biothings_client Documentation

Release v0.2.6

BioThings Project

Apr 05, 2021

Tutorials

1 Requirements **3**

2 Optional dependencies **5**

3 Installation **7**

4 Version history **9**

5 Documentation **11**

5.1 Quick Start . . . 11

5.1.1 Use the client for MyGene.info API - Genes . . . 11

5.1.2 Use the client for MyVariant.info API - Variants . . . 14

5.1.3 Use the client for MyChem.info API - Chemicals/Drugs . . . 14

5.1.4 Use the client for MyDisease.info API - Disease . . . 15

5.1.5 Use the client for t.biothings.io API - Taxnomy . . . 16

5.2 Instantiating a Client for a Custom BioThings API . . . 17

5.3 Subclassing a Client . . . 18

5.4 API Documentation . . . 18

5.4.1 get_client . . . 18

5.4.2 MyGeneInfo . . . 19

5.4.3 MyVariantInfo . . . 22

5.4.4 MyChemInfo . . . 26

5.4.5 MyDiseaseInfo . . . 31

5.4.6 MyTaxonInfo . . . 33

6 Indices and tables **37**

Python Module Index **39**

Index **41**

Biothings_client.py is a unified python client providing an easy-to-use wrapper for accessing *any* BioThings API (e.g. MyGene.Info, MyVariant.Info, MyChem.Info). It is the descendent and eventual replacement of both the MyGene.py and MyVariant.py python clients.

CHAPTER 1

Requirements

Python >=2.7 (including python3)

(Python 2.6 might still work, but is not supported any more since v0.2.0)

requests (install using `pip install requests`)

CHAPTER 2

Optional dependencies

- pandas (install using `pip install pandas`) is required for returning a list of objects as DataFrame.
- requests-cache (install using `pip install requests-cache`) is required to use the local data caching function.

CHAPTER 3

Installation

Option 1 Install directly from pip:

```
pip install biothings_client
```

Option 2 download/extract the source code and run:

```
python setup.py install
```

Option 3 install the latest code directly from the repository:

```
pip install -e git+https://github.com/biothings/biothings_client.py
↪#egg=biothings_client
```

CHAPTER 4

Version history

CHANGES.txt

CHAPTER 5

Documentation

http://biothings-clientpy.readthedocs.org/

5.1 Quick Start

biothings_client was made to allow easy programmatic access to any BioThings API backend. We do this by generating configuration parameters (and documentation) that define a particular API on-the-fly. This is done using the *get_client* function. To use the *get_client* function, you only need to specify the entity type you want a client for as a string. Consider the following code:

```
In [1]: from biothings_client import get_client

In [2]: gene_client = get_client('gene')

In [3]: type(gene_client)
Out[3]: biothings_client.MyGeneInfo
```

The `gene_client` variable in the code above *is* a `MyGeneInfo` object (exactly as obtained through the MyGene.py package). As such, all of the methods available in the MyGene.py client are available in the biothings_client generated gene client. So the above code block is equivalent to the way you use MyGene.py client package before:

```
In [1]: import mygene

In [2]: gene_client = mygene.MyGeneInfo()get_client('gene')

In [3]: type(gene_client)
Out[3]: mygene.MyGeneInfo
```

5.1.1 Use the client for MyGene.info API - Genes

Once you get the `gene_client` instance from biothings_client, the rest is exactly the same:

```
In [4]: gene_client.getgene('1017', fields='symbol,name')
Out[4]:
{'_id': '1017',
 '_score': 13.166854,
 'name': 'cyclin dependent kinase 2',
 'symbol': 'CDK2'}

In [5]: gene_client.getgenes(['1017', '1018'], species='human', fields='symbol,name')
querying 1-2...done.
Out[5]:
[{'_id': '1017',
  '_score': 20.773563,
  'name': 'cyclin dependent kinase 2',
  'query': '1017',
  'symbol': 'CDK2'},
 {'_id': '1018',
  '_score': 21.309164,
  'name': 'cyclin dependent kinase 3',
  'query': '1018',
  'symbol': 'CDK3'}]

In [6]: gene_client.query('uniprot:P24941', fields='symbol,name')
Out[6]:
{'hits': [{'_id': '1017',
 '_score': 14.752583,
 'name': 'cyclin dependent kinase 2',
 'symbol': 'CDK2'}],
 'max_score': 14.752583,
 'took': 3,
 'total': 1}

In [7]: gene_client.querymany(['P24941', 'O14727'], scopes='uniprot', fields='symbol,
↪name')
querying 1-2...done.
Finished.
Out[7]:
[{'_id': '1017',
  '_score': 14.176394,
  'name': 'cyclin dependent kinase 2',
  'query': 'P24941',
  'symbol': 'CDK2'},
{'_id': '317',
 '_score': 14.75709,
 'name': 'apoptotic peptidase activating factor 1',
 'query': 'O14727',
 'symbol': 'APAF1'}]

In [8]: gene_client.metadata()
Out[8]:
{'app_revision': 'c2a3aaa5fdac7b05fe243c1de62e6b3a3cf2b773',
 'available_fields': 'http://mygene.info/metadata/fields',
 'build_date': '2018-11-26T08:11:23.790634',
 'build_version': '20181126',
 'genome_assembly': {'frog': 'xenTro3',
  'fruitfly': 'dm3',
  'human': 'hg38',
  'mouse': 'mm10',
```

