject**(*id: ontobio.model.association.Curie, label: str, fullname: List[str], synonyms: List[str], type: Union[List[str], List[ontobio.model.association.Curie]], taxon: ontobio.model.association.Curie, encoded_by: List[ontobio.model.association.Curie] = None, parents: List[ontobio.model.association.Curie] = None, contained_complex_members: List[ontobio.model.association.Curie] = None, db_xrefs: List[ontobio.model.association.Curie] = None, properties: Dict = None*)

contained_complex_members = None
Optional, or cardinality 0+

db_xrefs = None
Optional, or cardinality 0+

encoded_by = None
Optional, or cardinality 0+

fullname = None
fullname is also *DB_Object_Name* in the GPI spec, cardinality 0+

fullname_field(*max=None*) → str
Converts the *fullname* or *DB_Object_Name* into the field text string used in files

label = None
label is also *DB_Object_Symbol* in the GPI spec

parents = None
Optional, or cardinality 0+

properties = None
Optional, or cardinality 0+

synonyms = None
Cardinality 0+

taxon = None
...

Type Should be NCBITaxon

type = None
In GPI 1.2, this was a string, corresponding to labels of the Sequence Ontology gene, protein_complex; protein; transcript; ncRNA; rRNA; tRNA; snRNA; snoRNA, any subclass of ncRNA. If the specific type is unknown, use *gene_product*.

When reading gpi 1.2, these labels should be mapped to the 2.0 spec, stating that the type must be a Curie in the Sequence Ontology OR Protein Ontology OR Gene Ontology

In GPI 1.2, there is only 1 value, and is required. In GPI 2.0 there is a minimum of 1, but maybe more.

If writing out to GPI 1.2/GAF just take the first value in the list.

class ontobio.model.association.**Term**(*id: ontobio.model.association.Curie*, *taxon: ontobio.model.association.Curie*)
Represents a Gene Ontology term

ontobio.model.association.**TwoTupleStr**(*items: List[str]*) → tuple
Create a tuple of of str that is guaranteed to be of length two from a list

If the list is larger, then only the first two elements will be used. If the list is smaller, then the empty string will be used

ontobio.model.association.**gp_type_label_to_curie**(*type: ontobio.model.association.Curie*) → str
This is the reverse of *map_gp_type_label_to_curie*

ontobio.model.association.**map_gp_type_label_to_curie**(*type_label: str*) → ontobio.model.association.Curie
Map entity types in GAF or GPI 1.2 into CURIEs in Sequence Ontology (SO), Protein Ontology (PRO), or Gene Ontology (GO).

This is a measure to upgrade the pseudo-labels into proper Curies. Present here are the existing set of labels in current use, and how they should be mapped into CURIEs.

GOlr Queries

class ontobio.golr.golr_query.**GolrAssociationQuery**(*subject_category=None*, *object_category=None*, *relation=None*, *relationship_type=None*, *subject_or_object_ids=None*, *subject_or_object_category=None*, *subject=None*, *subjects=None*, *object=None*, *objects=None*, *subject_direct=False*, *object_direct=False*, *subject_taxon=None*, *subject_taxon_direct=False*, *object_taxon=None*, *object_taxon_direct=False*, *invert_subject_object=None*, *evidence=None*, *exclude_automatic_assertions=False*, *q=None*, *id=None*, *use_compact_associations=False*, *include_raw=False*, *field_mapping=None*, *solr=None*, *config=None*, *url=None*, *select_fields=None*, *fetch_objects=False*, *fetch_subjects=False*, *fq=None*, *slim=None*, *json_facet=None*, *iterate=False*, *map_identifiers=None*, *facet_fields=None*, *facet_field_limits=None*, *facet_limit=25*, *facet_mincount=1*, *facet_pivot_fields=None*, *stats=False*, *stats_field=None*, *facet=True*, *pivot_subject_object=False*, *unselect_evidence=False*, *rows=10*, *start=None*, *homology_type=None*, *non_null_fields=None*, *user_agent=None*, *association_type=None*, *sort=None*, ***kwargs*)

A Query object providing a higher level of abstraction over either GO or Monarch Solr indexes

All of these can be set when creating a new object

fetch_objects : bool

we frequently want a list of distinct association objects (in the RDF sense). for example, when querying for all phenotype associations for a gene, it is convenient to get a list of distinct phenotype terms. Although this can be obtained by iterating over the list of associations, it can be expensive to obtain all associations.

