Dipper Documentation

Tom Conlin, Kent Shefchek, Tim Putman, Matthew Brush, Lilly Wi

Dec 27, 2021

Contents

Dipper is a Python package to generate RDF triples from common scientific resources. Dipper includes subpackages and modules to create graphical models of this data, including:

- Models package for generating common sets of triples, including common OWL axioms, complex genotypes, associations, evidence and provenance models.
- Graph package for building graphs with RDFLib or streaming n-triples
- Source package containing fetchers and parsers that interface with remote databases and web services

CHAPTER 1

Getting started

Installing, running, and the basics

1.1 Installation

Dipper requires Python version 3.5 or higher

Install with pip:

```
pip install dipper
```

1.1.1 Development version

The development version can be pulled from GitHub.

```
pip3 install git+git://github.com/monarch-initiative/dipper.git
```

1.1.2 Building locally

```
git clone https://github.com/monarch-initiative/dipper.git
cd dipper
pip install .
```

Alternatively, a subset of source specific requirements may be downloaded. To download the core requirements:

```
pip install -r requirements.txt
```

To download source specific requirements use the requirements/ directory, for example:

```
pip install -r requirements/mgi.txt
```

To download requirements for all sources:

```
pip install -r requirements/all-sources.txt
```

1.2 Getting started with Dipper

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

1.2.1 Command line

You can run the code by supplying a list of one or more sources on the command line. some examples:

```
dipper-etl.py --sources impc,hpoa
```

Furthermore, you can check things out by supplying a limit. this will only process the first N number of rows or data elements:

```
dipper-etl.py --sources hpoa --limit 100
```

Other command line parameters are explained if you request help:

```
dipper-etl.py --help
```

1.2.2 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.

1.2.3 Building models

This code example shows some of the basics of building RDF graphs using the model API:

```
import pandas as pd
from dipper.graph.RDFGraph import RDFGraph
from dipper.models.Model import Model

columns = ['variant', 'variant_label', 'variant_type',
           'phenotype','relation', 'source', 'evidence', 'dbxref']

data =  [
     ['ClinVarVariant:254143', 'C326F', 'SO:0000694',
      'HP:0000504','RO:0002200', 'PMID:12503095', 'ECO:0000220',
      'dbSNP:886037891']
]

# Initialize graph and model
graph = RDFGraph()
```

(continues on next page)

(continued from previous page)

```
model = Model(graph)

# Read file
dataframe = pd.DataFrame(data=data, columns=columns)

for index, row in dataframe.iterrows():
    # Add the triple ClinVarVariant:254143 RO:0002200 HP:0000504
    # RO:0002200 is the has_phenotype relation
    # HP:0000748 is the phenotype 'Inappropriate laughter'
    model.addTriple(row['variant'], row['relation'], row['phenotype'])

    # The addLabel method adds a label using the rdfs:label relation
    model.addLabel(row['variant'], row['variant_label'])

    # addType makes the variant an individual of a class,
    # in this case SO:0000694 'SNP'
    model.addType(row['variant'], row['variant_type'])

    # addXref uses the relation OIO:hasDbXref
    model.addXref(row['variant'], row['dbxref'])

    # Serialize the graph as turtle
    print(graph.serialize(format='turtle').decode("utf-8"))
```

For more information see the *Working with the model API* section.

1.3 Notebooks

1.3.1 Jupyter notebook examples

We use Jupyter Notebooks

Available tutorials include:

- Building graphs with the model API
- Querying IMPC evidence and provenance

1.3.2 Running jupyter locally

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

```
pip install jupyter
PYTHONPATH=. jupyter notebook ./docs/notebooks
```

1.4 Downloads

1.4.1 RDF

The dipper output is quality checked and released on a regular basis. The latest release can be found here:

- https://archive.monarchinitiative.org/latest/ttl/

The output from our development branch are made available here (may contain errors):

- https://data.monarchinitiative.org/ttl/

1.4.2 TSV

TSV downloads for common queries can be found here:

- https://archive.monarchinitiative.org/latest/tsv/

1.4.3 Neo4J

A dump of our Neo4J database that includes the output from dipper plus various ontologies:

- https://archive.monarchinitiative.org/latest/scigraph.tgz

A public version can be accessed via the SciGraph REST API:

- https://scigraph-data.monarchinitiative.org/scigraph/docs/

1.5 Ingest status

We use Jenkins to periodically build each source. A dashboard containing the current status of each ingest can be found here:

- http://ci.monarchinitiative.org/view/dipper/

1.6 Applications

1.6.1 BioLink API

A portion of BioLink is driven by the Dipper/SciGraph ETL pipeline:

- https://api.monarchinitiative.org/api/

1.6.2 Monarch Initiative

The Monarch application is powered in part by Dipper:

- https://monarchinitiative.org/

1.6.3 Owlsim

Annotations loaded into Owlsim are from the Dipper/SciGraph pipeline:

- http://owlsim3.monarchinitiative.org/api/docs/

CHAPTER 2

Deeper into Dipper

A look into the structure of the codebase and how to write ingests

2.1 Working with graphs

The Dipper graph package provides two graph implementations, a RDFGraph which is an extension of the RDFLib[1] Graph[2], and a StreamedGraph which prints triples to standard out in the ntriple format.

2.1.1 RDFGraphs

The RDFGraph class reads the curie_map.yaml file and converts strings formatted as curies to RDFLib URIRefs. Triples are added via the addTriple method, for example:

```
from dipper.graph.RDFGraph import RDFGraph

graph = RDFGraph()
graph.addTriple('foaf:John', 'foaf:knows', 'foaf:Joseph')
```

The graph can then be serialized in a variety of formats using the serialize method inherited from the parent RDFLib graph class[3]:

```
from dipper.graph.RDFGraph import RDFGraph

graph = RDFGraph()
graph.addTriple('foaf:John', 'foaf:knows', 'foaf:Joseph')
print(graph.serialize(format='turtle').decode("utf-8"))
```

(continues on next page)

[1] RDFLib: http://rdflib.readthedocs.io/en/stable/
[2] RDFLib Graphs: https://rdflib.readthedocs.io/en/stable/apidocs/rdflib.html#graph-module
[3] RDFLib Serializing: http://rdflib.readthedocs.io/en/stable/apidocs/rdflib.html#rdflib.graph.Graph.serialize

(continued from previous page)

```
# Or write to file
graph.serialize(destination="/path/to/output.ttl", format='turtle')
```

Prints:

```
@prefix OBO: <http://purl.obolibrary.org/obo/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

foaf:John foaf:knows foaf:Joseph .
```

When an object is a literal, set the object_is_literal param to True

```
from dipper.graph.RDFGraph import RDFGraph

graph = RDFGraph()
graph.addTriple('foaf:John', 'rdfs:label', 'John', object_is_literal=True)
```

Literal types can also be passed into the method:

```
from dipper.graph.RDFGraph import RDFGraph

graph = RDFGraph()
graph.addTriple(
    'foaf:John', 'foaf:age', 12,
     object_is_literal=True, literal_type="xsd:integer"
)
```

2.1.2 StreamedGraphs

StreamedGraphs print triples as they are processed by the addTriple method. This is useful for large sources where. The output should be sorted and uniquified as there is no checking for duplicate triples. For example:

```
from dipper.graph.StreamedGraph import StreamedGraph

graph = StreamedGraph()
graph.addTriple('foaf:John', 'foaf:knows', 'foaf:Joseph')
```

Prints:

```
<http://xmlns.com/foaf/0.1/John> <http://xmlns.com/foaf/0.1/knows> <http://xmlns.com/
↪foaf/0.1/Joseph> .
```

2.1.3 References

2.2 Working with the model API

The model package provides classes for building common sets of triples based on our modeling patterns.

For an example see the notebook on this topic: Building graphs with the model API

2.2.1 Basics

The model class provides methods for building common RDF and OWL statements

For a list of methods, see the API docs.

2.2.2 Building associations

We use the RDF Reification[1] pattern to create ternary statements, for example, adding frequency data to phenotype to disease associations. We utilize the Open Biomedical Association ontology[2] to reify statements, and the SEPIO ontology to add evidence and provenance.

For a list of classes and methods, see the API docs.

2.2.3 Building genotypes

We use the GENO ontology[4] to build complex genotypes and their parts.

For a list of methods, see the API docs.

GENO docs: The Genotype Ontology (GENO)

2.2.4 Building complex evidence and provenance graphs

We use the SEPIO ontology to build complex evidence and provenance. For an example see the IMPC source ingest.

For a list of methods, see the API docs for evidence and provenance.

SEPIO docs: The Scientific Evidence and Provenance Information Ontology (SEPIO)

2.2.5 References

2.3 Writing ingests with the source API

2.3.1 Overview

Although not required to write an ingest, we have provided a source parent class that can be extended to leverage reusable functionality in each ingest.

To create a new ingest using this method, first extend the Source class.

If the source contains flat files, include a files dictionary with this structure:

```
files = {
    'somekey': {
        'file': 'filename.tsv',
        'url': 'http://example.org/filename.tsv'
    },
    ...
}
```

[1] RDF Reification: https://www.w3.org/TR/rdf-primer/#reification
[2] OBAN: https://github.com/EBISPOT/OBAN
[4] GENO: https://github.com/monarch-initiative/GENO-ontology

For example:

```
from dipper.sources.Source import Source


class TPO(Source):
"""
The ToxicoPhenomicOmicsDB contains data on ...
"""

files = {
    'genes': {
        'file': 'genes.tsv',
        'url': 'http://example.org/genes.tsv'
    }
}
```

2.3.2 Initializing the class

Each source class takes a graph_type (string) and are_bnodes_skolemized (boolean) parameters. These parameters are used to initialize a graph object in the Source constructor.

Note: In the future this may be adjusted so that a graph object is passed into each source.

For example:

```
def __init__(self, graph_type, are_bnodes_skolemized):
    super().__init__(graph_type, are_bnodes_skolemized, 'TPO')
```

2.3.3 Writing the fetcher

This method is intended to fetch data from the remote locations (if it is newer than the local copy).

Extend the parent fetch function. If a the remote file has already been downloaded. The fetch method checks the remote headers to see if it has been updated. For sources not served over HTTP, this method may need to be overriden, for example in Bgee.

For example:

```
def fetch(self, is_dl_forced=False):
    """
    Fetches files from TPO

    :param is_dl_forced (bool): Force download
    :return None
    """
    self.get_files(is_dl_forced)
```

2.3.4 Writing the parser

Typically these are written by looping through the series of files that were obtained by the fetch method. The goal is to process each file minimally, adding classes and individuals as necessary, and adding triples to the sources' graph.

For example:

```
def parse(self, limit=None):
    """
    Parses genes from TPO

    :param limit (int, optional) limit the number of rows processed
    :return None
    """
    if limit is not None:
        logger.info("Only parsing first %d rows", limit)

    # Open file
    fh = open('/'.join((self.rawdir, self.files['genes']['file'])), 'r')
    # Parse file
    self._add_gene_toxicology(fh, limit)
    # Close file
    fh.close()
```

Considerations when writing a parser

There are certain conventions that we follow when parsing data:

1. Genes are a special case of genomic feature that are added as (OWL) Classes. But all other genomic features are added as individuals of an owl class.

2. If a source references an external identifier then, assume that it has been processed in another source script, and only add the identifier (but not the label) to it within this source's file. This will help prevent label collisions related to slightly different versions of the source data when integrating downstream.

3. You can instantiate a class or individual as many times as you want; they will get merged in the graph and will only show up once in the resulting output.

2.4 Testing ingests

2.4.1 Unit tests

Unit style tests can be achieved by mocking source classes (or specific functions) and testing single functions. The test_graph_equality function can be used to test graph equality by supplying a string formatted as headless (no prefixes) turtle and a graph object. Most dipper methods are not pure functions, and rely on side effects to a graph object. Therefore it is best to clear the graph object before any testing logic, eg:

```
from dipper.utils.TestUtils import TestUtils

source.graph = RDFGraph(True)  # Reset graph
test_util = TestUtils()
source.run_some_function()
expected_triples = """
    foaf:person1 foaf:knows foaf:person2 .
"""
self.assertTrue(self.test_util.test_graph_equality(
    expected_triples, source.graph))
```

2.4.2 Integration tests

Integration tests can be executed by generating a file that contains a subset of a source's data in the same format, and running it through the source.parse() method, serializing the graph, and then testing this file in some other piece of code or database.

You may see testing code within source classes, but these tests will be deleted or refactored and moved to the test directory.

2.5 Configuring dipper with keys and passwords

Add private configuration parameters into your private conf.yaml file. Examples of items to put into the config include:

- database connection parameters (in the "dbauth" object)
- ftp login credentials
- api keys (in the "keys" object)

These are organized such that within any object (dbauth, keys, etc), they are keyed again by the source's name.

Here is an example:: {

"keys": { "omim": "myomimkey1234", "omimftp": "myomimftpkey5678", "ncbi": "myncbikey",

}, "dbauth": {

"mgi": { 'user': 'mymgiuser', 'password': 'mymgipw', 'host': 'mgi-adhoc.jax.org', 'database': 'mgd', 'port': 5432

}

}

}

This file must be placed in the dipper package directory and named conf.yaml. If building locally this is in the dipper/dipper/ directory. If installed with pip this will be in path/to/env/lib/python3.x/site-packages/dipper/ directory.

2.6 Schemas

Although RDF is inherently schemaless, we aim to construct consistent models across sources. This allows us to build source agnostic queries and bridge data across sources.

The dipper schemas are documented as directed graphs. Examples can be found in the ingest artifacts repo.

Some ontologies contain documentation on how to describe data using the classes and properties defined in the ontology:

- The Scientific Evidence and Provenance Information Ontology (SEPIO)
- The Genotype Ontology (GENO)

While not yet implemented, in the future we plan on defining our schemas and constraints using the BioLink model specification.

The cypher queries that we use to cache inferred and direct relationships between entities are stored in GitHub.

CHAPTER 3

For developers

3.1 API Docs

3.1.1 dipper package

Subpackages

dipper.graph package

Submodules

dipper.graph.Graph module

class dipper.graph.Graph.**Graph**
Bases: object

Graph class, used by RDFGraph and StreamedGraph

addTriple(*subject_id*, *predicate_id*, *obj*, *object_is_literal=False*, *literal_type=None*, *subject_category=None*, *object_category=None*)

curie_regexp = re.compile('^[a-zA-Z_]?[a-zA-Z_0-9-]*:[A-Za-z0-9_][A-Za-z0-9_.-]*[A-Za-

serialize(***kwargs*)

skolemizeBlankNode(*curie*)

dipper.graph.RDFGraph module

class dipper.graph.RDFGraph.**RDFGraph**(*are_bnodes_skized=True*, *identifier=None*)
Bases: *dipper.graph.Graph.Graph*, rdflib.graph.ConjunctiveGraph

Extends RDFLibs ConjunctiveGraph The goal of this class is wrap the creation of triples and manage creation of URIRef, Bnodes, and literals from an input curie

addTriple(*subject_id*, *predicate_id*, *obj*, *object_is_literal=None*, *literal_type=None*, *subject_category=None*, *object_category=None*)

bind_all_namespaces()
Results in the RDF @prefix directives for every ingest being added to this ingest.

curie_map = {'': 'https://monarchinitiative.org/', 'APB': 'http://pb.apf.edu.au/phenba

curie_util = <dipper.utils.CurieUtil.CurieUtil object>

fhandle = <_io.TextIOWrapper name='/home/docs/checkouts/readthedocs.org/user_builds/dip

globaltcid = {':activating': 'activating_mutation', ':all_missense_or_inframe': 'all_

globaltt = {'3_prime_UTR_variant': 'SO:0001624', '3prime_overlapping_ncRNA': 'SO:0002

serialize(*destination=None*, *format='turtle'*, *base=None*, *encoding=None*)
Serialize the Graph to destination

If destination is None serialize method returns the serialization as bytes or string.