(continues on next page)

(continued from previous page)


```
 'nematode': 'ce10',
 'pig': 'susScr2',
 'rat': 'rn4',
 'zebrafish': 'zv9'},
'source': None,
'src': {'PantherDB': {'stats': {'PantherDB': 156054},
  'version': '2017-12-11'},
 'cpdb': {'stats': {'cpdb': 21141}, 'version': '33'},
 'ensembl': {'stats': {'ensembl_acc': 3228635,
   'ensembl_gene': 3187005,
   'ensembl_genomic_pos': 3183045,
   'ensembl_interpro': 2307500,
   'ensembl_pfam': 2100435,
   'ensembl_prosite': 1266847},
  'version': '94'},
 'ensembl_genomic_pos_hg19': {'stats': {'ensembl_genomic_pos_hg19': 55966},
  'version': None},
 'ensembl_genomic_pos_mm9': {'stats': {'ensembl_genomic_pos_mm9': 38646},
  'version': None},
 'entrez': {'stats': {'entrez_accession': 22406332,
   'entrez_gene': 22521690,
   'entrez_genomic_pos': 2632698,
   'entrez_go': 204359,
   'entrez_refseq': 22370423,
   'entrez_retired': 243656,
   'entrez_unigene': 543053},
  'version': '20181126'},
 'exac': {'stats': {'broadinstitute_exac': 18240}, 'version': '0.3.1'},
 'generif': {'stats': {'generif': 96431}, 'version': '20181126'},
 'homologene': {'stats': {'homologene': 269019}, 'version': '68'},
 'pharmgkb': {'stats': {'pharmgkb': 26833}, 'version': '2018-11-05'},
 'pharos': {'stats': {'pharos': 19828}, 'version': '5.2.0'},
 'reactome': {'stats': {'reactome': 71935}, 'version': '2018-09-24'},
 'reagent': {'stats': {'reagent': 38621}, 'version': None},
 'refseq': {'stats': {'entrez_ec': 19773, 'entrez_genesummary': 27713},
  'version': '91'},
 'reporter': {'stats': {'reporter': 426561}, 'version': None},
 'ucsc': {'stats': {'ucsc_exons': 208266}, 'version': '20181115'},
 'umls': {'stats': {'umls': 39665}, 'version': '2017-05-08'},
 'uniprot': {'stats': {'uniprot': 9411447}, 'version': '20181107'},
 'uniprot_ipi': {'stats': {'uniprot_ipi': 157025}, 'version': None},
 'uniprot_pdb': {'stats': {'uniprot_pdb': 30379}, 'version': '20181107'},
 'uniprot_pir': {'stats': {'uniprot_pir': 153446}, 'version': '20181107'},
 'wikipedia': {'stats': {'wikipedia': 11075}, 'version': None}},
'src_version': {'PantherDB': '2017-12-11',
 'cpdb': '33',
 'ensembl': '94',
 'ensembl_genomic_pos_hg19': None,
 'ensembl_genomic_pos_mm9': None,
 'entrez': '20181126',
 'exac': '0.3.1',
 'generif': '20181126',
 'homologene': '68',
 'pharmgkb': '2018-11-05',
 'pharos': '5.2.0',
 'reactome': '2018-09-24',
 'reagent': None,
```


(continues on next page)

(continued from previous page)

```
  'refseq': '91',
  'reporter': None,
  'ucsc': '20181115',
  'umls': '2017-05-08',
  'uniprot': '20181107',
  'uniprot_ipi': None,
  'uniprot_pdb': '20181107',
  'uniprot_pir': '20181107',
  'wikipedia': None},
 'stats': {'total_ensembl_genes': 24436578,
  'total_ensembl_genes_mapped_to_entrez': 1355996,
  'total_ensembl_only_genes': 1873808,
  'total_entrez_genes': 22521690,
  'total_genes': 24395498,
  'total_species': 23801},
 'taxonomy': {'frog': 8364,
  'fruitfly': 7227,
  'human': 9606,
  'mouse': 10090,
  'nematode': 6239,
  'pig': 9823,
  'rat': 10116,
  'thale-cress': 3702,
  'zebrafish': 7955}}
```

In addition to the `gene_client`, you can generate a client to any of the other BioThings API services we offer. See the following code snippet:

5.1.2 Use the client for MyVariant.info API - Variants

```
In [10]: variant_client = get_client('variant')

In [11]: variant_client.query('dbnsfp.genename:BTK', fields='_id')
Out[11]:
{'hits': [{'_id': 'chrX:g.100614336C>T', '_score': 10.192645},
  {'_id': 'chrX:g.100608911G>A', '_score': 10.192645},
  {'_id': 'chrX:g.100608917G>C', '_score': 10.192645},
  {'_id': 'chrX:g.100608872T>A', '_score': 10.192645},
  {'_id': 'chrX:g.100608887A>T', '_score': 10.192645},
  {'_id': 'chrX:g.100608891T>C', '_score': 10.192645},
  {'_id': 'chrX:g.100608282T>C', '_score': 10.192645},
  {'_id': 'chrX:g.100608230A>T', '_score': 10.192645},
  {'_id': 'chrX:g.100604881C>T', '_score': 10.192645},
  {'_id': 'chrX:g.100608204A>G', '_score': 10.192645}],
 'max_score': 10.192645,
 'took': 10,
 'total': 5143}
```