Results are in the 'objects' field

fetch_subjects : bool

This is the analog of the fetch_objects field. Note that due to an inherent asymmetry by which the list of subjects can be very large (e.g. all genes in all species for "metabolic process" or "metabolic phenotype") it's necessary to combine this with subject_category and subject_taxon filters

Results are in the 'subjects' field

slim : List

a list of either class ids (or in future subset ids), used to map up (slim) objects in associations. This will populate an additional 'slim' field in each association object corresponding to the slimmed-up value(s) from the direct objects. If fetch_objects is passed, this will be populated with slimmed IDs.

evidence: String

Evidence class from ECO. Inference is used.

exclude_automatic_assertions : bool

If true, then any annotations with ECO evidence code for IEA or subclasses will be excluded.

use_compact_associations : bool

If true, then the associations list will be false, instead compact_associations contains a more compact representation consisting of objects with (subject, relation and objects)

config : Config

See Config for details. The config object can be used to set values for the solr instance to be queried

TODO - Extract params into their own object

Fetch a set of association objects based on a query.

exec(***kwargs*)

Execute solr query

Result object is a dict with the following keys:

- raw
- associations : list
- compact_associations : list
- facet_counts
- facet_pivot

infer_category(*id*)

heuristic to infer a category from an id, e.g. DOID:nnn –> disease

make_canonical_identifier(*id*)

E.g. MGI:MGI:nnnn –> MGI:nnnn

make_gostyle_identifier(*id*)
E.g. MGI:nnnn –> MGI:MGI:nnnn

map_id(*id*, *prefix*, *closure_list*)
Map identifiers based on an equivalence closure list.

solr_params()
Generate HTTP parameters for passing to Solr.

In general you should not need to call this directly, calling exec() on a query object will transparently perform this step for you.

translate_doc(*d*, *field_mapping=None*, *map_identifiers=None*, ***kwargs*)
Translate a solr document (i.e. a single result row)

translate_docs(*ds*, ***kwargs*)
Translate a set of solr results

translate_docs_compact(*ds*, *field_mapping=None*, *slim=None*, *map_identifiers=None*, *invert_subject_object=False*, ***kwargs*)
Translate golr association documents to a compact representation

translate_obj(*d*, *fname*)
Translate a field value from a solr document.

This includes special logic for when the field value denotes an object, here we nest it

translate_objs(*d*, *fname*, *default=None*)
Translate a field whose value is expected to be a list

class ontobio.golr.golr_query.**GolrSearchQuery**(*term=None*, *category=None*, *is_go=False*, *url=None*, *solr=None*, *config=None*, *fq=None*, *fq_string=None*, *hl=True*, *facet_fields=None*, *facet=True*, *search_fields=None*, *taxon_map=True*, *rows=100*, *start=None*, *prefix=None*, *boost_fx=None*, *boost_q=None*, *highlight_class=None*, *taxon=None*, *min_match=None*, *minimal_tokenizer=False*, *include_eqs=False*, *exclude_groups=False*, *user_agent=None*)
Controller for monarch and go solr search cores Queries over a search document

autocomplete()
Execute solr autocomplete

search()
Execute solr search query

Lexmap

class ontobio.lexmap.**LexicalMapEngine**(*wsmap={'': '', 'a': '', 'i': '1', 'ii': '2', 'iii': '3', 'iv': '4', 'ix': '9', 'of': '', 'the': '', 'v': '5', 'vi': '6', 'vii': '7', 'viii': '8', 'x': '10', 'xi': '11', 'xii': '12', 'xiii': '13', 'xiv': '14', 'xix': '19', 'xv': '15', 'xvi': '16', 'xvii': '17', 'xviii': '18', 'xx': '20'}*, *config=None*)
generates lexical matches between pairs of ontology classes

Parameters

- **wdmap** (*dict*) – maps words to normalized synonyms.
- **config** (*dict*) – A configuration conforming to LexicalMapConfigSchema

assign_best_matches(*xg*)
For each node in the xref graph, tag best match edges

cliques(*xg*)
Return all equivalence set cliques, assuming each edge in the xref graph is treated as equivalent, and all edges in ontology are subClassOf

Parameters **xg** (*Graph*) – an xref graph

Returns

Return type list of sets

compare_to_xrefs(*xg1*, *xg2*)
Compares a base xref graph with another one

get_xref_graph()
Generate mappings based on lexical properties and return as nx graph.