If encoding is None and destination is None, returns a string If encoding is set, and Destination is None, returns bytes

Format defaults to turtle.

Format support can be extended with plugins, but "xml", "n3", "turtle", "nt", "pretty-xml", "trix", "trig" and "nquads" are built in.

skolemizeBlankNode(*curie*)

dipper.graph.StreamedGraph module

class dipper.graph.StreamedGraph.**StreamedGraph**(*are_bnodes_skized=True*, *identifier=None*, *file_handle=None*, *fmt='nt'*)

Bases: *dipper.graph.Graph.Graph*

Stream rdf triples to file or stdout Assumes a downstream process will sort then uniquify triples

Theoretically could support both ntriple, rdfxml formats, for now just support nt

addTriple(*subject_id*, *predicate_id*, *obj*, *object_is_literal=None*, *literal_type=None*, *subject_category=None*, *object_category=None*)

curie_map = {'': 'https://monarchinitiative.org/', 'APB': 'http://pb.apf.edu.au/phenba

curie_util = <dipper.utils.CurieUtil.CurieUtil object>

fhandle = <_io.TextIOWrapper name='/home/docs/checkouts/readthedocs.org/user_builds/dip

globaltcid = {':activating': 'activating_mutation', ':all_missense_or_inframe': 'all_

globaltt = {'3_prime_UTR_variant': 'SO:0001624', '3prime_overlapping_ncRNA': 'SO:0002

serialize(*subject_iri*, *predicate_iri*, *obj*, *object_is_literal=False*, *literal_type=None*, *subject_category_iri=None*, *predicate_category_iri='biolink:category'*, *object_category_iri=None*)

skolemizeBlankNode(*curie*)

dipper.models package

Subpackages

dipper.models.assoc package

Submodules

dipper.models.assoc.Association module

class dipper.models.assoc.Association.**Assoc**(*graph*, *definedby*, *sub=None*, *obj=None*, *pred=None*, *subject_category=None*, *object_category=None*)

Bases: object

A base class for OBAN (Monarch)-style associations, to enable attribution of source and evidence on statements.

add_association_to_graph(*association_category=None*)

add_date(*date*)

add_evidence(*identifier*)
Add an evidence code to the association object (maintained as a list) :param identifier:

Returns

add_predicate_object(*predicate*, *object_node*, *object_type=None*, *datatype=None*)

add_provenance(*identifier*)

add_source(*identifier*)
Add a source identifier (such as publication id) to the association object (maintained as a list) TODO we need to greatly expand this function!

Parameters **identifier** –

Returns

get_association_id()

static make_association_id(*definedby*, *sub*, *pred*, *obj*, *attributes=None*)
A method to create unique identifiers for OBAN-style associations, based on all the parts of the association If any of the items is empty or None, it will convert it to blank. It effectively digests the string of concatonated values. Subclasses of Assoc can submit an additional array of attributes that will be appeded to the ID.

Note this is equivalent to a RDF blank node

Parameters

- **definedby** – The (data) resource that provided the annotation
- **subject** –
- **predicate** –
- **object** –
- **attributes** –

Returns

set_association_id(*assoc_id=None*)
This will set the association ID based on the internal parts of the association. To be used in cases where an external association identifier should be used.

Parameters **assoc_id** –

Returns

set_description(*description*)

set_object(*identifier*)

set_relationship(*identifier*)

set_score(*score*, *unit=None*, *score_type=None*)

set_subject(*identifier*)

dipper.models.assoc.Chem2DiseaseAssoc module

class dipper.models.assoc.Chem2DiseaseAssoc.**Chem2DiseaseAssoc**(*graph*, *definedby*, *chem_id*, *phenotype_id*, *rel_id=None*)
Bases: *dipper.models.assoc.Association.Assoc*

Attributes: assoc_id (str): Association Curie (Prefix:ID) chem_id (str): Chemical Curie phenotype_id (str): Phenotype Curie pub_list (str,list): One or more publication curies rel (str): Property relating assoc_id and chem_id evidence (str): Evidence curie

make_c2p_assoc_id()

set_association_id(*assoc_id=None*)
This will set the association ID based on the internal parts of the association. To be used in cases where an external association identifier should be used.

Parameters **assoc_id** –

Returns

dipper.models.assoc.D2PAssoc module

class dipper.models.assoc.D2PAssoc.**D2PAssoc**(*graph*, *definedby*, *disease_id*, *phenotype_id*, *onset=None*, *frequency=None*, *rel=None*, *disease_category=None*, *phenotype_category=None*)
Bases: *dipper.models.assoc.Association.Assoc*

A specific association class for defining Disease-to-Phenotype relationships This assumes that a graph is created outside of this class, and nodes get added. By default, an association will assume the "has_phenotype" relationship, unless otherwise specified.

add_association_to_graph(*association_category=None*)
The reified relationship between a disease and a phenotype is decorated with some provenance information. This makes the assumption that both the disease and phenotype are classes.

Parameters

- **g** –
- **disease_category** – a biolink category CURIE for disease_id (defaults to

biolink:Disease via the constructor) :param phenotype_category: a biolink category CURIE for phenotype_id (defaults to biolink:PhenotypicFeature via the constructor) :return:

make_d2p_id()
Make an association id for phenotypic associations with disease that is defined by: source of association + disease + relationship + phenotype + onset + frequency

Returns

set_association_id(*assoc_id=None*)
This will set the association ID based on the internal parts of the association. To be used in cases where an external association identifier should be used.

Parameters **assoc_id** –

Returns

dipper.models.assoc.G2PAssoc module

class dipper.models.assoc.G2PAssoc.**G2PAssoc**(*graph*, *definedby*, *entity_id*, *phenotype_id*, *rel=None*, *entity_category=None*, *phenotype_category=None*)
Bases: *dipper.models.assoc.Association.Assoc*

A specific association class for defining Genotype-to-Phenotype relationships. This assumes that a graph is created outside of this class, and nodes get added. By default, an association will assume the "has_phenotype" relationship, unless otherwise specified. Note that genotypes are expected to be created and defined outside of this association, most likely by calling methods in the Genotype() class.

add_association_to_graph(*entity_category=None*, *phenotype_category=None*)
Overrides Association by including bnode support

The reified relationship between a genotype (or any genotype part) and a phenotype is decorated with some provenance information. This makes the assumption that both the genotype and phenotype are classes.

currently hardcoded to map the annotation to the monarch namespace :param g: :param entity_category: a biolink category CURIE for self.sub :param phenotype_category: a biolink category CURIE for self.obj :return:

make_g2p_id()
Make an association id for phenotypic associations that is defined by: source of association + (Annot subject) + relationship + phenotype/disease + environment + start stage + end stage

Returns

set_association_id(*assoc_id=None*)
This will set the association ID based on the internal parts of the association. To be used in cases where an external association identifier should be used.

Parameters **assoc_id** –

Returns

set_environment(*environment_id*)

set_stage(*start_stage_id*, *end_stage_id*)

dipper.models.assoc.InteractionAssoc module

class dipper.models.assoc.InteractionAssoc.**InteractionAssoc**(*graph*, *definedby*, *subj*, *obj*, *rel=None*)

Bases: *dipper.models.assoc.Association.Assoc*

dipper.models.assoc.OrthologyAssoc module

class dipper.models.assoc.OrthologyAssoc.**OrthologyAssoc**(*graph*, *definedby*, *gene1*, *gene2*, *rel=None*, *subject_category=None*, *object_category=None*)

Bases: *dipper.models.assoc.Association.Assoc*

add_gene_family_to_graph(*family_id*)

Make an association between a group of genes and some grouping class. We make the assumption that the genes in the association are part of the supplied family_id, and that the genes have already been declared as classes elsewhere. The family_id is added as an individual of type DATA:gene_family.

Triples: <family_id> a EDAM-DATA:gene_family <family_id> RO:has_member <gene1> <family_id> RO:has_member <gene2> <gene1> biolink:category <subject_category> <gene2> biolink:category <object_category> :param family_id: :param g: the graph to modify :return:

Submodules

dipper.models.BiolinkVocabulary module

class dipper.models.BiolinkVocabulary.**BioLinkVocabulary**

Bases: object

bl_file_with_path = '/home/docs/checkouts/readthedocs.org/user_builds/dipper/checkouts

bl_vocab = {'curie_prefix': 'biolink', 'terms': ['AnatomicalEntity', 'Association',

key = 'Zygosity'

terms = {'AnatomicalEntity': 'biolink:AnatomicalEntity', 'Association': 'biolink:Ass

yaml_file = <_io.TextIOWrapper name='/home/docs/checkouts/readthedocs.org/user_builds/

dipper.models.ClinVarRecord module

https://www.ncbi.nlm.nih.gov/clinvar/docs/details/ Object mapping to XML schema

class dipper.models.ClinVarRecord.**Allele**(*id: str*, *label: Optional[str] = None*, *variant_type: Optional[str] = None*, *genes: Optional[List[dipper.models.ClinVarRecord.Gene]] = None*, *synonyms: Optional[List[str]] = None*, *dbsnps: Optional[List[str]] = None*)

Bases: object

ClinVar Allele Alleles can have 0 to many genes

These are called alleles and variants on the ClinVar UI, and variant, single nucleotide variant, etc in the XML

id: allele id label: label variant_type: single nucleotide variant genes: gene(s) in which the variant is found synonyms: eg HGVC dbsnp: dbSNP curies

class dipper.models.ClinVarRecord.**ClinVarRecord**(*id: str, accession: str, created: str, updated: str, genovar: dipper.models.ClinVarRecord.Genovar, significance: str, conditions: Optional[List[dipper.models.ClinVarRecord.Condition]] = None*)

Bases: object

Reference ClinVar Record (RCV) id: RCV id accession: RCV accession (eg RCV000123456) created: Created date updated: Updated date genovar: the variant or genotype associated with the condition(s) significance: clinical significance (eg benign, pathogenic) condition: The condition(s) for which this allele set was interpreted, with

links to databases with defining information about that condition.

class dipper.models.ClinVarRecord.**Condition**(*id: str, label: Optional[str] = None, database: Optional[str] = None, medgen_id: Optional[str] = None*)

Bases: object

ClinVar condition

class dipper.models.ClinVarRecord.**Gene**(*id: Union[str, int, None], association_to_allele: str*)

Bases: object

ClinVar Gene Intentionally leaves out label/symbol, this should come from HGNC

class dipper.models.ClinVarRecord.**Genotype**(*id: str, label: Optional[str] = None, variants: Optional[List[dipper.models.ClinVarRecord.Variant]] = None, variant_type: Optional[str] = None*)

Bases: *dipper.models.ClinVarRecord.Genovar*

ClinVar genotype Example: Compound Heterozygote, Diplotype

These are called variants on the ClinVar UI, and a GenotypeSet in the XML

class dipper.models.ClinVarRecord.**Genovar**(*id: str, label: Optional[str] = None, variant_type: Optional[str] = None*)

Bases: object

Sequence feature entity that is linked to a disease in an Reference ClinVar Record, it is either a Variant or Genotype

class dipper.models.ClinVarRecord.**Variant**(*id: str, label: Optional[str] = None, alleles: Optional[List[dipper.models.ClinVarRecord.Allele]] = None, variant_type: Optional[str] = None*)

Bases: *dipper.models.ClinVarRecord.Genovar*

ClinVar variant, variants can have one or more alleles

These are called variants and alleles on the ClinVar UI, and Variants in the XML

dipper.models.Dataset module

Produces metadata about ingested data

class dipper.models.Dataset.**Dataset**(*identifier*, *data_release_version*, *ingest_name*, *ingest_title*, *ingest_url*, *ingest_logo=None*, *ingest_description=None*, *license_url=None*, *data_rights=None*, *graph_type='rdf_graph'*, *file_handle=None*, *distribution_type='ttl'*, *dataset_curie_prefix='MonarchArchive'*)

Bases: object

This class produces metadata about a dataset that is compliant with the HCLS dataset specification: https://www.w3.org/TR/2015/NOTE-hcls-dataset-20150514/#s4_4

Summary level: The summary level provides a description of a dataset that is independent of a specific version or format. (e.g. the Monarch ingest of CTD) CURIE for this is something like MonarchData:[SOURCE IDENTIFIER]

Version level: The version level captures version-specific characteristics of a dataset. (e.g. the 01-02-2018 ingest of CTD) CURIE for this is something like MonarchData:[SOURCE IDENTIFIER_INGESTTIMESTAMP]

Distribution level: The distribution level captures metadata about a specific form and version of a dataset (e.g. turtle file for 01-02-2018 ingest of CTD). There is a [distribution level resource] for each different downloadable file we emit, i.e. one for the TTL file, one for the ntriples file, etc. CURIE for this is like MonarchData:[SOURCE IDENTIFIER_INGESTTIMESTAMP].ttl or MonarchData:[SOURCE IDENTIFIER_INGESTTIMESTAMP].nt or MonarchData:[SOURCE IDENTIFIER_INGESTTIMESTAMP].[whatever file format]

We write out at least the following triples:

SUMMARY LEVEL TRIPLES: [summary level resource] - rdf:type -> dctypes:Dataset [summary level resource] - dc:title -> title (literal) [summary level resource] - dc:description -> description (literal)

(use docstring from Source class)

[summary level resource] - dc:source -> [source web page, e.g. omim.org] [summary level resource] - schema:logo -> [source logo IRI] [summary level resource] - dc:publisher -> monarchinitiative.org

n.b: about summary level resource triples: – HCLS spec says we "should" link to our logo and web page, but I'm not, because it would confuse the issue of whether we are pointing to our logo/page or the logo/page of the data source for this ingest. Same below for [version level resource] and [distibution level resource] - I'm not linking to our page/logo down there either. - spec says we "should" include summary level triples describing Update frequency and SPARQL endpoint but I'm omitting this for now, because these are not clearly defined at the moment

VERSION LEVEL TRIPLES: [version level resource] - rdf:type -> dctypes:Dataset [version level resource] - dc:title -> version title (literal) [version level resource] - dc:description -> version description (literal) [version level resource] - dc:created -> ingest timestamp [ISO 8601 compliant] [version level resource] - pav:version -> ingest timestamp (same one above) [version level resource] - dc:creator -> monarchinitiative.org [version level resource] - dc:publisher -> monarchinitiative.org [version level resource] - dc:isVersionOf -> [summary level resource] [version level resource] - dc:source -> [source file 1 IRI] [version level resource] - dc:source -> [source file 2 IRI] . . .

[source file 1 IRI] - pav:retrievedOn -> [download date timestamp] [source file 2 IRI] - pav:version -> [source version (if set, optional)] [source file 2 IRI] - pav:retrievedOn -> [download date timestamp] [source file 2 IRI] - pav:version -> [source version (if set, optional)] . . .