5.1.3 Use the client for MyChem.info API - Chemicals/Drugs

```
In [12]: chem_client = get_client('chem')

In [13]: chem_client.getchem('DB00551', fields='drugbank.name')
```

(continues on next page)

(continued from previous page)

```
Out[13]:
{'_id': 'RRUDCFGSUDOHDG-UHFFFAOYSA-N',
 'drugbank': {'_license': 'https://goo.gl/kvVASD',
  'name': 'Acetohydroxamic Acid'}}
```

5.1.4 Use the client for MyDisease.info API - Disease

```
In [13]: disease_client = get_client('disease')

In [14]: disease_client.query('diabetes')
Out[14]:
{'hits': [{'_id': 'MONDO:0005443',
   '_score': 3.466746,
   'mondo': {'label': 'type 2 diabetes nephropathy',
    'xrefs': {'efo': '0004997'}}},
  {'_id': 'MONDO:0023227',
   '_score': 3.466746,
   'mondo': {'definition': 'A form of diabetes insipidus that manifests during␣
↪pregnancy (or in some cases, after pregnancy). It is characterized by theappearance␣
↪of a polyuric-polydipsic syndrome that resultsin fluid intake ranging from 3 to 20␣
↪L/day. It is also charac-terized by excretion of abnormally high volumes of␣
↪dilutedurine. This polyuria is insipid, i.e., the urine concentrationof dissolved␣
↪substances is very low.',
    'label': 'gestational diabetes insipidus',
    'xrefs': {'gard': '0010702', 'mesh': 'C548014', 'umls': 'C2932666'}}},
  {'_id': 'MONDO:0001344',
   '_score': 3.466746,
   'mondo': {'label': 'obsolete neonatal diabetes mellitus'}},
  {'_id': 'MONDO:0019846',
   '_score': 3.4068294,
   'hpo': {'disease_name': 'Acquired central diabetes insipidus',
    'orphanet': '95626',
    'phenotype_related_to_disease': [{'aspect': 'P',
      'assigned_by': 'ORPHA:orphadata',
      'evidence': 'TAS',
      'frequency': 'HP:0040281',
      'hpo_id': 'HP:0000873'},
     {'aspect': 'P',
      'assigned_by': 'ORPHA:orphadata',
      'evidence': 'TAS',
      'frequency': 'HP:0040281',
      'hpo_id': 'HP:0001824'},
     {'aspect': 'P',
      'assigned_by': 'ORPHA:orphadata',
      'evidence': 'TAS',
      'frequency': 'HP:0040281',
      'hpo_id': 'HP:0001959'},
     {'aspect': 'P',
      'assigned_by': 'ORPHA:orphadata',
      'evidence': 'TAS',
      'frequency': 'HP:0040281',
      'hpo_id': 'HP:0100515'}]},
   'mondo': {'definition': 'Acquired central diabetes insipidus (acquired CDI) is a␣
↪subtype of central diabetes insipidus (CDI, see this term), characterized by␣
↪polyuria and polydipsia, due to an idiopathic or secondary decrease in vasopressin␣
↪(AVP) production.',
```

(continues on next page)

(continued from previous page)

```
    'label': 'acquired central diabetes insipidus',
    'xrefs': {'icd10': 'E23.2', 'orphanet': '95626'}}},
  {'_id': 'MONDO:0022650',
   '_score': 3.2161584,
   'mondo': {'label': 'cardiomyopathy diabetes deafness',
    'xrefs': {'gard': '0001103'}}},
  {'_id': 'MONDO:0005442',
   '_score': 3.1703691,
   'mondo': {'label': 'type 1 diabetes nephropathy',
    'xrefs': {'efo': '0004996'}}},
  {'_id': 'MONDO:0015967',
   '_score': 3.1703691,
   'mondo': {'definition': 'Rare genetic diabetes mellitus.',
    'label': 'rare genetic diabetes mellitus',
    'xrefs': {'orphanet': '183625'}}},
  {'_id': 'MONDO:0022971',
   '_score': 3.1703691,
   'mondo': {'label': 'diabetes persistent mullerian ducts',
    'xrefs': {'gard': '0001840'}}},
  {'_id': 'MONDO:0022993',
   '_score': 3.1703691,
   'mondo': {'definition': 'Diabetes insipidus caused by excessive intake of water␣
↪due to psychological factors or damage to the thirst-regulating mechanism.',
    'label': 'dipsogenic diabetes insipidus',
    'xrefs': {'gard': '0010703',
     'mesh': 'C548013',
     'ncit': 'C129735',
     'sctid': '82800008',
     'umls': 'C0268813'}}},
  {'_id': 'MONDO:0015888',
   '_score': 3.1530147,
   'mondo': {'label': 'other rare diabetes mellitus',
    'xrefs': {'orphanet': '181381'}}}],
 'max_score': 3.466746,
 'took': 17,
 'total': 120}
```