- A dictionary is stored between ref:*Synonym* values and synonyms. See ref:*index_synonym*. Note that Synonyms include the primary label
- Each key in the dictionary is examined to determine if there exist two Synonyms from different ontology classes

This avoids N^2 pairwise comparisons: instead the time taken is linear

After initial mapping is made, additional scoring is performed on each mapping

The return object is a nx graph, connecting pairs of ontology classes.

Edges are annotated with metadata about how the match was found:

syns: pair pair of *Synonym* objects, corresponding to the synonyms for the two nodes

score: int score indicating strength of mapping, between 0 and 100

Returns nx graph (bidirectional)

Return type Graph

grouped_mappings(*id*)
return all mappings for a node, grouped by ID prefix

index_ontology(*ont*)
Adds an ontology to the index

This iterates through all labels and synonyms in the ontology, creating an index

index_synonym(*syn*, *ont*)
Index a synonym

Typically not called from outside this object; called by *index_ontology*

score_xrefs_by_semsim(*xg*, *ont=None*)
Given an xref graph (see ref:*get_xref_graph*), this will adjust scores based on the semantic similarity of matches.

weighted_axioms(*x*, *y*, *xg*)
return a tuple (sub,sup,equiv,other) indicating estimated prior probabilities for an interpretation of a mapping between x and y.

See kboom paper

4.13 Additional Notes

How to contribute code to OntoBio

These guidelines are for developers of OntoBio, whether internal or in the broader community.

Mailing list

- [biolink-api google group](https://groups.google.com/forum/#!forum/biolink-api)

Code Style

- Use NumPy-style docstrings. See [Napoleon docs]((http://www.sphinx-doc.org/en/stable/ext/napoleon.html)

Basic principles of the Monarch-flavored [GitHub Workflow](http://guides.github.com/overviews/flow/)

Principle 1: Work from a personal fork * Prior to adopting the workflow, a developer will perform a *one-time setup* to create a personal Fork of the appropriate shared repo (e.g., *monarch-app*) and will subsequently perform their development and testing on a task-specific branch within their forked repo. This forked repo will be associated with that developer's GitHub account, and is distinct from the shared repo managed by the Monarch Initiative.

Principle 2: Commit to personal branches of that fork * Changes will never be committed directly to the master branch on the shared repo. Rather, they will be composed as branches within the developer's forked repo, where the developer can iterate and refine their code prior to submitting it for review.

Principle 3: Propose changes via pull request of personal branches * Each set of changes will be developed as a task-specific *branch* in the developer's forked repo, and then a [pull request](github.com/government/best-practices/compare) will be created to develop and propose changes to the shared repo. This mechanism provides a way for developers to discuss, revise and ultimately merge changes from the forked repo into the shared Monarch repo.

Principle 4: Delete or ignore stale branches, but don't recycle merged ones * Once a pull request has been merged, the task-specific branch is no longer needed and may be deleted or ignored. It is bad practice to reuse an existing branch once it has been merged. Instead, a subsequent branch and pull-request cycle should begin when a developer switches to a different coding task. * You may create a pull request in order to get feedback, but if you wish to continue working on the branch, so state with "DO NOT MERGE YET".

Table of contents

<!– MarkdownTOC –>

- **[One Time Setup - Forking a Shared Repo](#one-time-setup—forking-a-shared-repo)**
  - [Step 1 - Backup your existing repo (optional)](#step-1—backup-your-existing-repo-optional)
  - [Step 2 - Fork *monarch-app* via the Web](#step-2—fork-monarch-app-via-the-web)
  - [Step 3 - Clone the Fork Locally](#step-3—clone-the-fork-locally)
  - [Step 4 - Configure the local forked repo](#step-4—configure-the-local-forked-repo)
  - [Step 5 - Configure *.bashrc* to show current branch (optional)](#step-5—configure–bashrc-to-show-current-branch-optional)
- **[Typical Development Cycle](#typical-development-cycle)**
  - **[Refresh and clean up local environment](#refresh-and-clean-up-local-environment)**
    - [Step 1 - Fetch remotes](#step-1—fetch-remotes)
    - [Step 2 - Ensure that 'master' is up to date](#step-2—ensure-that-master-is-up-to-date)
  - [Create a new branch](#create-a-new-branch)