[version level resource] - pav:createdWith -> [Dipper github URI] [version level resource] - void:dataset -> [distribution level resource]

[version level resource] - cito:citesAsAuthoriy -> [citation id 1] [version level resource] - cito:citesAsAuthoriy -> [citation id 2] [version level resource] - cito:citesAsAuthoriy -> [citation id 3]

n.b: about version level resource triples: - spec says we "should" include Date of issue/dc:issued triple, but I'm not because it is redundant with this triple above: [version level resource] - dc:created -> time stamp and would introduce ambiguity and confusion if the two disagree. Same below for [distribution level resource] - dc:created -> tgiime stamp below Also omitting:

- triples linking to our logo and page, see above.
- License/dc:license triple, because we will make this triple via the [distribution level resource] below
- Language/dc:language triple b/c it seems superfluous. Same below for [distribution level resource] - no language triple.
- [version level resource] - pav:version triple is also a bit redundant

with the pav:version triple below, but the spec requires both these triples - I'm omitting the [version level resource] -> pav:previousVersion because Dipper doesn't know this info for certain at run time. Same below for [distribution level resource] - pav:previousVersion.

DISTRIBUTION LEVEL TRIPLES: [distribution level resource] - rdf:type -> dctypes:Dataset [distribution level resource] - rdf:type -> dcat:Distribution [distribution level resource] - dc:title -> distribution title (literal) [distribution level resource] - dc:description -> distribution description (lit.) [distribution level resource] - dc:created -> ingest timestamp[ISO 8601 compliant] [distribution level resource] - pav:version -> ingest timestamp (same as above) [distribution level resource] - dc:creator -> monarchinitiative.org [distribution level resource] - dc:publisher -> monarchinitiative.org [distribution level resource] - dc:license -> [license info, if available

otherwise indicate unknown]

[distribution level resource] - dc:rights -> [data rights IRI] [distribution level resource] - pav:createdWith -> [Dipper github URI] [distribution level resource] - dc:format -> [IRI of ttl|nt|whatever spec] [distribution level resource] - dcat:downloadURL -> [ttl|nt URI] [distribution level resource] - void:triples -> [triples count (literal)] [distribution level resource] - void:entities -> [entities count (literal)] [distribution level resource] - void:distinctSubjects -> [subject count (literal)] [distribution level resource] - void:distinctObjects -> [object count (literal)] [distribution level resource] - void:properties -> [properties count (literal)] ...

n.b: about distribution level resource triples: - omitting Vocabularies used/void:vocabulary and Standards used/dc:conformTo triples, because they are described in the ttl file - also omitting Example identifier/idot:exampleIdentifier and Example resource/void:exampleResource, because we don't really have one canonical example of either - they're all very different. - [distribution level resource] - dc:created should have the exact same time stamp as this triple above: [version level resource] - dc:created -> time stamp - this [distribution level resource] - pav:version triple should have the same object as [version level resource] - pav:version triple above - Data source provenance/dc:source triples are above in the [version level resource] - omitting Byte size/dc:byteSize, RDF File URL/void:dataDump, and Linkset/void:subset triples because they probably aren't necessary for MI right now - these triples "should" be emitted, but we will do this in a later iteration: # of classes void:classPartition IRI # of literals void:classPartition IRI # of RDF graphs void:classPartition IRI

Note: Do not use blank nodes in the dataset graph. This dataset graph is added to the main Dipper graph in Source.write() like so

$ mainGraph = mainGraph + datasetGraph

which apparently in theory could lead to blank node ID collisions between the two graphs.

Note also that this implementation currently does not support producing metadata for StreamedGraph graphs (see dipper/graph/StreamedGraph.py). StreamedGraph is currently not being used for any ingests, so this isn't

a problem. There was talk of using StreamedGraph for a rewrite/refactor of the Clinvar ingest, which would probably require adding support here for StreamedGraph's.

get_graph()
This method returns the dataset graph :param :return: dataset graph

get_license()
This method returns the license info :param :return: license info

static hash_id(*word*)
Given a string, make a hash Duplicated from Source.py.

Parameters **word** – str string to be hashed

Returns hash of id

static make_id(*long_string*, *prefix='MONARCH'*)
A method to create DETERMINISTIC identifiers based on a string's digest. currently implemented with sha1 Duplicated from Source.py to avoid circular imports. :param long_string: string to use to generate identifier :param prefix: prefix to prepend to identifier [Monarch] :return: a Monarch identifier

set_citation(*citation_id*)
This method adds [citaton_id] argument to the set of citations, and also adds a triple indicating that version level cito:citesAsAuthority [citation_id] :param: citation_id :return: none

set_ingest_source(*url*, *predicate=None*, *is_object_literal=False*)
This method writes a triple to the dataset graph indicating that the ingest used a file or resource at [url] during the ingest.

Triple emitted is version_level_curie dc:source [url]

This triple is likely to be redundant if Source.get_files() is used to retrieve the remote files/resources, since this triple should also be emitted as files/resources are being retrieved. This method is provided as a convenience method for sources that do their own downloading of files.

Parameters

- **url** – a remote resource used as a source during ingest
- **predicate** – the predicate to use for the triple ["dc:source"] from spec (https://www.w3.org/TR/2015/NOTE-hcls-dataset-20150514/) "Use dc:source when the source dataset was used in whole or in part. Use pav:retrievedFrom when the source dataset was used in whole and was not modified from its original distribution. Use prov:wasDerivedFrom when the source dataset was in whole or in part and was modified from its original distribution."

Returns None

set_ingest_source_file_version_date(*file_iri*, *date*, *datatype=rdflib.term.URIRef('http://www.w3.org/2001/XMLS*
This method sets the version that the source (OMIM, CTD, whatever) uses to refer to this version of the remote file/resource that was used in the ingest

It writes this triple:

file_iri - 'pav:version' -> date or timestamp

Version is added as a literal of datatype XSD date

Note: if file_iri was retrieved using get_files(), then the following triple was created and you might not need this method:

file_iri - 'pav:retrievedOn' -> download date

Parameters

- **file_iri** – a remote file or resource used in ingest
- **date** – a date in YYYYMMDD format that the source (OMIM, CTD). You can

add timestamp as a version by using a different datatype (below) :param datatype: an XSD literal datatype, default is XSD.date uses to refer to this version of the file/resource used during the ingest :return: None

set_ingest_source_file_version_num(*file_iri*, *version*)

This method sets the version of a remote file or resource that is used in the ingest. It writes this triple:

file_iri - 'pav:version' -> version

Version is an untyped literal

Note: if your version is a date or timestamp, use set_ingest_source_file_version_date() instead

Parameters

- **file_iri** – a remote file or resource used in ingest
- **version** – a number or string (e.g. v1.2.3) that the source (OMIM, CTD)

uses to refer to this version of the file/resource used during the ingest :return: None

set_ingest_source_file_version_retrieved_on(*file_iri*, *date*, *datatype=rdflib.term.URIRef('http://www.w3.org/2001/XMLSch*

This method sets the date on which a remote file/resource (from OMIM, CTD, etc) was retrieved.

It writes this triple:

file_iri - 'pav:retrievedOn' -> date or timestamp

Version is added as a literal of datatype XSD date by default

Note: if file_iri was retrieved using get_files(), then the following triple was created and you might not need this method:

file_iri - 'pav:retrievedOn' -> download date

Parameters

- **file_iri** – a remote file or resource used in ingest
- **date** – a date in YYYYMMDD format that the source (OMIM, CTD). You can

add timestamp as a version by using a different datatype (below) :param datatype: an XSD literal datatype, default is XSD.date uses to refer to this version of the file/resource used during the ingest :return: None

dipper.models.Environment module

class dipper.models.Environment.**Environment**(*graph*)

Bases: object

These methods provide convenient methods to add items related to an experimental environment and it's parts to a supplied graph.

This is a stub.

addComponentAttributes(*component_id*, *entity_id*, *value=None*, *unit=None*, *component_category=None*, *entity_category=None*)

addComponentToEnvironment(*env_id*, *component_id*, *environment_category=None*, *component_category=None*)

addEnvironment(*env_id*, *env_label*, *env_type=None*, *env_description=None*)

addEnvironmentalCondition(*cond_id*, *cond_label*, *cond_type=None*, *cond_description=None*, *condition_category=None*)

dipper.models.Evidence module

class dipper.models.Evidence.**Evidence**(*graph*, *association*)
Bases: object

To model evidence as the basis for an association. This encompasses:

- **measurements taken from the lab, and their significance.** these can be derived from papers or other agents.
- papers

>1 measurement may result from an assay, each of which may have it's own significance

add_data_individual(*data_curie*, *label=None*, *ind_type=None*, *data_curie_category=None*)
Add data individual :param data_curie: str either curie formatted or long string,

long strings will be converted to bnodes

Parameters

- **data_curie_category** – a biolink category CURIE for data_curie
- **type** – str curie
- **label** – str

Returns None

add_evidence(*evidence_line*, *evidence_type=None*, *label=None*)
Add line of evidence node to association id

Parameters

- **evidence_line** – curie or iri, evidence line
- **evidence_type** – curie or iri, evidence type if available

Returns None

add_source(*evidence_line*, *source*, *label=None*, *src_type=None*, *source_category=None*)
Applies the triples: <evidence> <dc:source> <source> <source> <rdf:type> <type> <source> <rdfs:label> "label"

TODO this should belong in a higher level class :param evidence_line: str curie :param source: str source as curie :param label: optional, str type as curie :param type: optional, str type as curie :return: None

add_supporting_data(*evidence_line*, *measurement_dict*)
Add supporting data :param evidence_line: :param data_object: dict, where keys are curies or iris and values are measurement values for example:

{ "_:1234" : "1.53E07" "_:4567": "20.25"

}

Note: assumes measurements are RDF:Type 'ed elsewhere :return: None

add_supporting_evidence(*evidence_line*, *evidence_type=None*, *label=None*)
Add supporting line of evidence node to association id

Parameters

- **evidence_line** – curie or iri, evidence line
- **evidence_type** – curie or iri, evidence type if available

Returns None

add_supporting_publication(*evidence_line*, *publication*, *label=None*, *pub_type=None*)
<evidence> <has_supporting_reference> <source> <source> <rdf:type> <type> <source> <rdfs:label> "label" :param evidence_line: str curie :param publication: str curie :param label: optional, str type as curie :param type: optional, str type as curie :return:

dipper.models.Family module

class dipper.models.Family.**Family**(*graph*)
Bases: object

Model mereological/part whole relationships

Although these relations are more abstract, we often use them to model family relationships (proteins, humans, etc.) The naming of this class may change in the future to better reflect the meaning of the relations it is modeling

addMember(*group_id*, *member_id*, *group_category=None*, *member_category=None*)

addMemberOf(*member_id*, *group_id*, *member_category=None*, *group_category=None*)

dipper.models.GenomicFeature module

class dipper.models.GenomicFeature.**Feature**(*graph*, *feature_id=None*, *label=None*, *feature_type=None*, *description=None*, *feature_category=None*)
Bases: object

Dealing with genomic features here. By default they are all faldo:Regions. We use SO for typing genomic features. At the moment, RO:has_subsequence is the default relationship between the regions, but this should be tested/verified.

TODO: the graph additions are in the addXToFeature functions, but should be separated. TODO: this will need to be extended to properly deal with fuzzy positions in faldo.

addFeatureEndLocation(*coordinate*, *reference_id*, *strand=None*, *position_types=None*)
Adds the coordinate details for the end of this feature :param coordinate: :param reference_id: :param strand:

addFeatureProperty(*property_type*, *feature_property*)

addFeatureStartLocation(*coordinate*, *reference_id*, *strand=None*, *position_types=None*)
Adds coordinate details for the start of this feature. :param coordinate: :param reference_id: :param strand: :param position_types:

addFeatureToGraph(*add_region=True*, *region_id=None*, *feature_as_class=False*, *feature_category=None*)
We make the assumption here that all features are instances. The features are located on a region, which begins and ends with faldo:Position The feature locations leverage the Faldo model, which has a general structure like: Triples: feature_id a feature_type (individual) faldo:location region_id region_id a faldo:region faldo:begin start_position faldo:end end_position start_position a (any of: faldo:(((Both|Plus|Minus)Strand)|Exact)Position) faldo:position Integer(numeric position) faldo:reference

reference_id end_position a (any of: faldo:(((Both|Plus|Minus)Strand)|Exact)Position) faldo:position Integer(numeric position) faldo:reference reference_id

:param add_region [True] :param region_id [None] :param feature_as_class [False] :param feature_category: a biolink category CURIE for feature

addPositionToGraph(*reference_id*, *position*, *position_types=None*, *strand=None*)
Add the positional information to the graph, following the faldo model. We assume that if the strand is None, we give it a generic "Position" only. Triples: my_position a (any of: faldo:(((Both|Plus|Minus)Strand)|Exact)Position) faldo:position Integer(numeric position) faldo:reference reference_id

Parameters

- **graph** –
- **reference_id** –
- **position** –
- **position_types** –
- **strand** –

Returns Identifier of the position created

addRegionPositionToGraph(*region_id*, *begin_position_id*, *end_position_id*)

addSubsequenceOfFeature(*parentid*, *subject_category=None*, *object_category=None*)
This will add reciprocal triples like: feature <is subsequence of> parent parent has_subsequence feature :param graph: :param parentid:

Returns

addTaxonToFeature(*taxonid*)
Given the taxon id, this will add the following triple: feature in_taxon taxonid :param graph: :param taxonid: :return:

dipper.models.GenomicFeature.**makeChromID**(*chrom*, *reference=None*, *prefix=None*)
This will take a chromosome number and a NCBI taxon number, and create a unique identifier for the chromosome. These identifiers are made in the @base space like: Homo sapiens (9606) chr1 ==> :9606chr1 Mus musculus (10090) chrX ==> :10090chrX

Parameters

- **chrom** – the chromosome (preferably without any chr prefix)
- **reference** – the numeric portion of the taxon id

Returns

dipper.models.GenomicFeature.**makeChromLabel**(*chrom*, *reference=None*)

dipper.models.Genotype module

class dipper.models.Genotype.**Genotype**(*graph*)
Bases: object

These methods provide convenient methods to add items related to a genotype and it's parts to a supplied graph. They follow the patterns set out in GENO https://github.com/monarch-initiative/GENO-ontology. For specific sequence features, we use the GenomicFeature class to create them.

addAffectedLocus(*allele_id*, *gene_id*, *rel_id=None*)
We make the assumption here that if the relationship is not provided, it is a GENO:has_affected_feature.

Here, the allele should be a variant_locus, not a sequence alteration. :param allele_id: :param gene_id: :param rel_id: :return:

addAllele(*allele_id*, *allele_label*, *allele_type=None*, *allele_description=None*)
Make an allele object. If no allele_type is added, it will default to a geno:allele :param allele_id: curie for allele (required) :param allele_label: label for allele (required) :param allele_type: id for an allele type (optional, recommended SO or GENO class) :param allele_description: a free-text description of the allele :return:

addAlleleOfGene(*allele_id*, *gene_id*, *rel_id=None*)
We make the assumption here that if the relationship is not provided, it is a GENO:is_allele_of.