5.1.5 Use the client for t.biothings.io API - Taxnomy

```
In [15]: taxon_client = get_client('taxon')

In [16]: taxon_client.gettaxon(9606)
Out[16]:
{'_id': '9606',
 '_version': 1,
 'authority': ['homo sapiens linnaeus, 1758'],
 'common_name': 'man',
 'genbank_common_name': 'human',
 'has_gene': True,
 'lineage': [9606,
  9605,
  207598,
  9604,
  314295,
  9526,
```

(continues on next page)

(continued from previous page)

```
  314293,
  376913,
  9443,
  314146,
  1437010,
  9347,
  32525,
  40674,
  32524,
  32523,
  1338369,
  8287,
  117571,
  117570,
  7776,
  7742,
  89593,
  7711,
  33511,
  33213,
  6072,
  33208,
  33154,
  2759,
  131567,
  1],
 'parent_taxid': 9605,
 'rank': 'species',
 'scientific_name': 'homo sapiens',
 'taxid': 9606,
 'uniprot_name': 'homo sapiens'}
```

5.2 Instantiating a Client for a Custom BioThings API

The Quick Start tutorial shows how to get clients for all publicly available BioThings APIs. What if you create your own custom BioThings API?

With biothings_client, you can generate client settings for any future APIs created with BioThings API. The BioThings API Single source tutorial explains how to set up a simple BioThings API from PharmGKB gene data. The following code snippet shows an example of how to setup biothings_client to access that custom API:

```
In [1]: from biothings_client import get_client

In [2]: pharmgkb_client = get_client('gene', url='http://35.164.95.182:8000/v1')

In [3]: pharmgkb_client.query('ncbi_gene_id:1017', fields='pharmgkb_accession_id')
Out[3]:
{'hits': [{'_id': 'cOydWmcBViFqgJfo4gdM',
   '_score': 7.495912,
   'pharmgkb_accession_id': 'PA101'}],
 'max_score': 7.495912,
 'took': 10,
 'total': 1}
```

The **url** parameter to **get_client** specifies where the BioThings API endpoint is located (the address above is temporary

and is no longer serviced by us). The entity parameter is still 'gene' (as the entity type of PharmGKB gene is gene).

5.3 Subclassing a Client

The biothings_client **get_client** function generates settings for a biothings client on-the-fly. However, sometimes a user may want to subclass an existing client. This can be accomplished by use of the **instance** parameter in the **get_client** function. Normally the **get_client** function returns an instantiated client object, however if **instance** is False, **get_client** returns the class itself (allowing it to be used as a parent class). See the following code snippet for an example:

```
In [1]: class CustomMyGeneClient(get_client('gene', instance=False)):
   ...:     def getgene(self, _id, fields=None, **kwargs):
   ...:         ''' overridden gene function '''
   ...:         r =  super(CustomMyGeneClient, self).getgene(_id=_id, fields=fields,␣
↪**kwargs)
   ...:         r['custom-field'] = 'My custom field'
   ...:         return r
   ...:

In [2]: gene_client = CustomMyGeneClient()

In [3]: gene_client.getgene('1017', fields='symbol,name')
Out[3]:
{'_id': '1017',
 '_score': 13.409985,
 'custom-field': 'My custom field',
 'name': 'cyclin dependent kinase 2',
 'symbol': 'CDK2'}
```

Here we overrode the normal functioning of the **getgene** function in the MyGene client by adding a custom field to the result.

5.4 API Documentation

Detailed documentation of the biothings_client package can be found on this page.

5.4.1 get_client

`biothings_client.`**`get_client`**(*biothing_type=None*, *instance=True*, **args*, ***kwargs*)

Function to return a new python client for a Biothings API service.

Parameters

- **`biothing_type`** – the type of biothing client, currently one of: 'gene', 'variant', 'taxon', 'chem', 'disease'
- **`instance`** – if True, return an instance of the derived client, if False, return the class of the derived client

All other args/kwargs are passed to the derived client instantiation (if applicable)

5.4.2 MyGeneInfo

class biothings_client.**MyGeneInfo**(*url=None*)

clear_cache()
Clear the globally installed cache.

findgenes(*id_li*, ***kwargs*)
Deprecated since version 2.0.0.

Use `querymany()` instead. It's kept here as an alias of `querymany()` method.

get_fields(*search_term=None*, *verbose=True*)
Return all available fields can be return from MyGene.info services.

This is a wrapper for http://mygene.info/metadata/fields

Parameters **search_term** – an optional string to search (case insensitive) for matching field names. If not provided, all available fields will be returned.

Example:

```
>>> mv.get_fields()
>>> mv.get_fields("uniprot")
>>> mv.get_fields("refseq")
>>> mv.get_fields("kegg")
```

Hint: This is useful to find out the field names you need to pass to **fields** parameter of other methods.

getgene(*_id*, *fields=None*, ***kwargs*)
Return the gene object for the give geneid. This is a wrapper for GET query of "/gene/<geneid>" service.