  - [Changes, Commits and Pushes](#changes-commits-and-pushes)
  - **[Reconcile branch with upstream changes](#reconcile-branch-with-upstream-changes)**
    - [Fetching the upstream branch](#fetching-the-upstream-branch)
    - [Rebasing to avoid Conflicts and Merge Commits](#rebasing-to-avoid-conflicts-and-merge-commits)
    - [Dealing with merge conflicts during rebase](#dealing-with-merge-conflicts-during-rebase)
    - [Advanced: Interactive rebase](#advanced-interactive-rebase)
  - [Submitting a PR (pull request)](#submitting-a-pr-pull-request)
  - [Reviewing a pull request](#reviewing-a-pull-request)
  - [Respond to TravisCI tests](#respond-to-travisci-tests)
  - [Respond to peer review](#respond-to-peer-review)
  - [Repushing to a PR branch](#repushing-to-a-pr-branch)
  - [Merge a pull request](#merge-a-pull-request)
  - [Celebrate and get back to work](#celebrate-and-get-back-to-work)
- [GitHub Tricks and Tips](#github-tricks-and-tips)
- [References and Documentation](#references-and-documentation)

<!– /MarkdownTOC –>

One Time Setup - Forking a Shared Repo

The official shared Monarch repositories (e.g., *monarch-app*, *phenogrid*) are intended to be modified solely via pull requests that are reviewed and merged by a set of responsible 'gatekeeper' developers within the Monarch development team. These pull requests are initially created as task-specific named branches within a developer's personal forked repo.

Typically, a developer will fork a shared repo once, which creates a personal copy of the repo that is associated with the developer's GitHub account. Subsequent pull requests are developed as branches within this personal forked repo. The repo need never be forked again, although each pull request will be based upon a new named branch within this forked repo.

Step 1 - Backup your existing repo (optional)

The Monarch team has recently adopted the workflow described in this document. Many developers will have an existing clone of the shared repo that they have been using for development. This cloned local directory must be *moved aside* so that a proper clone of the forked repo can be used instead.

If you do not have an existing local copy of the shared repo, then skip to [Step 2](#step-2—fork-monarch-app-via-the-web) below.

Because there may be valuable files stored in the developer's local directory but not stored in GitHub, we recommend that the developer keep this copy around for a few weeks until they are confident any useful information has been migrated. The following instructions should be effective at moving your existing *monarch-app* directory aside. Adapt these for use with *phenogrid* and other Monarch repos.

> cd . . . /monarch-app # Your local copy of the shared repo. > rm -rf ./node_modules # You won't need this anymore. Free up disk > cd .. > mv monarch-app monarch-app.old # Keep dir around, but avoid accidental use

Step 2 - Fork *monarch-app* via the Web

The easiest way to fork the *monarch-app* repository is via the GitHub web interface:

- Ensure you are logged into GitHub as your GitHub user.
- Navigate to the monarch-app shared repo at [https://github.com/monarch-initiative/monarch-app{]}(https://github.com/monarch-initiative/monarch-app).
- Notice the 'Fork' button in the upper right corner. It has a number to the right of the button.

 - Click the Fork button. The resulting behavior will depend upon whether your GitHub user is a member of a GitHub organization. If not a member of an organization, then the fork operation will be performed and the forked repo will be created in the user's account. - If your user is a member of an organization (e.g., monarch-initiative or acme-incorporated), then GitHub will present a dialog for the user to choose where to place the forked repo. The user should click on the icon corresponding to their username. - *If you accidentally click the number, you will be on the Network Graphs page and should go back.*

Step 3 - Clone the Fork Locally

At this point, you will have a fork of the shared repo (e.g., monarch-app) stored within GitHub, but it is not yet available on your local development machine. This is done as follows:

> # Assumes that directory ~/MI/ will contain your Monarch repos. # Assumes that your username is MarieCurie. # Adapt these instructions to suit your environment > cd ~/MI > git clone git@github.com:MarieCurie/monarch-app.git > cd monarch-app

Notice that we are using the SSH transport to clone this repo, rather than the HTTPS transport. The telltale indicator of this is the *git@github.com:MarieCurie...* rather than the alternative *https://github.com/MarieCurie....*

Note: If you encounter difficulties with the above 'git clone', you may need to associate your local public SSH key with your GitHub account. See [Which remote URL should I use?](https://help.github.com/articles/which-remote-url-should-i-use/) for information.