Here, the allele should be a variant_locus, not a sequence alteration. :param allele_id: :param gene_id: :param rel_id: :return:

addChromosome(*chrom*, *tax_id*, *tax_label=None*, *build_id=None*, *build_label=None*)
if it's just the chromosome, add it as an instance of a SO:chromosome, and add it to the genome. If a build is included, punn the chromosome as a subclass of SO:chromsome, and make the build-specific chromosome an instance of the supplied chr. The chr then becomes part of the build or genome.

addChromosomeClass(*chrom_num*, *taxon_id*, *taxon_label*)

addChromosomeInstance(*chr_num*, *reference_id*, *reference_label*, *chr_type=None*)
Add the supplied chromosome as an instance within the given reference :param chr_num: :param reference_id: for example, a build id like UCSC:hg19 :param reference_label: :param chr_type: this is the class that this is an instance of. typically a genome-specific chr

Returns

addConstruct(*construct_id*, *construct_label*, *construct_type=None*, *construct_description=None*, *construct_category=None*, *construct_type_category=None*)

Parameters

- **construct_id** –
- **construct_label** –
- **construct_type** –
- **construct_description** –
- **construct_category** – a biolink category CURIE for construct_id
- **construct_type_category** – a biolink category CURIE for construct_type

Returns

addDerivesFrom(*child_id*, *parent_id*, *child_category=None*, *parent_category=None*)
We add a derives_from relationship between the child and parent id. Examples of uses include between: an allele and a construct or strain here, a cell line and it's parent genotype. Adding the parent and child to the graph should happen outside of this function call to ensure graph integrity. :param child_id: :param parent_id: :return:

addGene(*gene_id*, *gene_label=None*, *gene_type=None*, *gene_description=None*)
genes are classes

addGeneProduct(*sequence_id*, *product_id*, *product_label=None*, *product_type=None*, *sequence_category=None*, *product_category=None*)
Add gene/variant/allele has_gene_product relationship Can be used to either describe a gene to transcript relationship or gene to protein :param sequence_id: :param product_id: :param product_label: :param

product_type: :param sequence_category: bl category CURIE for seq_id [blv.terms.Gene].value :param product_category: biolink category CURIE for product_id :return:

addGeneTargetingReagent(*reagent_id*, *reagent_label*, *reagent_type*, *gene_id*, *description=None*, *reagent_category=None*)

Here, a gene-targeting reagent is added. The actual targets of this reagent should be added separately. :param reagent_id: :param reagent_label: :param reagent_type:

Returns

addGeneTargetingReagentToGenotype(*reagent_id*, *genotype_id*)

Add genotype has_variant_part reagent_id. For example, add a morphant reagent thingy to the genotype, assuming it's a extrinsic_genotype Also a triple to assign biolink categories to genotype and reagent. :param reagent_id :param genotype_id :return:

addGenome(*taxon_num*, *taxon_label=None*, *genome_id=None*)

addGenomicBackground(*background_id*, *background_label*, *background_type=None*, *background_description=None*)

addGenomicBackgroundToGenotype(*background_id*, *genotype_id*, *background_type=None*)

addGenotype(*genotype_id*, *genotype_label*, *genotype_type=None*, *genotype_description=None*)

If a genotype_type is not supplied, we will default to 'intrinsic genotype' :param genotype_id: :param genotype_label: :param genotype_type: :param genotype_description: :return:

addMemberOfPopulation(*member_id*, *population_id*)

addParts(*part_id*, *parent_id*, *part_relationship=None*, *part_category=None*, *parent_category=None*)

This will add a has_part (or subproperty) relationship between a parent_id and the supplied part. By default the relationship will be BFO:has_part, but any relationship could be given here. :param part_id: :param parent_id: :param part_relationship: :param part_category: a biolink vocab curie for part_id :param parent_category: a biolink vocab curie for parent_id :return:

addPartsToVSLC(*vslc_id*, *allele1_id*, *allele2_id*, *zygosity_id=None*, *allele1_rel=None*, *allele2_rel=None*)

Here we add the parts to the VSLC. While traditionally alleles (reference or variant loci) are traditionally added, you can add any node (such as sequence_alterations for unlocated variations) to a vslc if they are known to be paired. However, if a sequence_alteration's loci is unknown, it probably should be added directly to the GVC. :param vslc_id: :param allele1_id: :param allele2_id: :param zygosity_id: :param allele1_rel: :param allele2_rel: :return:

addPolypeptide(*polypeptide_id*, *polypeptide_label=None*, *transcript_id=None*, *polypeptide_type=None*)

Parameters

- **polypeptide_id** –
- **polypeptide_label** –
- **polypeptide_type** –
- **transcript_id** –

Returns

addReagentTargetedGene(*reagent_id*, *gene_id*, *targeted_gene_id=None*, *targeted_gene_label=None*, *description=None*, *reagent_category=None*)

This will create the instance of a gene that is targeted by a molecular reagent (such as a morpholino or rnai). If an instance id is not supplied, we will create it as an anonymous individual which is of the type GENO:reagent_targeted_gene. We will also add the targets relationship between the reagent and gene class.

<targeted_gene_id> a GENO:reagent_targeted_gene rdfs:label targeted_gene_label dc:description description <reagent_id> GENO:targets_gene <gene_id>

Parameters

- **reagent_id** –
- **gene_id** –
- **targeted_gene_id** –
- **reagent_category** – a biolink category CURIE for reagent_id

Returns

addReferenceGenome(*build_id*, *build_label*, *taxon_id*)

addSequenceAlteration(*sa_id*, *sa_label*, *sa_type=None*, *sa_description=None*)

addSequenceAlterationToVariantLocus(*sa_id*, *vl_id*)

addSequenceDerivesFrom(*child_id*, *parent_id*, *child_category=None*, *parent_category=None*)

addTargetedGeneComplement(*tgc_id*, *tgc_label*, *tgc_type=None*, *tgc_description=None*)

addTargetedGeneSubregion(*tgs_id*, *tgs_label*, *tgs_type=None*, *tgs_description=None*)

addTaxon(*taxon_id*, *genopart_id*, *genopart_category=None*)
The supplied geno part will have the specified taxon added with RO:in_taxon relation. Generally the taxon is associated with a genomic_background, but could be added to any genotype part (including a gene, regulatory element, or sequence alteration). :param taxon_id: :param genopart_id: :param genopart_category: a biolink term for genopart_id :return:

addVSLCtoParent(*vslc_id*, *parent_id*, *part_category=None*, *parent_category=None*)
The VSLC can either be added to a genotype or to a GVC. The vslc is added as a part of the parent. :param vslc_id: :param parent_id: :param part_category: a biolink category CURIE for part :param parent_category: a biolink category CURIE for parent :return:

static makeGenomeID(*taxon_id*)

make_experimental_model_with_genotype(*genotype_id*, *genotype_label*, *taxon_id*, *taxon_label*)

static make_variant_locus_label(*gene_label*, *allele_label*)

make_vslc_label(*gene_label*, *allele1_label*, *allele2_label*)
Make a Variant Single Locus Complement (VSLC) in monarch-style. :param gene_label: :param allele1_label: :param allele2_label: :return:

dipper.models.Model module

class dipper.models.Model.**Model**(*graph*)
Bases: object

Utility class to add common triples to a graph (subClassOf, type, label, sameAs)

addBlankNodeAnnotation(*node_id*)
Add an annotation property to the given `node_id` to be a pseudo blank node. This is a monarchism. :param node_id: :return:

addClassToGraph(*class_id*, *label=None*, *class_type=None*, *description=None*, *class_category=None*, *class_type_category=None*)
Any node added to the graph will get at least 3 triples: *(node, type, owl:Class) and *(node, label, literal(label)) *if a type is added,

then the node will be an OWL:subclassOf that the type

*__if a description is provided,__ it will also get added as a dc:description

Parameters

- **class_id** –
- **label** –
- **class_type** –
- **description** –
- **class_category** – a biolink category CURIE for class
- **class_type_category** – a biolink category CURIE for class type

Returns

addComment(*subject_id*, *comment*, *subject_category=None*)

addDefinition(*class_id*, *definition*, *class_category=None*)

addDepiction(*subject_id*, *image_url*)

addDeprecatedClass(*old_id*, *new_ids=None*, *old_id_category=None*, *new_ids_category=None*)
Will mark the oldid as a deprecated class. if one newid is supplied, it will mark it as replaced by. if >1 newid is supplied, it will mark it with consider properties :param old_id: str - the class id to deprecate :param new_ids: list - the class list that is

the replacement(s) of the old class. Not required.

:param old_id_category - a biolink category CURIE for old id :param new_ids_category - a biolink category CURIE for new ids :return: None

addDeprecatedIndividual(*old_id*, *new_ids=None*, *old_id_category=None*, *new_id_category=None*)
Will mark the oldid as a deprecated individual. if one newid is supplied, it will mark it as replaced by. if >1 newid is supplied, it will mark it with consider properties :param g: :param oldid: the individual id to deprecate :param newids: the individual idlist that is the replacement(s) of

the old individual. Not required.

:param old_id_category - a biolink category CURIE for old id :param new_ids_category - a biolink category CURIE for new ids :return:

addDescription(*subject_id*, *description*, *subject_category=None*)

addEquivalentClass(*sub*, *obj*, *subject_category=None*, *object_category=None*)

addIndividualToGraph(*ind_id*, *label*, *ind_type=None*, *description=None*, *ind_category=None*, *ind_type_category=None*)

addLabel(*subject_id*, *label*, *subject_category=None*)

addOWLPropertyClassRestriction(*class_id*, *property_id*, *property_value*, *class_category=None*, *property_id_category=None*, *property_value_category=None*)

addOWLVersionIRI(*ontology_id*, *version_iri*)

addOWLVersionInfo(*ontology_id*, *version_info*)

addOntologyDeclaration(*ontology_id*)

addPerson(*person_id*, *person_label=None*)

addSameIndividual(*sub*, *obj*, *subject_category=None*, *object_category=None*)

addSubClass(*child_id*, *parent_id*, *child_category=None*, *parent_category=None*)

addSynonym(*class_id*, *synonym*, *synonym_type=None*, *class_category=None*)
Add the synonym as a property of the class cid. Assume it is an exact synonym, unless otherwise specified :param self: :param class_id: class id :param synonym: the literal synonym label :param synonym_type: the CURIE of the synonym type (not the URI) :param class_category: biolink category CURIE for class_id (no biolink category is possible for synonym, since this is added to the triple as a literal) :return:

addTriple(*subject_id*, *predicate_id*, *obj*, *object_is_literal=False*, *literal_type=None*, *subject_category=None*, *object_category=None*)

addType(*subject_id*, *subject_type*, *subject_category=None*, *subject_type_category=None*)

addXref(*class_id*, *xref_id*, *xref_as_literal=False*, *class_category=None*, *xref_category=None*)

makeLeader(*node_id*)
Add an annotation property to the given `node_id` to be the clique_leader. This is a monarchism. :param node_id: :param node_category: a biolink category CURIE for node_id :return:

dipper.models.Pathway module

class dipper.models.Pathway.**Pathway**(*graph*)
Bases: object

This provides convenience methods to deal with gene and protein collections in the context of pathways.

addComponentToPathway(*component_id*, *pathway_id*)
This can be used directly when the component is directly involved in the pathway. If a transforming event is performed on the component first, then the addGeneToPathway should be used instead.

Parameters

- **pathway_id** –
- **component_id** –
- **component_category** – biolink category for component_id
- **pathway_category** – biolink category for pathway_id

Returns

addGeneToPathway(*gene_id*, *pathway_id*)
When adding a gene to a pathway, we create an intermediate 'gene product' that is involved in the pathway, through a blank node.

gene_id RO:has_gene_product _gene_product _gene_product RO:involved_in pathway_id

Parameters

- **pathway_id** –
- **gene_id** –

Returns

addPathway(*pathway_id*, *pathway_label*, *pathway_type=None*, *pathway_description=None*)
Adds a pathway as a class. If no specific type is specified, it will default to a subclass of "GO:cellular_process" and "PW:pathway". :param pathway_id: :param pathway_label: :param pathway_type: :param pathway_description: :return:

dipper.models.Provenance module

class dipper.models.Provenance.**Provenance**(*graph*)
Bases: object

To model provenance as the basis for an association. This encompasses:

- Process history leading to a claim being made, including processes through which evidence is evaluated
- Processes through which information used as evidence is created.

Provenance metadata includes accounts of who conducted these processes, what entities participated in them, and when/where they occurred.

add_agent_to_graph(*agent_id*, *agent_label*, *agent_type=None*, *agent_description=None*, *agent_category=None*)

add_assay_to_graph(*assay_id*, *assay_label*, *assay_type=None*, *assay_description=None*)

add_assertion(*assertion*, *agent*, *agent_label*, *date=None*)
Add assertion to graph :param assertion: :param agent: :param evidence_line: :param date: :return: None

add_date_created(*prov_type*, *date*)

add_study_measure(*study*, *measure*, *object_is_literal=None*)

add_study_parts(*study*, *study_parts*, *study_parts_category=None*)

add_study_to_measurements(*study*, *measurements*)

dipper.models.Reference module

class dipper.models.Reference.**Reference**(*graph*, *ref_id=None*, *ref_type=None*)
Bases: object

To model references for associations (such as journal articles, books, etc.).

By default, references will be typed as "documents", unless if the type is set otherwise.

If a short_citation is set, this will be used for the individual's label. We may wish to subclass this later.

addAuthor(*author*)

addPage(*subject_id*, *page_url*, *subject_category=None*, *page_category=None*)

addRefToGraph()

addTitle(*subject_id*, *title*)

setAuthorList(*author_list*)

Parameters **author_list** – Array of authors

Returns

setShortCitation(*citation*)

setTitle(*title*)

setType(*reference_type*)

setYear(*year*)

dipper.sources package

Submodules

dipper.sources.AnimalQTLdb module

class dipper.sources.AnimalQTLdb.**AnimalQTLdb**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

The Animal Quantitative Trait Loci (QTL) database (Animal QTLdb) is designed to house publicly all available QTL and single-nucleotide polymorphism/gene association data on livestock animal species. This includes:

- chicken
- horse
- cow
- sheep
- rainbow trout
- pig

While most of the phenotypes here are related to animal husbandry, production, and rearing, integration of these phenotypes with other species may lead to insight for human disease.

Here, we use the QTL genetic maps and their computed genomic locations to create associations between the QTLs and their traits. The traits come in their internal Animal Trait ontology vocabulary, which they further map to [Vertebrate Trait](http://bioportal.bioontology.org/ontologies/VT), Product Trait, and Clinical Measurement Ontology vocabularies.

Since these are only associations to broad locations, we link the traits via "is_marker_for", since there is no specific causative nature in the association. p-values for the associations are attached to the Association objects. We default to the UCSC build for the genomic coordinates, and make equivalences.

Any genetic position ranges that are <0, we do not include here.

GENEINFO = 'ftp://ftp.ncbi.nih.gov/gene/DATA/GENE_INFO'

GITDIP = 'https://raw.githubusercontent.com/monarch-initiative/dipper/master'

fetch(*is_dl_forced=False*)

abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'Bos_taurus_info': {'columns': ['tax_id', 'GeneID', 'Symbol', 'LocusTag', '

gene_info_columns = ['tax_id', 'GeneID', 'Symbol', 'LocusTag', 'Synonyms', 'dbXrefs',

getTestSuite()

An abstract method that should be overwritten with tests appropriate for the specific source. :return:

gff_columns = ['SEQNAME', 'SOURCE', 'FEATURE', 'START', 'END', 'SCORE', 'STRAND', 'FRA

parse(*limit=None*)

Parameters **limit** –

Returns

qtl_columns = ['QTL_ID', 'QTL_symbol', 'Trait_name', 'assotype', '(empty)', 'Chromosom

test_ids = {1795, 1798, 8945, 12532, 14234, 17138, 28483, 29016, 29018, 29385, 31023,

trait_mapping_columns = ['VT', 'LPT', 'CMO', 'ATO', 'Species', 'Class', 'Type', 'QTL_C

dipper.sources.Bgee module

class dipper.sources.Bgee.**Bgee**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*, *version=None*)

Bases: *dipper.sources.Source.Source*

Bgee is a database to retrieve and compare gene expression patterns between animal species.