Parameters

- **geneid** – entrez/ensembl gene id, entrez gene id can be either a string or integer
- **fields** – fields to return, a list or a comma-separated string. If **fields="all"**, all available fields are returned
- **species** – optionally, you can pass comma-separated species names or taxonomy ids
- **email** – optionally, pass your email to help us to track usage
- **filter** – alias for **fields** parameter

Returns a gene object as a dictionary, or None if geneid is not valid.

Ref http://docs.mygene.info/en/latest/doc/data.html for available fields, extra *kwargs* and more.

Example:

```
>>> mg.getgene(1017, email='abc@example.com')
>>> mg.getgene('1017', fields='symbol,name,entrezgene,refseq')
>>> mg.getgene('1017', fields='symbol,name,entrezgene,refseq.rna')
>>> mg.getgene('1017', fields=['symbol', 'name', 'pathway.kegg'])
>>> mg.getgene('ENSG00000123374', fields='all')
```

Hint: The supported field names passed to **fields** parameter can be found from any full gene object (when **fields="all"**). Note that field name supports dot notation for nested data structure as well, e.g. you can

pass "refseq.rna" or "pathway.kegg".

---

getgenes(*ids*, *fields=None*, ***kwargs*)

Return the list of gene objects for the given list of geneids. This is a wrapper for POST query of "/gene" service.

Parameters

- **geneids** – a list/tuple/iterable or comma-separated entrez/ensembl gene ids
- **fields** – fields to return, a list or a comma-separated string. If **fields="all"**, all available fields are returned
- **species** – optionally, you can pass comma-separated species names or taxonomy ids
- **email** – optionally, pass your email to help us to track usage
- **filter** – alias for fields
- **as_dataframe** – if True, return object as DataFrame (requires Pandas).
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of gene objects or a pandas DataFrame object (when **as_dataframe** is True)

Ref http://mygene.info/doc/annotation_service.html for available fields, extra *kwargs* and more.

Example:

```
>>> mg.getgenes([1017, '1018','ENSG00000148795'], email='abc@example.com')
>>> mg.getgenes([1017, '1018','ENSG00000148795'], fields="entrezgene,uniprot")
>>> mg.getgenes([1017, '1018','ENSG00000148795'], fields="all")
>>> mg.getgenes([1017, '1018','ENSG00000148795'], as_dataframe=True)
```

Hint: A large list of more than 1000 input ids will be sent to the backend web service in batches (1000 at a time), and then the results will be concatenated together. So, from the user-end, it's exactly the same as passing a shorter list. You don't need to worry about saturating our backend servers.

---

metadata(*verbose=True*, ***kwargs*)

Return a dictionary of MyGene.info metadata.

Example:

```
>>> metadata = mg.metadata
```

query(*q*, ***kwargs*)

Return the query result. This is a wrapper for GET query of "/query?q=<query>" service.

Parameters

- **q** – a query string, detailed query syntax here
- **fields** – fields to return, a list or a comma-separated string. If **fields="all"**, all available fields are returned
- **species** – optionally, you can pass comma-separated species names or taxonomy ids. Default: human,mouse,rat.
- **size** – the maximum number of results to return (with a cap of 1000 at the moment). Default: 10.

- **skip** – the number of results to skip. Default: 0.
- **sort** – Prefix with "-" for descending order, otherwise in ascending order. Default: sort by matching scores in decending order.
- **entrezonly** – if True, return only matching entrez genes, otherwise, including matching Ensemble-only genes (those have no matching entrez genes).
- **email** – optionally, pass your email to help us to track usage
- **as_dataframe** – if True, return object as DataFrame (requires Pandas).
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.
- **fetch_all** – if True, return a generator to all query results (unsorted). This can provide a very fast return of all hits from a large query. Server requests are done in blocks of 1000 and yielded individually. Each 1000 block of results must be yielded within 1 minute, otherwise the request will expire on the server side.

Returns a dictionary with returned gene hits or a pandas DataFrame object (when **as_dataframe** is True)

Ref http://mygene.info/doc/query_service.html for available fields, extra *kwargs* and more.

Example:

```
>>> mg.query('cdk2')
>>> mg.query('reporter:1000_at')
>>> mg.query('symbol:cdk2', species='human')
>>> mg.query('symbol:cdk*', species=10090, size=5, as_dataframe=True)
>>> mg.query('q=chrX:151073054-151383976', species=9606)
```

querymany(*qterms*, *scopes=None*, ***kwargs*)

Return the batch query result. This is a wrapper for POST query of "/query" service.