Step 4 - Configure the local forked repo

The *git clone* above copied the forked repo locally, and configured the symbolic name 'origin' to point back to the *remote* GitHub fork. We will need to create an additional *remote* name to point back to the shared version of the repo (the one that we forked in Step 2). The following should work:

> # Assumes that you are already in the local monarch-app directory > git remote add upstream https://github.com/monarch-initiative/monarch-app.git

Verify that remotes are configured correctly by using the command *git remote -v*. The output should resemble:

> upstream https://github.com/monarch-initiative/monarch-app.git (fetch) upstream https://github.com/monarch-initiative/monarch-app.git (push) origin git@github.com:MarieCurie/monarch-app.git (fetch) origin git@github.com:MarieCurie/monarch-app.git (push)

Step 5 - Configure *.bashrc* to show current branch (optional)

One of the important things when using Git is to know what branch your working directory is tracking. This can be easily done with the *git status* command, but checking your branch periodically can get tedious. It is easy to configure your *bash* environment so that your current git branch is always displayed in your bash prompt.

If you want to try this out, add the following to your *~/.bashrc* file:

> function parse_git_branch() {
>
> git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* (.*)/ 1/'
>
> } LIGHT_GRAYBG="[033[0;47m]" LIGHT_PURPLE="[033[0;35m]" NO_COLOR="[033[0m]" export PS1="$LIGHT_PURPLEw$LIGHT_GRAYBG$(parse_git_branch)$NO_COLOR $ "

You will need to open up a new Terminal window (or re-login to your existing terminal) to see the effect of the above *.bashrc* changes.

If you cd to a git working directory, the branch will be displayed in the prompt. For example:

> ~ $ ~ $ # This isn't a git directory, so no branch is shown ~ $ ~ $ cd /tmp /tmp $ /tmp $ # This isn't a git directory, so no branch is shown /tmp $ /tmp $ cd ~/MI/monarch-app/ ~/MI/monarch-app fix-feedback-button $ ~/MI/monarch-app fix-feedback-button $ # The current branch is shown ~/MI/monarch-app fix-feedback-button $ ~/MI/monarch-app fix-feedback-button $ git status On branch fix-feedback-button Changes not staged for commit:
>
> > (use "git add <file>..." to update what will be committed) (use "git checkout – <file>..." to discard changes in working directory)
> >
> > ... remaining output of git status elided ...

—

Typical Development Cycle

Once you have completed the One-time Setup above, then it will be possible to create new branches and pull requests using the instructions below. The typical development cycle will have the following phases:

- Refresh and clean up local environment
- Create a new task-specific branch
- Perform ordinary development work, periodically committing to the branch
- Prepare and submit a Pull Request (PR) that refers to the branch
- Participate in PR Review, possibly making changes and pushing new commits to the branch
- Celebrate when your PR is finally Merged into the shared repo.
- Move onto the next task and repeat this cycle

Refresh and clean up local environment

Git will not automatically sync your Forked repo with the original shared repo, and will not automatically update your local copy of the Forked repo. These tasks are part of the developer's normal *cycle*, and should be the first thing done prior to beginning a new development effort and creating a new branch. In addition, this

Step 1 - Fetch remotes

In the (likely) event that the *upstream* repo (the monarch-app shared repo) has changed since the developer last began a task, it is important to update the local copy of the upstream repo so that its changes can be incorporated into subsequent development.

> > git fetch upstream # Updates the local copy of shared repo BUT does not affect the working directory, it simply makes the upstream code available locally for subsequent Git operations. See step 2.

Step 2 - Ensure that 'master' is up to date

Assuming that new development begins with branch 'master' (a good practice), then we want to make sure our local 'master' has all the recent changes from 'upstream'. This can be done as follows:

> > git checkout master > git reset –hard upstream/master

The above command is potentially dangerous if you are not paying attention, as it will remove any local commits to master (which you should not have) as well as any changes to local files that are also in the upstream/master version (which you should not have). In other words, the above command ensures a proper clean slate where your local master branch is identical to the upstream master branch.

Some people advocate the use of *git merge upstream/master* or *git rebase upstream/master* instead of the *git reset –hard*. One risk of these options is that unintended local changes accumulate in the branch and end up in an eventual pull request. Basically, it leaves open the possibility that a developer is not really branching from upstream/master, but is branching from some developer-specific branch point.