Bgee first maps heterogeneous expression data (currently RNA-Seq, Affymetrix, in situ hybridization, and EST data) to anatomy and development of different species.

Then, in order to perform automated cross species comparisons, homology relationships across anatomies, and comparison criteria between developmental stages, are designed.

check_if_remote_is_newer(*localfile*, *remote_size*, *remote_modify*)

Overrides check_if_remote_is_newer in Source class

Parameters

- **localfile** – str file path
- **remote_size** – str bytes
- **remote_modify** – str last modify date in the form 20160705042714

Returns boolean True if remote file is newer else False

default_species = ['Cavia porcellus', 'Mus musculus', 'Rattus norvegicus', 'Monodelphi

fetch(*is_dl_forced=False*)

Parameters **is_dl_forced** – boolean, force download

Returns

files = {'anat_entity': {'columns': ['Ensembl gene ID', 'gene name', 'anatomical ent

parse(*limit=None*)

Given the input taxa, expects files in the raw directory with the name {tax_id}_anat_entity_all_data_Pan_troglodytes.tsv.zip

Parameters **limit** – int Limit to top ranked anatomy associations per group

Returns None

dipper.sources.BioGrid module

class dipper.sources.BioGrid.**BioGrid**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*)

Bases: *dipper.sources.Source.Source*

Biogrid interaction data

biogrid_ids = [106638, 107308, 107506, 107674, 107675, 108277, 108506, 108767, 108814,

fetch(*is_dl_forced=False*)

Parameters **is_dl_forced** –

Returns None

files = {'identifiers': {'file': 'BIOGRID-IDENTIFIERS-LATEST.tab.zip', 'url': 'http

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)

Parameters **limit** –

Returns

dipper.sources.CTD module

class dipper.sources.CTD.**CTD**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

The Comparative Toxicogenomics Database (CTD) includes curated data describing cross-species chemical–gene/protein interactions and chemical– and gene–disease associations to illuminate molecular mechanisms underlying variable susceptibility and environmentally influenced diseases. (updated monthly).

Here, we fetch, parse, and convert data from CTD into triples, leveraging only the associations based on DIRECT evidence (not using the inferred associations). We currently process the following associations: * chemical-disease * gene-pathway * gene-disease

CTD curates relationships between genes and chemicals/diseases with 'marker/mechanism' or 'therapeutic'. (observe strictly OR) Unfortunately, we cannot disambiguate between marker (gene expression) and mechanism (causation) for these associations. Therefore, we are left to relate these simply by "marker".

We DISCONTIUED at some point prior to 202005 # CTD also pulls in genes and pathway membership from KEGG and REACTOME. # We create groups of these following the pattern that the specific pathway # is a subclass of 'cellular process' (a go process), and the gene is # "involved in" that process.

For diseases, we preferentially use OMIM identifiers when they can be used uniquely over MESH. Otherwise, we use MESH ids.

Note that we scrub the following identifiers and their associated data: * REACT:REACT_116125 - generic disease class * MESH:D004283 - dog diseases * MESH:D004195 - disease models, animal * MESH:D030342 - genetic diseases, inborn * MESH:D040181 - genetic dieases, x-linked * MESH:D020022 - genetic predisposition to a disease

fetch(*is_dl_forced=False*)
Override Source.fetch() Fetches resources from CTD using the CTD.files dictionary Args: :param is_dl_forced (bool): Force download Returns: :return None

files = {'chemical_disease_associations': {'columns': ['ChemicalName', 'ChemicalID',

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)
Override Source.parse() Parses version and interaction information from CTD Args: :param limit (int, optional) limit the number of rows processed Returns: :return None

dipper.sources.ClinVar module

Converts ClinVar XML into RDF triples to be ingested by SciGraph. These triples conform to the core of the SEPIO Evidence & Provenance model

We also use the clinvar curated gene to disease mappings to discern the functional consequence of a variant on a gene in cases where this is ambiguous. For example, some variants are located in two genes overlapping on different

strands, and may only have a functional consequence on one gene. This is suboptimal and we should look for a source that directly provides this.

creating a test set. get a full dataset default ClinVarFullRelease_00-latest.xml.gz get the mapping file default gene_condition_source_id get a list of RCV default CV_test_RCV.txt put the input files the raw directory write the test set back to the raw directory

./scripts/ClinVarXML_Subset.sh | gzip > raw/clinvar/ClinVarTestSet.xml.gz

parsing a test set (Skolemizing blank nodes i.e. for Protege) dipper/sources/ClinVar.py -f ClinVarTestSet.xml.gz -o ClinVarTestSet_‘datestamp‘.nt

For while we are still required to redundantly conflate the owl properties in with the data files.

python3 ./scripts/add-properties2turtle.py –input ./out/ClinVarTestSet_‘datestamp‘.nt –output ./out/ClinVarTestSet_‘datestamp‘.nt –format nt

dipper.sources.ClinVar.**allele_to_triples**(*allele*, *triples*) → None
Process allele info such as dbsnp ids and synonyms :param allele: Allele :param triples: List, Buffer to store the triples :return: None

dipper.sources.ClinVar.**digest_id**(*wordage*)
return a deterministic digest of input the ‘b’ is an experiment forcing the first char to be non numeric but valid hex; which is in no way required for RDF but may help when using the identifier in other contexts which do not allow identifiers to begin with a digit

:param wordage the string to hash :returns 20 hex char digest

dipper.sources.ClinVar.**expand_curie**(*this_curie*)

dipper.sources.ClinVar.**is_literal**(*thing*)
make inference on type (literal or CURIE)

return: logical

dipper.sources.ClinVar.**make_biolink_category_triple**(*subj*, *cat*)

dipper.sources.ClinVar.**make_spo**(*sub*, *prd*, *obj*, *subject_category=None*, *object_category=None*)
Decorates the three given strings as a line of ntriples (also writes a triple for subj biolink:category and obj biolink:category)

dipper.sources.ClinVar.**parse**()
Main function for parsing a clinvar XML release and outputting triples

dipper.sources.ClinVar.**process_measure_set**(*measure_set*, *rcv_acc*) → dipper.models.ClinVarRecord.Variant
Given a MeasureSet, create a Variant object :param measure_set: XML object :param rcv_acc: str rcv accession :return: Variant object

dipper.sources.ClinVar.**record_to_triples**(*rcv: dipper.models.ClinVarRecord.ClinVarRecord*, *triples: List[T]*, *g2p_map: Dict[KT, VT]*) → None
Given a ClinVarRecord, adds triples to the triples list

Parameters

- **rcv** – ClinVarRecord
- **triples** – List, Buffer to store the triples
- **g2p_map** – Gene to phenotype dict

Returns None

dipper.sources.ClinVar.**resolve**(*label*)

composite mapping given f(x) and g(x) here: GLOBALTT & LOCALTT respectivly in order of preference return g(f(x))|f(x)|g(x) | x TODO consider returning x on fall through

the decendent resolve(label) function in Source.py # should be used instead and this f(x) removed

: return label's mapping

dipper.sources.ClinVar.**scv_link**(*scv_sig*, *rcv_trip*)

Creates links between SCV based on their pathonicty/significance calls

GENO:0000840 - GENO:0000840 –> is_equilavent_to SEPIO:0000098 # GENO:0000841 - GENO:0000841 –> is_equilavent_to SEPIO:0000098 # GENO:0000843 - GENO:0000843 –> is_equilavent_to SEPIO:0000098 # GENO:0000844 - GENO:0000844 –> is_equilavent_to SEPIO:0000098 # GENO:0000840 - GENO:0000844 –> contradicts SEPIO:0000101 # GENO:0000841 - GENO:0000844 –> contradicts SEPIO:0000101 # GENO:0000841 - GENO:0000843 –> contradicts SEPIO:0000101 # GENO:0000840 - GENO:0000841 –> is_consistent_with SEPIO:0000099 # GENO:0000843 - GENO:0000844 –> is_consistent_with SEPIO:0000099 # GENO:0000840 - GENO:0000843 –> strongly_contradicts SEPIO:0000100

dipper.sources.ClinVar.**write_review_status_scores**()

Make triples that attach a "star" score to each of ClinVar's review statuses. (Stars are basically a 0-4 rating of the review status.)

Per https://www.ncbi.nlm.nih.gov/clinvar/docs/details/ Table 1. The review status and assignment of stars(with changes made mid-2015) Number of gold stars Description and review statuses

NO STARS: <ReviewStatus> "no assertion criteria provided" <ReviewStatus> "no assertion provided" No submitter provided an interpretation with assertion criteria (no assertion criteria provided), or no interpretation was provided (no assertion provided)

ONE STAR: <ReviewStatus> "criteria provided, single submitter" <ReviewStatus> "criteria provided, conflicting interpretations" One submitter provided an interpretation with assertion criteria (criteria provided, single submitter) or multiple submitters provided assertion criteria but there are conflicting interpretations in which case the independent values are enumerated for clinical significance (criteria provided, conflicting interpretations)

TWO STARS: <ReviewStatus> "criteria provided, multiple submitters, no conflicts" Two or more submitters providing assertion criteria provided the same interpretation (criteria provided, multiple submitters, no conflicts)

THREE STARS: <ReviewStatus> "reviewed by expert panel" reviewed by expert panel

FOUR STARS: <ReviewStatus> "practice guideline" practice guideline A group wishing to be recognized as an expert panel must first apply to ClinGen by completing the form that can be downloaded from our ftp site.

:param None :return: list of triples that attach a "star" score to each of ClinVar's review statuses

dipper.sources.ClinVar.**write_spo**(*sub*, *prd*, *obj*, *triples*, *subject_category=None*, *object_category=None*)

write triples to a buffer in case we decide to drop them

dipper.sources.Coriell module

class dipper.sources.Coriell.**Coriell**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

The Coriell Catalog provided to Monarch includes metadata and descriptions of NIGMS, NINDS, NHGRI, and NIA cell lines. These lines are made available for research purposes. Here, we create annotations for the cell lines as models of the diseases from which they originate.

We create a handle for a patient from which the given cell line is derived (since there may be multiple cell lines created from a given patient). A genotype is assembled for a patient, which includes a karyotype (if specified) and/or a collection of variants. Both the genotype (has_genotype) and disease are linked to the patient (has_phenotype), and the cell line is listed as derived from the patient. The cell line is classified by it's [CLO cell type](http://www.ontobee.org/browser/index.php?o=clo), which itself is linked to a tissue of origin.

Unfortunately, the omim numbers listed in this file are both for genes & diseases; we have no way of knowing a priori if a designated omim number is a gene or disease; so we presently link the patient to any omim id via the has_phenotype relationship.

Notice: The Coriell catalog is delivered to Monarch in a specific format, and requires ssh rsa fingerprint identification. Other groups wishing to get this data in it's raw form will need to contact Coriell for credential This needs to be placed into your configuration file for it to work.

column_labels = ['catalog_id', 'description', 'omim_num', 'sample_type', 'cell_line_av

fetch(*is_dl_forced=False*)

Here we connect to the coriell sftp server using private connection details. They dump bi-weekly files with a timestamp in the filename. For each catalog, we ping the remote site and pull the most-recently updated file, renaming it to our local latest.csv.

Be sure to have pg user/password connection details in your conf.yaml file, like: dbauth : {"coriell" : { "user" : "<username>", "password" : "<password>", "host" : <host>, "private_key"=path/to/rsa_key} }

Parameters **is_dl_forced** –

Returns

files = {'NHGRI': {'columns': ['catalog_id', 'description', 'omim_num', 'sample_type'

getTestSuite()

An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)

abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

test_lines = ['ND02380', 'ND02381', 'ND02383', 'ND02384', 'GM17897', 'GM17898', 'GM178

dipper.sources.Decipher module

dipper.sources.EBIGene2Phen module

class dipper.sources.EBIGene2Phen.**EBIGene2Phen**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

From EBI: The gene2phenotype dataset (G2P) integrates data on genes, variants and phenotypes for example relating to developmental disorders. It is constructed entirely from published literature, and is primarily an inclusion list to allow targeted filtering of genome-wide data for diagnostic purposes. The dataset was compiled with respect to published genes, and annotated with types of disease- causing gene variants. Each row of the dataset associates a gene with a disease phenotype via an evidence level, inheritance mechanism and mutation consequence. Some genes therefore appear in the database more than once, where different genetic mechanisms result in different phenotypes.

Disclaimer: https://www.ebi.ac.uk/gene2phenotype/disclaimer Terms of Use: https://www.ebi.ac.uk/about/terms-of-use#general Documentation: https://www.ebi.ac.uk/gene2phenotype/documentation

https://www.clinicalgenome.org/site/assets/files/ 2757/fitzpatrick_ddg2p.pdf

This script operates on the Developmental Disorders (DDG2P.csv) file In the future we may update to include the cancer gene disease pairs in the CancerG2P.csv file

EBI_BASE = 'https://www.ebi.ac.uk/gene2phenotype/downloads/'

fetch(*is_dl_forced: bool = False*)
Fetch DDG2P.csv.gz and check headers to see if it has been updated

Parameters **is_dl_forced** – {bool}

Returns None

files = {'developmental_disorders': {'columns': ['gene_symbol', 'gene_omim_id', 'dis

map_files = {'mondo_map': 'https://data.monarchinitiative.org/dipper/cache/unmapped_e

parse(*limit: Optional[int] = None*)
Here we parse each row of the gene to phenotype file

We create anonymous variants along with their attributes (allelic requirement, functional consequence) and connect these to genes and diseases

genes are connected to variants via global_terms['has_affected_locus']

variants are connected to attributes via: global_terms['has_allelic_requirement'] global_terms['has_functional_consequence']

variants are connected to disease based on mappings to the DDD category column, see the translationtable specific to this source for mappings

For cases where there are no disease OMIM id, we either use a disease cache file with mappings to MONDO that has been manually curated

Parameters **limit** – {int} number of rows to parse

Returns None

dipper.sources.EOM module

class dipper.sources.EOM.**EOM**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.PostgreSQLSource.PostgreSQLSource*

Elements of Morphology is a resource from NHGRI that has definitions of morphological abnormalities, together with image depictions. We pull those relationships, as well as our local mapping of equivalences between EOM and HP terminologies.

The website is crawled monthly by NIF's DISCO crawler system, which we utilize here. Be sure to have pg user/password connection details in your conf.yaml file, like: dbauth : {'disco' : {'user' : '<username>', 'password' : '<password>'}}

Monarch-curated data for the HP to EOM mapping is stored at https://raw.githubusercontent.com/obophenotype/human-phenotype-ontology/master/src/mappings/hp-to-eom-mapping.tsv

Since this resource is so small, the entirety of it is the "test" set.

GHRAW = 'https://raw.githubusercontent.com/obophenotype/human-phenotype-ontology'

fetch(*is_dl_forced=False*)
connection details for DISCO

files = {'map': {'columns': ['morphology_term_id', 'morphology_term_label', 'HP ID',

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)
Over ride Source.parse inherited via PostgreSQLSource

resources = {'tables': {'columns': ['morphology_term_id', 'morphology_term_num', 'mo

tables = ['dvp.pr_nlx_157874_1']

dipper.sources.Ensembl module

class dipper.sources.Ensembl.**Ensembl**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*, *gene_ids=None*)
Bases: *dipper.sources.Source.Source*

This is the processing module for Ensembl.