Parameters

- **qterms** – a list/tuple/iterable of query terms, or a string of comma-separated query terms.
- **scopes** – type of types of identifiers, either a list or a comma-separated fields to specify type of input qterms, e.g. "entrezgene", "entrezgene,symbol", ["ensemblgene", "symbol"]. Refer to official MyGene.info docs for full list of fields.
- **fields** – fields to return, a list or a comma-separated string. If **fields="all"**, all available fields are returned
- **species** – optionally, you can pass comma-separated species names or taxonomy ids. Default: human,mouse,rat.
- **entrezonly** – if True, return only matching entrez genes, otherwise, including matching Ensemble-only genes (those have no matching entrez genes).
- **returnall** – if True, return a dict of all related data, including dup. and missing qterms
- **verbose** – if True (default), print out infomation about dup and missing qterms
- **email** – optionally, pass your email to help us to track usage
- **as_dataframe** – if True, return object as DataFrame (requires Pandas).
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of gene objects or a pandas DataFrame object (when **as_dataframe** is True)

Ref http://mygene.info/doc/query_service.html for available fields, extra *kwargs* and more.

Example:

```
>>> mg.querymany(['DDX26B', 'CCDC83'], scopes='symbol', species=9606)
>>> mg.querymany(['1255_g_at', '1294_at', '1316_at', '1320_at'], scopes=
↪'reporter')
>>> mg.querymany(['NM_003466', 'CDK2', 695, '1320_at', 'Q08345'],
...              scopes='refseq,symbol,entrezgene,reporter,uniprot', species=
↪'human')
>>> mg.querymany(['1255_g_at', '1294_at', '1316_at', '1320_at'], scopes=
↪'reporter',
...              fields='ensembl.gene,symbol', as_dataframe=True)
```

Hint: `querymany()` is perfect for doing id mappings.

Hint: Just like `getgenes()`, passing a large list of ids (>1000) to `querymany()` is perfectly fine.

set_caching(*cache_db=None*, *verbose=True*, ***kwargs*)

Installs a local cache for all requests.

cache_db is the path to the local sqlite cache database.

stop_caching()

Stop caching.

5.4.3 MyVariantInfo

class biothings_client.**MyVariantInfo**(*url=None*)

clear_cache()

Clear the globally installed cache.

format_hgvs(*chrom*, *pos*, *ref*, *alt*)

get a valid hgvs name from VCF-style "chrom, pos, ref, alt" data. Example:

```
>>> utils.variant.format_hgvs("1", 35366, "C", "T")
>>> utils.variant.format_hgvs("2", 17142, "G", "GA")
>>> utils.variant.format_hgvs("MT", 8270, "CACCCCCTCT", "C")
>>> utils.variant.format_hgvs("X", 107930849, "GGA", "C")
```

get_fields(*search_term=None*, *verbose=True*)

Wrapper for http://myvariant.info/v1/metadata/fields

Parameters

- **search_term** – a case insensitive string to search for in available field names. If not provided, all available fields will be returned.
- **assembly** – return the metadata for either hg19 or hg38 variants, "hg19" (default) or "hg38".

Example:

```
>>> mv.get_fields()
>>> mv.get_fields("rsid")
>>> mv.get_fields("sift")
```

Hint: This is useful to find out the field names you need to pass to **fields** parameter of other methods.

get_hgvs_from_vcf(*input_vcf*)

From the input VCF file (filename or file handle), return a generator of genomic based HGVS ids. :param input_vcf: input VCF file, can be a filename or a file handle :returns: a generator of genomic based HGVS ids. To get back a list instead,

using `list(get_hgvs_from_vcf("your_vcf_file"))`

Note: This is a lightweight VCF parser to return valid genomic-based HGVS ids from the *input_vcf* file. For more sophisticated VCF parser, consider using PyVCF module.

getvariant(*_id*, *fields=None*, ***kwargs*)

Return the variant object for the give HGVS-based variant id. This is a wrapper for GET query of "/variant/<hgvsid>" service.

Parameters

- **vid** – an HGVS-based variant id. More about HGVS id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.
- **assembly** – specify the human genome assembly used in HGVS-based variant id, "hg19" (default) or "hg38".

Returns a variant object as a dictionary, or None if vid is not found.

Example:

```
>>> mv.getvariant('chr9:g.107620835G>A')
>>> mv.getvariant('chr9:g.107620835G>A', fields='dbnsfp.genename')
>>> mv.getvariant('chr9:g.107620835G>A', fields=['dbnsfp.genename', 'cadd.
↪phred'])
>>> mv.getvariant('chr9:g.107620835G>A', fields='all')
>>> mv.getvariant('chr1:g.161362951G>A', assembly='hg38')
```

Hint: The supported field names passed to **fields** parameter can be found from any full variant object (without **fields**, or **fields="all"**). Note that field name supports dot notation for nested data structure as well, e.g. you can pass "dbnsfp.genename" or "cadd.phred".

getvariants(*ids*, *fields=None*, ***kwargs*)

Return the list of variant annotation objects for the given list of hgvs-base varaint ids. This is a wrapper for POST query of "/variant" service.

Parameters

- **ids** – a list/tuple/iterable or a string of comma-separated HGVS ids. More about hgvs id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.

- **assembly** – specify the human genome assembly used in HGVS-based variant id, "hg19" (default) or "hg38".
- **as_generator** – if True, will yield the results in a generator.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of variant objects or a pandas DataFrame object (when **as_dataframe** is True)

Ref http://docs.myvariant.info/en/latest/doc/variant_annotation_service.html.