Create a new branch

Once you have updated the local copy of the master branch of your forked repo, you can create a named branch from this copy and begin to work on your code and pull-request. This is done with:

> git checkout -b fix-feedback-button # This is an example name

This will create a local branch called 'fix-feedback-button' and will configure your working directory to track that branch instead of 'master'.

You may now freely make modifications and improvements and these changes will be accumulated into the new branch when you commit.

If you followed the instructions in [Step 5 - Configure *.bashrc* to show current branch (optional)](#step-5—configure–bashrc-to-show-current-branch-optional), your shell prompt should look something like this:

~/MI/monarch-app fix-feedback-button $

Changes, Commits and Pushes

Once you are in your working directory on a named branch, you make changes as normal. When you make a commit, you will be committing to the named branch by default, and not to master.

You may wish to periodically *git push* your code to GitHub. Note the use of an explicit branch name that matches the branch you are on (this may not be necessary; a git expert may know better):

> git push origin fix-feedback-button # This is an example name

Note that we are pushing to 'origin', which is our forked repo. We are definitely NOT pushing to the shared 'upstream' remote, for which we may not have permission to push.

Reconcile branch with upstream changes

If you have followed the instructions above at [Refresh and clean up local environment](#refresh-and-clean-up-local-environment), then your working directory and task-specific branch will be based on a starting point from the latest-and-greatest version of the shared repo's master branch. Depending upon how long it takes you to develop your changes, and upon how much other developer activity there is, it is possible that changes to the upstream master will conflict with changes in your branch.

So it is a good practice to periodically pull down these upstream changes and reconcile your task branch with the upstream master branch. At the least, this should be performed prior to submitting a PR.

Fetching the upstream branch

The first step is to fetch the update upstream master branch down to your local development machine. Note that this command will NOT affect your working directory, but will simply make the upstream master branch available in your local Git environment.

> git fetch upstream

Rebasing to avoid Conflicts and Merge Commits

Now that you've fetched the upstream changes to your local Git environment, you will use the *git rebase* command to adjust your branch

> # Make that your changes are committed to your branch > # before doing any rebase operations > git status

... Review the git status output to ensure your changes are committed # ... Also a good chance to double-check that you are on your # ... task branch and not accidentally on master

> git rebase upstream/master

The rebase command will have the effect of adjusting your commit history so that your task branch changes appear to be based upon the most recently fetched master branch, rather than the older version of master you may have used when you began your task branch.

By periodically rebasing in this way, you can ensure that your changes are in sync with the rest of Monarch development and you can avoid hassles with merge conflicts during the PR process.

Dealing with merge conflicts during rebase

Sometimes conflicts happen where another developer has made changes and committed them to the upstream master (ideally via a successful PR) and some of those changes overlap with the code you are working on in your branch. The *git rebase* command will detect these conflicts and will give you an opportunity to fix them before continuing the rebase operation. The Git instructions during rebase should be sufficient to understand what to do, but a very verbose explanation can be found at [Rebasing Step-by-Step](http://gitforteams.com/resources/rebasing.html)

Advanced: Interactive rebase

As you gain more confidence in Git and this workflow, you may want to create PRs that are easier to review and best reflect the intent of your code changes. One technique that is helpful is to use the *interactive rebase* capability of Git to help you clean up your branch prior to submitting it as a PR. This is completely optional for novice Git users, but it does produce a nicer shared commit history.

See [squashing commits with rebase](http://gitready.com/advanced/2009/02/10/squashing-commits-with-rebase.html) for a good explanation.

Submitting a PR (pull request)

Once you have developed code and are confident it is ready for review and final integration into the upstream version, you will want to do a final *git push origin ...* (see Changes, Commits and Pushes above). Then you will use the GitHub website to perform the operation of creating a Pull Request based upon the newly pushed branch.

See [submitting a pull request](https://help.github.com/articles/creating-a-pull-request).

Reviewing a pull request

The set of open PRs for the monarch-app can be viewed by first visiting the shared monarch-app GitHub page at [https://github.com/monarch-initiative/monarch-app{]}(https://github.com/monarch-initiative/monarch-app).

Click on the 'Pull Requests' link on the right-side of the page:

Note that the Pull Request you created from your forked repo shows up in the shared repo's Pull Request list. One way to avoid confusion is to think of the shared repo's PR list as a queue of changes to be applied, pending their review and approval.