It only includes methods to acquire the equivalences between NCBIGene and ENSG ids using ENSEMBL's Biomart services.

columns = {'bmq_attributes': ['ensembl_gene_id', 'external_gene_name', 'description',

fetch(*is_dl_forced=True*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

fetch_protein_gene_map(*taxon_id*)
Fetch a mapping from proteins to ensembl_gene(S)? for a species in biomart :param taxid: :return: dict

fetch_uniprot_gene_map(*taxon_id*)
Fetch a dict of uniprot-gene for a species in biomart :param taxid: :return: dict

files = {'10090': {'file': 'ensembl_10090.txt'}, '10116': {'file': 'ensembl_10116.

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.FlyBase module

class dipper.sources.FlyBase.**FlyBase**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.PostgreSQLSource.PostgreSQLSource*

This is the [Drosophila Genetics](http://www.flybase.org/) resource, from which we process genotype and phenotype data about the fruit fly.

Here, we connect to their public database and download preprocessed files

Queries from the relational db 1. allele-phenotype data: ../../sources/sql/fb/allele_phenotype.sql 2. gene dbxrefs: ../../resources/sql/fb/gene_xref.sql

Downloads: 1. allele_human_disease_model_data_fb_*.tsv.gz - models of disease 2. species.ab.gz - species prefix mappings 3. fbal_to_fbgn_fb*.tsv.gz - allele to gene 4. fbrf_pmid_pmcid_doi_fb_*.tsv.gz - flybase ref to pmid

We connect using the [Direct Chado Access](http://gmod.org/wiki/Public_Chado_Databases#Direct_Chado_Access)

When running the whole set, it performs best by dumping raw triples using the flag `--format nt`.

Note that this script underwent a major revision after commit bd5f555 in which genotypes, stocks, and environments were removed

CURREL = 'releases/current/precomputed_files'

FLYFTP = 'ftp.flybase.net'

fetch(*is_dl_forced=False*)
Fetch flat files and sql queries

Parameters **is_dl_forced** – force download

Returns None

files = {'allele_gene': {'columns': ['AlleleID', 'AlleleSymbol', 'GeneID', 'GeneSymb

parse(*limit=None*)
Parse flybase files and add to graph

Parameters **limit** – number of rows to process

Returns None

queries = {'allele_phenotype': {'columns': ['allele_id', 'pheno_desc', 'pheno_type',

dipper.sources.GWASCatalog module

class dipper.sources.GWASCatalog.**GWASCatalog**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

The NHGRI-EBI Catalog of published genome-wide association studies.

We link the variants recorded here to the curated EFO-classes using a "contributes to" linkage because the only thing we know is that the SNPs are associated with the trait/disease, but we don't know if it is actually causative.

Description of the GWAS catalog is here: http://www.ebi.ac.uk/gwas/docs/fileheaders#_file_headers_for_catalog_version_1_0_1

GWAS also pulishes Owl files described here http://www.ebi.ac.uk/gwas/docs/ontology

Status: IN PROGRESS

GWASFILE = 'gwas-catalog-associations_ontology-annotated.tsv'

GWASFTP = 'ftp://ftp.ebi.ac.uk/pub/databases/gwas/releases/latest/'

fetch(*is_dl_forced=False*)

Parameters **is_dl_forced** –

Returns

files = {'catalog': {'columns': ['DATE ADDED TO CATALOG', 'PUBMEDID', 'FIRST AUTHOR'

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

process_catalog(*limit=None*)

Parameters **limit** –

Returns

dipper.sources.GeneOntology module

class dipper.sources.GeneOntology.**GeneOntology**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*)

Bases: *dipper.sources.Source.Source*

This is the parser for the [Gene Ontology Annotations](http://www.geneontology.org), from which we process gene-process/function/subcellular location associations.

We generate the GO graph to include the following information: * genes * gene-process * gene-function * gene-location

We process only a subset of the organisms:

Status: IN PROGRESS / INCOMPLETE

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'10090': {'columnns': ['DB', 'DB_Object_ID', 'DB_Object_Symbol', 'Qualifier

gaf_columns = ['DB', 'DB_Object_ID', 'DB_Object_Symbol', 'Qualifier', 'GO_ID', 'DB:Ref

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

get_uniprot_entrez_id_map()

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

process_gaf(*gaffile*, *limit*, *id_map=None*)

wont_prefix = ['zgc', 'wu', 'si', 'im', 'BcDNA', 'sb', 'anon-EST', 'EG', 'id', 'zmp',

dipper.sources.GeneReviews module

dipper.sources.HGNC module

dipper.sources.HPOAnnotations module

class dipper.sources.HPOAnnotations.**HPOAnnotations**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

The [Human Phenotype Ontology](http://human-phenotype-ontology.org) group curates and assembles over 115,000 annotations to hereditary diseases using the HPO ontology. Here we create OBAN-style associations between diseases and phenotypic features, together with their evidence, and age of onset and frequency (if known). The parser currently only processes the "abnormal" annotations. Association to "remarkable normality" will be added in the near future.

We create additional associations from text mining. See info at http://pubmed-browser.human-phenotype-ontology.org/.

Also, you can read about these annotations in [PMID:26119816](http://www.ncbi.nlm.nih.gov/pubmed/26119816).

In order to properly test this class, you should have a resources/test_ids.yaml file configured with some test ids, in the structure of: # as examples. put your favorite ids in the config. <pre> test_ids: {"disease" : ["OMIM:119600", "OMIM:120160"]} </pre>

add_common_files_to_file_list()
The (several thousands) common-disease files from the repo tarball are added to the files object. try adding the 'common-disease-mondo' files as well?

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'doid': {'file': 'doid.owl', 'url': 'http://purl.obolibrary.org/obo/doid.o

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

get_common_files()
Fetch the hpo-annotation-data [repository](https://github.com/monarch-initiative/hpo-annotation-data.git) as a tarball

Returns

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

process_all_common_disease_files(*limit=None*)
Loop through all of the files that we previously fetched from git, creating the disease-phenotype association. :param limit: :return:

process_common_disease_file(*raw*, *unpadded_doids*, *limit=None*)
Make disaese-phenotype associations. Some identifiers need clean up: * DOIDs are listed as DOID-DOID: –> DOID: * DOIDs may be unnecessarily zero-padded. these are remapped to their non-padded equivalent.

Parameters

- **raw** –
- **unpadded_doids** –
- **limit** –

Returns

small_files = {'columns': ['Disease ID', 'Disease Name', 'Gene ID', 'Gene Name', 'Gen

dipper.sources.IMPC module

class dipper.sources.IMPC.**IMPC**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

From the [IMPC](https://mousephenotype.org) website: The IMPC is generating a knockout mouse strain for every protein coding gene by using the embryonic stem cell resource generated by the International Knockout Mouse Consortium (IKMC). Systematic broad-based phenotyping is performed by each IMPC center using standardized procedures found within the International Mouse Phenotyping Resource of Standardised Screens (IMPReSS) resource. Gene-to-phenotype associations are made by a versioned statistical analysis with all data freely available by this web portal and by several data download features.

Here, we pull the data and model the genotypes using GENO and the genotype-to-phenotype associations using the OBAN schema.

We use all identifiers given by the IMPC with a few exceptions:

- For identifiers that IMPC provides, but does not resolve,

we instantiate them as Blank Nodes. Examples include things with the pattern of: UROALL, EUROCURATE, NULL-*,

- We mint three identifiers:

1. Intrinsic genotypes not including sex, based on:

- colony_id (ES cell line + phenotyping center)
- strain
- zygosity

2. For the Effective genotypes that are attached to the phenotypes:

- colony_id (ES cell line + phenotyping center)
- strain
- zygosity
- sex

3. Associations based on: effective_genotype_id + phenotype_id + phenotyping_center + pipeline_stable_id + procedure_stable_id + parameter_stable_id

We DO NOT yet add the assays as evidence for the G2P associations here. To be added in the future.

compare_checksums()
test to see if fetched file matches checksum from ebi :return: True or False

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'checksum': {'file': 'genotype-phenotype-assertions-ALL.csv.tgz.md5', 'url'

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)
IMPC data is delivered in three separate csv files OR in one integrated file, each with the same file format.

Parameters **limit** –

Returns

parse_checksum_file(*file*)
:param file :return dict

dipper.sources.KEGG module

dipper.sources.MGI module

class dipper.sources.MGI.**MGI**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.PostgreSQLSource.PostgreSQLSource*

This is the [Mouse Genome Informatics](http://www.informatics.jax.org/) resource, from which we process genotype and phenotype data about laboratory mice. Genotypes leverage the GENO genotype model.

Here, we connect to their public database, and download a subset of tables/views to get specifically at the geno-pheno data, then iterate over the tables. We end up effectively performing joins when adding nodes to the graph. In order to use this parser, you will need to have user/password connection details in your conf.yaml file, like: dbauth : {'mgi' : {'user' : '<username>', 'password' : '<password>'}} You can request access by contacting mgi-help@jax.org

fetch(*is_dl_forced=False*)

For the MGI resource, we connect to the remote database, and pull the tables into local files. We'll check the local table versions against the remote version :return:

fetch_transgene_genes_from_db(*cxn*)

This is a custom query to fetch the non-mouse genes that are part of transgene alleles.

Parameters **cxn** –

Returns

parse(*limit=None*)

We process each of the postgres tables in turn. The order of processing is important here, as we build up a hashmap of internal vs external identifers (unique keys by type to MGI id). These include allele, marker (gene), publication, strain, genotype, annotation (association), and descriptive notes. :param limit: Only parse this many rows in each table :return:

process_mgi_note_allele_view(*limit=None*)

These are the descriptive notes about the alleles. Note that these notes have embedded HTML - should we do anything about that? :param limit: :return:

process_mgi_relationship_transgene_genes(*limit=None*)

Here, we have the relationship between MGI transgene alleles, and the non-mouse gene ids that are part of them. We augment the allele with the transgene parts.

Parameters **limit** –

Returns

resources = {'query_map': [{'query': '../../resources/sql/mgi/mgi_dbinfo.sql', 'outf

tables = {'all_allele_mutation_view': {'columns': ['_allele_key', 'mutation']}, 'all

unknown_taxa = ['Not Applicable', 'Not Specified']

dipper.sources.MGISlim module

class dipper.sources.MGISlim.**MGISlim**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

slim mgi model only containing Gene to phenotype associations Uses mousemine: http://www.mousemine.org/mousemine/begin.do python lib api http://intermine.org/intermine-ws-python/

fetch(*is_dl_forced=False*)

abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.MMRRC module

class dipper.sources.MMRRC.**MMRRC**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Here we process the Mutant Mouse Resource and Research Center (https://www.mmrrc.org) strain data, which includes: * strains, their mutant alleles * phenotypes of the alleles * descriptions of the research uses of the strains

Note that some gene identifiers are not included (for many of the transgenics with human genes) in the raw data. We do our best to process the links between the variant and the affected gene, but sometimes the mapping is not clear, and we do not include it. Many of these details will be solved by merging this source with the MGI data source, who has the variant-to-gene designations.

Also note that even though the strain pages at the MMRRC site do list phenotypic differences in the context of the strain backgrounds, they do not provide that data to us, and thus we cannot supply that disambiguation here.

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'catalog': {'columns': ['STRAIN/STOCK_ID', 'STRAIN/STOCK_DESIGNATION', 'STR

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

test_ids = ['MMRRC:037507-MU', 'MMRRC:041175-UCD', 'MMRRC:036933-UNC', 'MMRRC:037884-U

dipper.sources.MPD module

class dipper.sources.MPD.**MPD**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

From the [MPD](http://phenome.jax.org/) website: This resource is a collaborative standardized collection of measured data on laboratory mouse strains and populations. Includes baseline phenotype data sets as well as studies of drug, diet, disease and aging effect. Also includes protocols, projects and publications, and SNP, variation and gene expression studies.

Here, we pull the data and model the genotypes using GENO and the genotype-to-phenotype associations using the OBAN schema.

MPD provide measurements for particular assays for several strains. Each of these measurements is itself mapped to a MP or VT term as a phenotype. Therefore, we can create a strain-to-phenotype association based on those strains that lie outside of the "normal" range for the given measurements. We can compute the average of the measurements for all strains tested, and then threshold any extreme measurements being beyond some threshold beyond the average.

Our default threshold here, is +/-2 standard deviations beyond the mean.

Because the measurements are made and recorded at the level of a specific sex of each strain, we associate the MP/VT phenotype with the sex-qualified genotype/strain.

MPDDL = 'http://phenomedoc.jax.org/MPD_downloads'

static build_measurement_description(*row*, *localtt*)

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'assay_metadata': {'columns': ['measnum', 'mpdsector', 'projsym', 'varname'

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

mgd_agent_id = 'MPD:db/q?rtn=people/allinv'

mgd_agent_label = 'Mouse Phenotype Database'

mgd_agent_type = 'foaf:organization'

parse(*limit=None*)
MPD data is delivered in four separate csv files and one xml file, which we process iteratively and write out as one large graph.

Parameters **limit** –

Returns

test_ids = ['MPD:6', 'MPD:849', 'MPD:425', 'MPD:569', 'MPD:10', 'MPD:1002', 'MPD:39',

dipper.sources.Monarch module

class dipper.sources.Monarch.**Monarch**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

This is the parser for data curated by the [Monarch Initiative](https://monarchinitiative.org). Data is currently maintained in a private repository, soon to be released.

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'omia_d2p': {'columns': ['Disease ID', 'Species ID', 'Breed Name', 'Variant

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

process_omia_phenotypes(*limit*)

dipper.sources.Monochrom module

class dipper.sources.Monochrom.**Monochrom**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*)
Bases: *dipper.sources.Source.Source*

This class will leverage the GENO ontology and modeling patterns to build an ontology of chromosomes for any species. These classes represent major structural pieces of Chromosomes which are often universally referenced, using physical properties/observations that remain constant over different genome builds (such as banding patterns and arms). The idea is to create a scaffold upon which we can hang build-specific chromosomal coordinates, and reason across them.

In general, this will take the cytogenic bands files from UCSC, and create missing grouping classes, in order to build the partonomy from a very specific chromosomal band up through the chromosome itself and enable overlap and containment queries. We use RO:subsequence_of as our relationship between nested chromosomal parts. For example, 13q21.31 ==> 13q21.31, 13q21.3, 13q21, 13q2, 13q, 13

At the moment, this only computes the bands for Human, Mouse, Zebrafish, and Rat but will be expanding in the future as needed.

Because this is a universal framework to represent the chromosomal structure of any species, we must mint identifiers for each chromosome and part. (note: in truth we create blank nodes and then pretend they are something else. TEC)

We differentiate species by first creating a species-specific genome, then for each species-specific chromosome we include the NCBI taxon number together with the chromosome number, like: `<species number>chr<num><band>`. For 13q21.31, this would be 9606chr13q21.31. We then create triples for a given band like: <pre> CHR:9606chr1p36.33 rdf[type] SO:chromosome_band CHR:9606chr1p36 subsequence_of :9606chr1p36.3 </pre> where any band in the file is an instance of a chr_band (or a more specific type), is a subsequence of it's containing region.

We determine the containing regions of the band by parsing the band-string; since each alphanumeric is a significant "place", we can split it with the shorter strings being parents of the longer string

Since this is small, and we have limited other items in our test set to a small region, we simply use the whole graph (genome) for testing purposes, and copy the main graph to the test graph.

Since this Dipper class is building an ONTOLOGY, rather than instance-level data, we must also include domain and range constraints, and other owl-isms.

TODO: any species by commandline argument

We are currently mapping these to the **CHR idspace**, but this is NOT YET APPROVED and is subject to change.