Example:

```
>>> vars = ['chr1:g.866422C>T',
...         'chr1:g.876664G>A',
...         'chr1:g.69635G>C',
...         'chr1:g.69869T>A',
...         'chr1:g.881918G>A',
...         'chr1:g.865625G>A',
...         'chr1:g.69892T>C',
...         'chr1:g.879381C>T',
...         'chr1:g.878330C>G']
>>> mv.getvariants(vars, fields="cadd.phred")
>>> mv.getvariants('chr1:g.876664G>A,chr1:g.881918G>A', fields="all")
>>> mv.getvariants(['chr1:g.876664G>A', 'chr1:g.881918G>A'], as_
↪dataframe=True)
>>> mv.getvariants(['chr1:g.161362951G>A', 'chr2:g.51032181G>A'], assembly=
↪'hg38')
```

Hint: A large list of more than 1000 input ids will be sent to the backend web service in batches (1000 at a time), and then the results will be concatenated together. So, from the user-end, it's exactly the same as passing a shorter list. You don't need to worry about saturating our backend servers.

Hint: If you need to pass a very large list of input ids, you can pass a generator instead of a full list, which is more memory efficient.

metadata(*verbose=True*, ***kwargs*)

Return a dictionary of MyVariant.info metadata.

Parameters **assembly** – return the metadata for either hg19 or hg38 variants, "hg19" (default) or "hg38".

Example:

```
>>> metadata = mv.metadata()
>>> metadata = mv.metadata(assembly='hg38')
```

query(*q*, ***kwargs*)

Return the query result. This is a wrapper for GET query of "/query?q=<query>" service.

Parameters

- **q** – a query string, detailed query syntax here.

- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.
- **assembly** – specify the human genome assembly used for the query, "hg19" (default) or "hg38".
- **size** – the maximum number of results to return (with a cap of 1000 at the moment). Default: 10.
- **skip** – the number of results to skip. Default: 0.
- **sort** – Prefix with "-" for descending order, otherwise in ascending order. Default: sort by matching scores in decending order.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **fetch_all** – if True, return a generator to all query results (unsorted). This can provide a very fast return of all hits from a large query. Server requests are done in blocks of 1000 and yielded individually. Each 1000 block of results must be yielded within 1 minute, otherwise the request will expire at server side.

Returns a dictionary with returned variant hits or a pandas DataFrame object (when **as_dataframe** is True) or a generator of all hits (when **fetch_all** is True)

Ref http://docs.myvariant.info/en/latest/doc/variant_query_service.html.

Example:

```
>>> mv.query('_exists_:dbsnp AND _exists_:cosmic')
>>> mv.query('dbnsfp.polyphen2.hdiv.score:>0.99 AND chrom:1')
>>> mv.query('cadd.phred:>50')
>>> mv.query('dbnsfp.genename:CDK2', size=5)
>>> mv.query('dbnsfp.genename:CDK2', size=5, assembly='hg38')
>>> mv.query('dbnsfp.genename:CDK2', fetch_all=True)
>>> mv.query('chrX:151073054-151383976')
```

Hint: By default, **query** method returns the first 10 hits if the matched hits are >10. If the total number of hits are less than 1000, you can increase the value for **size** parameter. For a query returns more than 1000 hits, you can pass "fetch_all=True" to return a generator of all matching hits (internally, those hits are requested from the server-side in blocks of 1000).

querymany (*qterms*, *scopes=None*, ***kwargs*)

Return the batch query result. This is a wrapper for POST query of "/query" service.

Parameters

- **qterms** – a list/tuple/iterable of query terms, or a string of comma-separated query terms.
- **scopes** – specify the type (or types) of identifiers passed to **qterms**, either a list or a comma-separated fields to specify type of input qterms, e.g. "dbsnp.rsid", "clinvar.rcv_accession", ["dbsnp.rsid", "cosmic.cosmic_id"]. See here for full list of supported fields.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.
- **assembly** – specify the human genome assembly used for the query, "hg19" (default) or "hg38".
- **returnall** – if True, return a dict of all related data, including dup. and missing qterms

- **verbose** – if True (default), print out information about dup and missing qterms
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of matching variant objects or a pandas DataFrame object.

Ref http://docs.myvariant.info/en/latest/doc/variant_query_service.html for available fields, extra *kwargs* and more.

Example:

```
>>> mv.querymany(['rs58991260', 'rs2500'], scopes='dbsnp.rsid')
>>> mv.querymany(['rs58991260', 'rs2500'], scopes='dbsnp.rsid', assembly='hg38
↪')
>>> mv.querymany(['RCV000083620', 'RCV000083611', 'RCV000083584'], scopes=
↪'clinvar.rcv_accession')
>>> mv.querymany(['COSM1362966', 'COSM990046', 'COSM1392449'], scopes='cosmic.
↪cosmic_id', fields='cosmic')
>>> mv.querymany(['COSM1362966', 'COSM990046', 'COSM1392449'], scopes='cosmic.
↪cosmic_id',
...              fields='cosmic.tumor_site', as_dataframe=True)
```

Hint: `querymany()` is perfect for query variants based different ids, e.g. rsid, clinvar ids, etc.