Respond to TravisCI tests

The GitHub Pull Request mechanism is designed to allow review and refinement of code prior to its final merge to the shared repo. After creating your Pull Request, the TravisCI tests for monarch-app will be executed automatically, ensuring that the code that 'worked fine' on your development machine also works in the production-like environment provided by TravisCI. The current status of the tests can be found near the bottom of the individual PR page, to the right of the Merge Request symbol:

TBD - Something should be written about developers running tests PRIOR to TravisCI and the the PR. This may already be in the README.md, but should be cited.

Respond to peer review

The GitHub Pull Request mechanism is designed to allow review and refinement of code prior to its final merge to the shared repo. After creating your Pull Request, the TravisCI tests for monarch-app will be executed automatically, ensuring that the code that 'worked fine' on your development machine also works in the production-like environment provided by TravisCI. The current status of the tests can be found

Repushing to a PR branch

It's likely that after created a Pull Request, you will receive useful peer review or your TravisCI tests will have failed. In either case, you will make the required changes on your development machine, retest your changes, and you can then push your new changes back to your task branch and the PR will be automatically updated. This allows a PR to evolve in response to feedback from peers. Once everyone is satisfied, the PR may be merged. (see below).

Merge a pull request

One of the goals behind the workflow described here is to enable a large group of developers to meaningfully contribute to the Monarch codebase. The Pull Request mechanism encourages review and refinement of the proposed code changes. As a matter of informal policy, Monarch expects that a PR will not be merged by its author and that a PR will not be merged without at least one reviewer approving it (via a comment such as +1 in the PR's Comment section).

Celebrate and get back to work

You have successfully gotten your code improvements into the shared repository. Congratulations! The branch you created for this PR is no longer useful, and may be deleted from your forked repo or may be kept. But in no case should the branch be further developed or reused once it has been successfully merge. Subsequent development should be on a new branch. Prepare for your next work by returning to [Refresh and clean up local environment](#refresh-and-clean-up-local-environment).

—

GitHub Tricks and Tips

- Add *?w=1* to a GitHub file compare URL to ignore whitespace differences.

References and Documentation

- The instructions presented here are derived from several sources. However, a very readable and complete article is [Using the Fork-and-Branch Git Workflow](http://blog.scottlowe.org/2015/01/27/using-fork-branch-git-workflow/). Note that the article doesn't make clear that certain steps like Forking are one-time setup steps, after which Branch-PullRequest-Merge steps are used; the instructions below will attempt to clarify this.
- New to GitHub? The [GitHub Guides](http://guides.github.com) are a great place to start.
- Advanced GitHub users might want to check out the [GitHub Cheat Sheet](https://github.com/tiimgreen/github-cheat-sheet/blob/master/README.md)
-

Notes

The process described below is initially intended to be used in the *monarch-app* repository, although it may later be adopted by the other Monarch-related source code repositories, such as *phenogrid*.

4.13.1 CHANGES

0.2.19

- gaf parsing: reject expressions in extensions field that have bad IDs, fixes #99
- lexical mapping: improved handling of xrefs

0.2.18

- lexmap output now excludes index column
- allow custom synsets for lexmap
- fixed bug whereby bulk golr fetch not iterated

0.2.17

- Fixed bug where CHEBI xref labels were treated as class labels
- Various lexical mapping improvements #97 #95

0.2.16

- Added ability to parse skos
- Added more detailed scoring and documentation for lexical mapping.
- lexmap fixes: Fixed #93, #94

0.2.15

- lexical mappings #88 #89
- set ontology id when retrieving from JSON or SPARQL

0.2.11

- #63, added rdf generation
- #62, python version check, @diekhans
- using rst for README
- #56 , assocmodel now allows retrieval of full association objects
- Added GPI writer

0.2.10

- Fixed bug with handling of replaced_by fields in obsolete nodes, #51

0.2.9

- Turned down logging from info to warn for skipped lines

0.2.7

- gaf parsing is more robust to gaf errors
- bugfix function call parameter ordering

0.2.6

- Implementing paging start parameters. For https://github.com/biolink/biolink-api/issues/60

0.2.5

- bugfix for processing gaf lines that do not have the right number of columns

0.2.4

- added ecomap.py
- fixes for planteome

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

A

C

M

N

O

P

Q

R

S

T

U

W

X

Y