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'10090': {'build_num': 'mm10', 'file': '10090cytoBand.txt.gz', 'genome_lab

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

make_parent_bands(*band*, *child_bands*)
this will determine the grouping bands that it belongs to, recursively 13q21.31 ==> 13, 13q, 13q2, 13q21, 13q21.3, 13q21.31

Parameters

- **band** –
- **child_bands** –

Returns

map_type_of_region(*regiontype*)
Note that "stalk" refers to the short arm of acrocentric chromosomes chr13,14,15,21,22 for human. :param regiontype: :return:

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.Monochrom.**getChrPartTypeByNotation**(*notation*, *graph*)
This method will figure out the kind of feature that a given band is based on pattern matching to standard karyotype notation. (e.g. 13q22.2 ==> chromosome sub-band)

This has been validated against human, mouse, fish, and rat nomenclature. :param notation: the band (without the chromosome prefix) :return:

dipper.sources.MyChem module

class dipper.sources.MyChem.**MyChem**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

static add_relation(*results*, *relation*)

static check_uniprot(*target_dict*)

static chunks(*l*, *n*)
Yield successive n-sized chunks from l.

static execute_query(*query*)

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

fetch_from_mychem()

static format_actions(*target_dict*)

static get_drug_record(*ids*, *fields*)

static get_inchikeys()

make_triples(*source*, *package*)

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

static return_target_list(*targ_in*)

dipper.sources.MyDrug module

class dipper.sources.MyDrug.**MyDrug**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Drugs and Compounds stored in the BioThings database

MY_DRUG_API = 'http://c.biothings.io/v1/query'

check_if_remote_is_newer(*localfile*)
Need to figure out how biothings records releases, for now if the file exists we will assume it is a fully downloaded cache :param localfile: str file path :return: boolean True if remote file is newer else False

fetch(*is_dl_forced=False*)
Note there is a unpublished mydrug client that works like this: from mydrug import MyDrugInfo md = MyDrugInfo() r = list(md.query('_exists_:aeolus', fetch_all=True))

Parameters **is_dl_forced** – boolean, force download

Returns

files = {'aeolus': {'file': 'aeolus.json'}}

parse(*limit=None*, *or_limit=1*)
Parse mydrug files :param limit: int limit json docs processed :param or_limit: int odds ratio limit :return: None

dipper.sources.NCBIGene module

dipper.sources.OMIA module

dipper.sources.OMIM module

dipper.sources.OMIMSource module

dipper.sources.Orphanet module

class dipper.sources.Orphanet.**Orphanet**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Orphanet's aim is to help improve the diagnosis, care and treatment of patients with rare diseases. For Orphanet, we are currently only parsing the disease-gene associations.

fetch(*is_dl_forced=False*)

Parameters **is_dl_forced** –

Returns

files = {'disease-gene': {'file': 'en_product6.xml', 'url': 'http://www.orphadata.o

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.Panther module

class dipper.sources.Panther.**Panther**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*)
Bases: *dipper.sources.Source.Source*

The pairwise orthology calls from Panther DB: http://pantherdb.org/ encompass 22 species, from the RefGenome and HCOP projects. Here, we map the orthology classes to RO homology relationships This resource may be extended in the future with additional species.

This currently makes a graph of orthologous relationships between genes, with the assumption that gene metadata (labels, equivalent ids) are provided from other sources.

Gene families are nominally created from the orthology files, though these are incomplete with no hierarchical (subfamily) information. This will get updated from the HMM files in the future.

Note that there is a fair amount of identifier cleanup performed to align with our standard CURIE prefixes.

The test graph of data is output based on configured "protein" identifiers in resources/test_id.yaml.

By default, this will produce a file with ALL orthologous relationships. IF YOU WANT ONLY A SUBSET, YOU NEED TO PROVIDE A FILTER UPON CALLING THIS WITH THE TAXON IDS

PNTHDL = 'ftp://ftp.pantherdb.org/ortholog/current_release'

fetch(*is_dl_forced=False*)

Returns None

files = {'Orthologs_HCOP': {'columns': ['Gene', 'Ortholog', 'Type of ortholog', 'Comm

getTestSuite()

An abstract method that should be overwritten with tests appropriate for the specific source. :return:

panther_format = ['Gene', 'Ortholog', 'Type of ortholog', 'Common ancestor for the ort

parse(*limit=None*)

Returns None

dipper.sources.PostgreSQLSource module

class dipper.sources.PostgreSQLSource.**PostgreSQLSource**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *name=None*, *ingest_title=None*, *ingest_url=None*, *ingest_logo=None*, *ingest_description=None*, *license_url=None*, *data_rights=None*, *file_handle=None*)

Bases: *dipper.sources.Source.Source*

Class for interfacing with remote Postgres databases

fetch(*is_dl_forced=False*)

abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

fetch_from_pgdb(*tables*, *cxn*, *limit=None*)

Will fetch all Postgres tables from the specified database in the cxn connection parameters.

This will save them to a local file named the same as the table, in tab-delimited format, including a header.

Parameters

- **tables** – Names of tables to fetch
- **cxn** – database connection details
- **limit** – A max row count to fetch for each table

Returns None

fetch_query_from_pgdb(*qname*, *query*, *con*, *cxn*, *limit=None*)
Supply either an already established connection, or connection parameters. The supplied connection will override any separate cxn parameter :param qname: The name of the query to save the output to :param query: The SQL query itself :param con: The already-established connection :param cxn: The postgres connection information :param limit: If you only want a subset of rows from the query :return:

files = {}

parse(*limit*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.RGD module

class dipper.sources.RGD.**RGD**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Ingest of Rat Genome Database gene to mammalian phenotype gaf file

RGD_BASE = 'ftp://ftp.rgd.mcw.edu/pub/data_release/annotated_rgd_objects_by_ontology/'

fetch(*is_dl_forced=False*)
Override Source.fetch() Fetches resources from rat_genome_database via the rat_genome_database ftp site Args:

param is_dl_forced (bool) Force download

Returns: :return None

files = {'rat_gene2mammalian_phenotype': {'columns': ['DB', 'DB Object ID', 'DB Obje

make_association(*record*)
contstruct the association :param record: :return: modeled association of genotype to mammalian phenotype

parse(*limit=None*)
Override Source.parse() Args:

:param limit (int, optional) limit the number of rows processed

Returns: :return None

dipper.sources.Reactome module

class dipper.sources.Reactome.**Reactome**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Reactome is a free, open-source, curated and peer reviewed pathway database. (http://reactome.org/)

REACTOME_BASE = 'http://www.reactome.org/download/current/'

fetch(*is_dl_forced=False*)
Override Source.fetch() Fetches resources from reactome using the Reactome.files dictionary Args:

param is_dl_forced (bool) Force download

Returns: :return None

files = {'chebi2pathway': {'columns': ['component', 'pathway_id', 'pathway_iri', 'pa

parse(*limit=None*)
Override Source.parse() Args:

:param limit (int, optional) limit the number of rows processed

Returns: :return None

dipper.sources.SGD module

class dipper.sources.SGD.**SGD**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Ingest of Saccharomyces Genome Database (SGD) phenotype associations

SGD_BASE = 'https://downloads.yeastgenome.org/curation/literature/'

fetch(*is_dl_forced=False*)
Override Source.fetch() Fetches resources from yeast_genome_database using the yeast_genome_doenload site.

Args:

param is_dl_forced (bool) Force download

Returns: :return None

files = {'sgd_phenotype': {'columns': ['Feature Name', 'Feature Type', 'Gene Name',

static make_apo_map()

make_association(*record*)
contstruct the association :param record: :return: modeled association of genotype to mammalian??? phenotype

parse(*limit=None*)
Override Source.parse() Args:

:param limit (int, optional) limit the number of rows processed

Returns: :return None

dipper.sources.Source module

class dipper.sources.Source.**Source**(*graph_type='rdf_graph'*, *are_bnodes_skized=False*, *data_release_version=None*, *name=None*, *ingest_title=None*, *ingest_url=None*, *ingest_logo=None*, *ingest_description=None*, *license_url=None*, *data_rights=None*, *file_handle=None*)
Bases: object

Abstract class for any data sources that we'll import and process. Each of the subclasses will fetch() the data, scrub() it as necessary, then parse() it into a graph. The graph will then be written out to a single self.name().<dest_fmt> file.

Also provides a means to marshal metadata in a consistent fashion

Houses the global translation table (from ontology label to ontology term) so it may as well be used everywhere.

ARGV = {}

DIPPERCACHE = 'https://archive.monarchinitiative.org/DipperCache'

static check_fileheader(*expected*, *received*, *src_key=None*)
Compare file headers received versus file headers expected if the expected headers are a subset (proper or not) of received headers report suscess (warn if proper subset)

param: expected list param: received list

return: truthyness

check_if_remote_is_newer(*remote*, *local*, *headers*)
Given a remote file location, and the corresponding local file this will check the datetime stamp on the files to see if the remote one is newer. This is a convenience method to be used so that we don't have to re-fetch files that we already have saved locally :param remote: URL of file to fetch from remote server :param local: pathname to save file to locally :return: True if the remote file is newer and should be downloaded

command_args()
To make arbitrary variables from dipper-etl.py's calling enviroment available when working in source ingests in a hopefully universal way

Does not appear to be populated till after an ingest's _init_() finishes.

compare_local_remote_bytes(*remotefile*, *localfile*, *remote_headers=None*)
test to see if fetched file is the same size as the remote file using information in the content-length field in the HTTP header :return: True or False

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

fetch_from_url(*remoteurl*, *localfile=None*, *is_dl_forced=False*, *headers=None*)
Given a remote url and a local filename, attempt to determine if the remote file is newer; if it is, fetch the remote file and save it to the specified localfile, reporting the basic file information once it is downloaded :param remoteurl: URL of remote file to fetch :param localfile: pathname of file to save locally

Returns bool

static file_len(*fname*)

files = {}

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

static get_file_md5(*directory*, *filename*, *blocksize=1048576*)

get_files(*is_dl_forced*, *files=None*, *delay=0*)
Given a set of files for this source, it will go fetch them, and set a default version by date. If you need to set the version number by another method, then it can be set again. :param is_dl_forced - boolean :param files dict - override instance files dict :return: None

static get_local_file_size(*localfile*)

Parameters **localfile** –

Returns size of file

get_remote_content_len(*remote*, *headers=None*)

Parameters **remote** –

Returns size of remote file

static hash_id(*wordage*)
prepend 'b' to avoid leading with digit truncate to a 20 char sized word with a leading 'b' return truncated sha1 hash of string.

by the birthday paradox; expect 50% chance of collision after 69 billion invocations however these are only hoped to be unique within a single file

Consider reducing to 17 hex chars to fit in a 64 bit word 16 discounting a leading constant gives a 50% chance of collision at about 4.3b billion unique input strings (currently _many_ orders of magnitude below that)

Parameters **long_string** – str string to be hashed

Returns str hash of id

load_local_translationtable(*name*)

Load "ingest specific" translation from whatever they called something to the ontology label we need to map it to. To facilitate seeing more ontology labels in dipper ingests a reverse mapping from ontology labels to external strings is also generated and available as a dict localtcid

'__

%s.yaml "": "" # example'

static make_id(*long_string*, *prefix='MONARCH'*)
a method to create DETERMINISTIC identifiers based on a string's digest. currently implemented with sha1 :param long_string: :return:

namespaces = {}

static open_and_parse_yaml(*yamlfile*)

Parameters **file** – String, path to file containing label-id mappings in the first two columns of each row

Returns dict where keys are labels and values are ids

parse(*limit*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

static parse_mapping_file(*file*)

Parameters **file** – String, path to file containing label-id mappings in the first two columns of each row

Returns dict where keys are labels and values are ids

process_xml_table(*elem*, *table_name*, *processing_function*, *limit*)
This is a convenience function to process the elements of an xml dump of a mysql relational database. The "elem" is akin to a mysql table, with it's name of `table_name`. It will process each `row` given the `processing_function` supplied. :param elem: The element data :param table_name: The name of the table to process :param processing_function: The row processing function :param limit:

Appears to be making calls to the elementTree library although it not explicitly imported here.

Returns

static remove_backslash_r(*filename*, *encoding*)
A helpful utility to remove Carriage Return from any file. This will read a file into memory, and overwrite the contents of the original file.

TODO: This function may be a liability

Parameters **filename** –

Returns

resolve(*word*, *mandatory=True*, *default=None*)
composite mapping given f(x) and g(x) here: localtt & globaltt respectivly return g(f(x))|g(x)||f(x)|x in order of preference returns x|default on fall through if finding a mapping is not mandatory (by default finding is mandatory).

This may be specialized further from any mapping to a global mapping only; if need be.

Parameters

- **word** – the string to find as a key in translation tables
- **mandatory** – boolean to cause failure when no key exists
- **default** – string to return if nothing is found (& not manandatory)

:return value from global translation table, or value from local translation table, or the query key if finding a value is not mandatory (in this order)

settestmode(*mode*)
Set testMode to (mode). - True: run the Source in testMode; - False: run it in full mode :param mode: :return: None

settestonly(*testonly*)
Set that this source should only be processed in testMode :param testOnly: :return: None

whoami()
pointless convieniance

write(*fmt='turtle'*, *stream=None*, *write_metadata_in_main_graph=True*)

This convenience method will write out all of the graphs associated with the source.

Right now these are hardcoded to be a single main "graph" and a "src_dataset.ttl" and a "src_test.ttl" If you do not supply stream='stdout' it will default write these to files.

In addition, if the version number isn't yet set in the dataset, it will be set to the date on file. :return: None

dipper.sources.StringDB module

class dipper.sources.StringDB.**StringDB**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*, *version=None*)
Bases: *dipper.sources.Source.Source*

STRING is a database of known and predicted protein-protein interactions. The interactions include direct (physical) and indirect (functional) associations; they stem from computational prediction, from knowledge transfer between organisms, and from interactions aggregated from other (primary) databases. From: http://string-db.org/cgi/about.pl?footer_active_subpage=content

STRING uses one protein per gene. If there is more than one isoform per gene, we usually select the longest isoform, unless we have information that suggest that other isoform regarded as cannonical (e.g., proteins in the CCDS database). From: http://string-db.org/cgi/help.pl

fetch(*is_dl_forced=False*)
Override Source.fetch() Fetches resources from String

We also fetch ensembl to determine if protein pairs are from the same species Args:

param is_dl_forced (bool) Force download

Returns: :return None

parse(*limit=None*)
Override Source.parse() Args:

:param limit (int, optional) limit the number of rows processed

Returns: :return None

dipper.sources.UCSCBands module

class dipper.sources.UCSCBands.**UCSCBands**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*, *tax_ids=None*)
Bases: *dipper.sources.Source.Source*

This will take the UCSC defintions of cytogenic bands and create the nested structures to enable overlap and containment queries. We use `Monochrom.py` to create the OWL-classes of the chromosomal parts. Here, we simply worry about the instance-level values for particular genome builds.

Given a chr band definition, the nested containment structures look like: 13q21.31 ==> 13q21.31, 13q21.3, 13q21, 13q2, 13q, 13

We determine the containing regions of the band by parsing the band-string; since each alphanumeric is a significant "place", we can split it with the shorter strings being parents of the longer string. # Here we create build-specific chroms, which are instances of the classes produced from `Monochrom.py`. You can instantiate any number of builds for a genome.

We leverage the Faldo model here for region definitions, and map each of the chromosomal parts to SO.

We differentiate the build by adding the build id to the identifier prior to the chromosome number. These then are instances of the species-specific chromosomal class.

The build-specific chromosomes are created like: <pre> <build number>chr<num><band> with triples for a given band like: _:hg19chr1p36.33

rdf:type SO:chromosome_band, faldo:Region, CHR:9606chr1p36.33, subsequence_of _:hg19chr1p36.3, faldo:location [a faldo:BothStrandPosition

faldo:begin 0, faldo:end 2300000, faldo:reference 'hg19'

] .

</pre> where any band in the file is an instance of a chr_band (or a more specific type), is a subsequence of it's containing region, and is located in the specified coordinates.

We do not have a separate graph for testing.

TODO: any species by commandline argument

HGGP = 'http://hgdownload.cse.ucsc.edu/goldenPath'

cytobandideo_columns = ['chrom', 'chromStart', 'chromEnd', 'name', 'gieStain']

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'10090': {'assembly': ('UCSC:mm10', 'UCSC:mm9'), 'build_num': 'mm10', 'col

getTestSuite()
An abstract method that should be overwritten with tests appropriate for the specific source. :return:

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.UDP module

class dipper.sources.UDP.**UDP**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

The National Institutes of Health (NIH) Undiagnosed Diseases Program (UDP) is part of the Undiagnosed Disease Network (UDN), an NIH Common Fund initiative that focuses on the most puzzling medical cases referred to the NIH Clinical Center in Bethesda, Maryland. from https://www.genome.gov/27544402/the-undiagnosed-diseases-program/

Data is available by request for access via the NHGRI collaboration server: https://udplims-collab.nhgri.nih.gov/api

Note the fetcher requires credentials for the UDP collaboration server Credentials are added via a config file, config.json, in the following format {

“dbauth” [{]

“udp”: { “user”: “foo” “password”: “bar”

}

} See dipper/config.py for more information

Output of fetcher: udp_variants.tsv ‘Patient’, ‘Family’, ‘Chr’, ‘Build’, ‘Chromosome Position’, ‘Reference Allele’, ‘Variant Allele’, ‘Parent of origin’, ‘Allele Type’, ‘Mutation Type’, ‘Gene’, ‘Transcript’, ‘Original Amino Acid’, ‘Variant Amino Acid’, ‘Amino Acid Change’, ‘Segregates with’, ‘Position’, ‘Exon’, ‘Inheritance model’, ‘Zygosity’, ‘dbSNP ID’, ‘1K Frequency’, ‘Number of Alleles’

udp_phenotypes.tsv ‘Patient’, ‘HPID’, ‘Present’

The script also utilizes two mapping files udp_gene_map.tsv - generated from scripts/fetch-gene-ids.py,

gene symbols from udp_variants

udp_chr_rs.tsv - rsid(s) per coordinate greped from hg19 dbsnp file, then disambiguated with eutils, see scripts/dbsnp/dbsnp.py

UDP_SERVER = 'https://udplims-collab.nhgri.nih.gov/api'

fetch(*is_dl_forced=True*)
Fetches data from udp collaboration server, see top level comments for class for more information :return:

files = {'patient_phenotypes': {'file': 'udp_phenotypes.tsv'}, 'patient_variants':

map_files = {'dbsnp_map': '../../resources/udp/udp_chr_rs.tsv', 'gene_coord_map': '.

parse(*limit=None*)
Override Source.parse() Args:

:param limit (int, optional) limit the number of rows processed

Returns: :return None

dipper.sources.WormBase module

class dipper.sources.WormBase.**WormBase**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)

Bases: *dipper.sources.Source.Source*

This is the parser for the [C. elegans Model Organism Database (WormBase)](http://www.wormbase.org), from which we process genotype and phenotype data for laboratory worms (C.elegans and other nematodes).

We generate the wormbase graph to include the following information: * genes * sequence alterations (includes SNPs/del/ins/indel and

large chromosomal rearrangements)

- RNAi as expression-affecting reagents
- genotypes, and their components
- strains
- publications (and their mapping to PMIDs, if available)
- allele-to-phenotype associations (including variants by RNAi)
- genetic positional information for genes and sequence alterations

Genotypes leverage the GENO genotype model and includes both intrinsic and extrinsic genotypes. Where necessary, we create anonymous nodes of the genotype partonomy (i.e. for variant single locus complements, genomic variation complements, variant loci, extrinsic genotypes, and extrinsic genotype parts).

TODO: get people and gene expression

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'allele_pheno': {'columns': ['DB', 'DB Object ID', 'DB Object Symbol', 'Qua

static make_reagent_targeted_gene_id(*gene_id*, *reagent_id*)

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

process_allele_phenotype(*limit=None*)
This file compactly lists variant to phenotype associations, such that in a single row, there may be >1 variant listed per phenotype and paper. This indicates that each variant is individually assocated with the given phenotype, as listed in 1+ papers. (Not that the combination of variants is producing the phenotype.) :param limit: :return:

process_disease_association(*limit*)

process_feature_loc(*limit*)

process_gene_desc(*limit*)

process_gene_ids(*limit*)

process_gene_interaction(*limit*)
The gene interaction file includes identified interactions, that are between two or more gene (products). In the case of interactions with >2 genes, this requires creating groups of genes that are involved in the interaction. From the wormbase help list: In the example WBInteraction000007779 it would likely be misleading to suggest that lin-12 interacts with (suppresses in this case) smo-1 ALONE or that lin-12

suppresses let-60 ALONE; the observation in the paper; see Table V in paper PMID:15990876 was that a lin-12 allele (heterozygous lin-12(n941/+)) could suppress the "multivulva" phenotype induced synthetically by simultaneous perturbation of BOTH smo-1 (by RNAi) AND let-60 (by the n2021 allele). So this is necessarily a three-gene interaction.

Therefore, we can create groups of genes based on their "status" of Effector | Effected.

Status: IN PROGRESS

Parameters **limit** –

Returns

process_pub_xrefs(*limit=None*)

process_rnai_phenotypes(*limit=None*)

species = '/species/c_elegans/PRJNA13758'

update_wsnum_in_files(*vernum*)
With the given version number `vernum`, update the source's version number, and replace in the file hashmap. the version number is in the CHECKSUMS file. :param vernum: :return:

wbdev = 'ftp://ftp.wormbase.org/pub/wormbase/releases/current-development-release'

wbprod = 'ftp://ftp.wormbase.org/pub/wormbase/releases/current-production-release'

wbrel = 'ftp://ftp.wormbase.org/pub/wormbase/releases'

dipper.sources.Xenbase module

class dipper.sources.Xenbase.**Xenbase**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

Xenbase is a web-accessible resource that integrates all the diverse biological, genomic, genotype and phenotype data available from Xenopus research.

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'g2p_assertions': {'columns': ['SUBJECT', 'SUBJECT_LABEL', 'SUBJECT_TAXON',

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.sources.ZFIN module

class dipper.sources.ZFIN.**ZFIN**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

This is the parser for the [Zebrafish Model Organism Database (ZFIN)](http://www.zfin.org), from which we process genotype and phenotype data for laboratory zebrafish.

We generate the zfin graph to include the following information: * genes * sequence alterations (includes SNPs/del/ins/indel and large chromosomal rearrangements) * transgenic constructs * morpholinos, talens, crisprs as expression-affecting reagents * genotypes, and their components * fish (as comprised of intrinsic and extrinsic genotypes) * publications (and their mapping to PMIDs, if available) * genotype-to-phenotype

associations (including environments and stages at which they are assayed) * environmental components * orthology to human genes * genetic positional information for genes and sequence alterations * fish-to-disease model associations

Genotypes leverage the GENO genotype model and include both intrinsic and extrinsic genotypes. Where necessary, we create anonymous nodes of the genotype partonomy (such as for variant single locus complements, genomic variation complements, variant loci, extrinsic genotypes, and extrinsic genotype parts).

Genotype labels are output as ZFIN genotype name + "[background]". We also process the genotype components to build labels in a monarch-style, and these are added as synonyms. The monarch-style genotype label includes: * all genes targeted by reagents (morphants, crisprs, etc), in addition to the ones that the reagent was designed against. * all affected genes within deficiencies * complex hets being listed as gene<mutation1>/gene<mutation2> rather than gene<mutation1>/+; gene<mutation2>/+

see: resources/zfin/README for column extraction from downloads page

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

fhandle = <_io.TextIOWrapper name='/home/docs/checkouts/readthedocs.org/user_builds/di

files = {'backgrounds': {'columns': ['Genotype ID', 'Genotype Name', 'Background',

static get_orthology_evidence_code(*abbrev*)
move to localtt & globltt

get_orthology_sources_from_zebrafishmine()
Fetch the zfin gene to other species orthology annotations, together with the evidence for the assertion. Write the file locally to be read in a separate function. :return:

static make_targeted_gene_id(*geneid*, *reagentid*)

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

process_fish(*limit=None*)
Fish give identifiers to the "effective genotypes" that we create. We can match these by: Fish = (intrinsic) genotype + set of morpholinos

We assume here that the intrinsic genotypes and their parts will be processed separately, prior to calling this function.

Parameters **limit** –

Returns

process_fish_disease_models(*limit=None*)

process_orthology_evidence(*limit*)

test_ids = {'allele': ['ZDB-ALT-010426-4', 'ZDB-ALT-010427-8', 'ZDB-ALT-011017-8',

dipper.sources.ZFINSlim module

class dipper.sources.ZFINSlim.**ZFINSlim**(*graph_type*, *are_bnodes_skolemized*, *data_release_version=None*)
Bases: *dipper.sources.Source.Source*

zfin mgi model only containing Gene to phenotype associations Using the file here: https://zfin.org/downloads/phenoGeneCleanData_fish.txt

fetch(*is_dl_forced=False*)
abstract method to fetch all data from an external resource. this should be overridden by subclasses :return: None

files = {'g2p_clean': {'columns': ['ID', 'Gene Symbol', 'Gene ID', 'Affected Structu

parse(*limit=None*)
abstract method to parse all data from an external resource, that was fetched in fetch() this should be overridden by subclasses :return: None

dipper.utils package

Submodules

dipper.utils.CurieUtil module

class dipper.utils.CurieUtil.**CurieUtil**(*curie_map*)
Bases: object

Create compact URI

get_base()

get_curie(*uri*)
Get a CURIE from a URI

get_curie_prefix(*uri*)
Return the CURIE's prefix:

get_uri(*curie*)
Get a URI from a CURIE

prefix_exists(*pfx*)

dipper.utils.DipperUtil module

class dipper.utils.DipperUtil.**DipperUtil**
Bases: object

Various utilities and quick methods used in this application

(A little too quick) Per: https://www.ncbi.nlm.nih.gov/books/NBK25497/ NCBI recommends that users post

> no more than three URL requests per second and limit large jobs to either weekends or between 9:00 PM and 5:00 AM Eastern time during weekdays

restructuring to make bulk queries is less likely to result in another ban for peppering them with one offs

static get_hgnc_id_from_symbol(*gene_symbol*)
Get HGNC curie from symbol using monarch and mygene services :param gene_symbol: :return:

static get_homologene_by_gene_num(*gene_num*)

static get_ncbi_taxon_num_by_label(*label*)
Here we want to look up the NCBI Taxon id using some kind of label. It will only return a result if there is a unique hit.

Returns

static is_id_in_mondo(*curie*, *mondo_min*)

Parameters

- **curie** – curie formatted ID
- **mondo_min** – a json decoded mondo-minimal.json file, eg https://github.com/monarch-initiative/mondo/releases/ download/2019-04-06/mondo-minimal.json

Returns boolean, true if ID is in mondo and false otherwise

static remove_control_characters(*string*)

Filters out charcters in any of these unicode catagories [Cc] Other, Control (65 characters)

,... [Cf] Other, Format (151 characters) [Cn] Other, Not Assigned (0 characters – none have this property) [Co] Other, Private Use (6 characters) [Cs] Other, Surrogate (6 characters)

dipper.utils.GraphUtils module

class dipper.utils.GraphUtils.**GraphUtils**(*curie_map*)

Bases: object

static add_property_axioms(*graph*, *properties*)

static add_property_to_graph(*results*, *graph*, *property_type*, *property_list*)

static compare_graph_predicates(*graph1*, *graph2*)

From rdf graphs, count predicates in each and return a list of : param graph1 graph, hopefully RDFlib-like : param graph2 graph, ditto : return dict with count of predicates in each graph: : e.g.: : { : "has_a_property": { : "graph1": 1234, : "graph2": 1023}, : "has_another_property": { : "graph1": 94, : "graph2": 51} : }

static count_predicates(*graph*)

From rdf graphs, count predicates in each and return a list of : param graph : return dict with count of predicates in each graph: : e.g.: : { : "has_a_property": 1234, : "has_another_property": 482 : }

static digest_id(*wordage*)

Form a deterministic digest of input Leading 'b' is an experiment forcing the first char to be non numeric but valid hex Not required for RDF but some other contexts do not want the leading char to be a digit

: param str wordage arbitrary string : return str

static get_properties_from_graph(*graph*)

Wrapper for RDFLib.graph.predicates() that returns a unique set :param graph: RDFLib.graph :return: set, set of properties

static write(*graph*, *fileformat=None*, *filename=None*)

A basic graph writer (to stdout) for any of the sources. this will write raw triples in rdfxml, unless specified. to write turtle, specify format='turtle' an optional file can be supplied instead of stdout :return: None

dipper.utils.TestUtils module

class dipper.utils.TestUtils.**TestUtils**

Bases: object

static remove_ontology_axioms(*graph*)

Given an rdflib graph, remove any triples connected to an ontology node: { } a owl:Ontology :param graph: RDFGraph :return: None

static test_graph_equality(*turtlish*, *graph*)

Parameters

- **turtlish** – file path or string of triples in turtle format without prefix header
- **graph** – Graph object to test against

Returns Boolean, True if graphs contain same set of triples

dipper.utils.rdf2dot module

A fork of rdflib rdf2dot utility, see https://rdflib.readthedocs.io/en/stable/_modules/rdflib/tools/rdf2dot.html#rdf2dot

We apply the formatliteral function to labels, but otherwise the code is the same

This is necessary for variants with HGVS primary labels that contain characters that need to be url encoded (<, >)

Also replaces cgi.escape with html.escape

TO DO make a PR

dipper.utils.rdf2dot.**rdf2dot**(*g*, *stream*, *graph_opts=None*)
Convert the RDF graph to DOT writes the dot output to the stream

dipper.utils.romanplus module

Convert to and from Roman numerals This program is part of "Dive Into Python", a free Python tutorial for experienced programmers. Visit http://diveintopython.org/ for the latest version.

This program is free software; you can redistribute it and/or modify it under the terms of the Python 2.1.1 license, available at http://www.python.org/2.1.1/license.html

Note: This has been modified to add optional characters after the initial roman numbers by nlw.

dipper.utils.romanplus.**fromRoman**(*strng*)
convert Roman numeral to integer

dipper.utils.romanplus.**toRoman**(*num*)
convert integer to Roman numeral

Submodules

dipper.config module

dipper.config.**conf = {'keys': {'omim': ''}}**
Load the configuration file 'conf.yaml', if it exists. it isn't always required, but may be for some sources. conf.yaml may contain sensitive info and should not live in a public repo

dipper.config.**get_config**()

dipper.curie_map module

Acroname central

Load the curie mapping file 'curie_map.yaml', it is necessary for most resources

dipper.curie_map.**get**()

dipper.curie_map.**get_base**()

3.2 Source APIs

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

T

U

V

W

X

Y

Z