Hint: Just like `getvariants()`, passing a large list of ids (>1000) to `querymany()` is perfectly fine.

Hint: If you need to pass a very large list of input qterms, you can pass a generator instead of a full list, which is more memory efficient.

set_caching(*cache_db=None*, *verbose=True*, ***kwargs*)
Installs a local cache for all requests.

cache_db is the path to the local sqlite cache database.

stop_caching()
Stop caching.

5.4.4 MyChemInfo

class biothings_client.**MyChemInfo**(*url=None*)

clear_cache()
Clear the globally installed cache.

get_fields(*search_term=None*, *verbose=True*)
Wrapper for http://mychem.info/v1/metadata/fields

Parameters **search_term** – a case insensitive string to search for in available field names. If not provided, all available fields will be returned.

Example:

```
>>> mc.get_fields()
>>> mc.get_fields("pubchem")
>>> mc.get_fields("drugbank.targets")
```

Hint: This is useful to find out the field names you need to pass to **fields** parameter of other methods.

getchem(*_id*, *fields=None*, ***kwargs*)

Return the chemical/drug object for the give id. This is a wrapper for GET query of "/chem/<chem_id>" service.

Parameters

- **_id** – a chemical/drug id, supports InchiKey, Drugbank ID, ChEMBL ID, ChEBI ID, PubChem CID and UNII. More about chemical/drug id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.

Returns a chemical/drug object as a dictionary, or None if _id is not found.

Example:

```
>>> mc.getchem("ZRALSGWEFCBTJO-UHFFFAOYSA-N")
>>> mc.getchem("DB00553", fields="chebi.name,drugbank.id,pubchem.cid")
>>> mc.getchem("CHEMBL1308", fields=["chebi.name", "drugbank.id", "pubchem.cid
↪"])
>>> mc.getchem("7AXV542LZ4", fields="unii")
>>> mc.getchem("CHEBI:6431", fields="chembl.smiles")
```

Hint: The supported field names passed to **fields** parameter can be found from any full chemical/drug object (without **fields**, or **fields="all"**). Note that field name supports dot notation for nested data structure as well, e.g. you can pass "drugbank.id" or "chembl.smiles".

getchems(*ids*, *fields=None*, ***kwargs*)

Return the list of chemical/drug annotation objects for the given list of chemical/drug ids. This is a wrapper for POST query of "/chem" service.

Parameters

- **ids** – a list/tuple/iterable or a string of comma-separated chem/drug ids. More about chem/drug id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.
- **as_generator** – if True, will yield the results in a generator.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of variant objects or a pandas DataFrame object (when **as_dataframe** is True)

Ref http://docs.mychem.info/en/latest/doc/chem_annotation_service.html.

Example:

```
>>> chems = [
...     "KTUFNOKKBVMGRW-UHFFFAOYSA-N",
...     "HXHWSAZORRCQMX-UHFFFAOYSA-N",
...     "DQMZLTXERSFNPB-UHFFFAOYSA-N"
... ]
>>> mc.getchems(chems, fields="pubchem")
>>> mc.getchems('KTUFNOKKBVMGRW-UHFFFAOYSA-N,DB00553', fields="all")
>>> mc.getchems(chems, fields='chembl', as_dataframe=True)
```

Hint: A large list of more than 1000 input ids will be sent to the backend web service in batches (1000 at a time), and then the results will be concatenated together. So, from the user-end, it's exactly the same as passing a shorter list. You don't need to worry about saturating our backend servers.

Hint: If you need to pass a very large list of input ids, you can pass a generator instead of a full list, which is more memory efficient.

getdrug(*_id*, *fields=None*, ***kwargs*)

Return the object given id. This is a wrapper for GET query of the biothings annotation service.

Parameters

- **_id** – an entity id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.

Returns an entity object as a dictionary, or None if _id is not found.

getdrugs(*ids*, *fields=None*, ***kwargs*)

Return the list of annotation objects for the given list of ids. This is a wrapper for POST query of the biothings annotation service.

Parameters

- **ids** – a list/tuple/iterable or a string of ids.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **as_generator** – if True, will yield the results in a generator.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of objects or a pandas DataFrame object (when **as_dataframe** is True)

Hint: A large list of more than 1000 input ids will be sent to the backend web service in batches (1000 at a time), and then the results will be concatenated together. So, from the user-end, it's exactly the same as passing a shorter list. You don't need to worry about saturating our backend servers.

Hint: If you need to pass a very large list of input ids, you can pass a generator instead of a full list, which is more memory efficient.

metadata (*verbose=True*, ***kwargs*)

Return a dictionary of MyChem.info metadata, a wrapper for http://mychem.info/v1/metadata

Example:

```
>>> metadata = mv.metadata()
```

query (*q*, ***kwargs*)

Return the query result. This is a wrapper for GET query of "/query?q=<query>" service.

Parameters

- **q** – a query string, detailed query syntax here.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.
- **size** – the maximum number of results to return (with a cap of 1000 at the moment). Default: 10.
- **skip** – the number of results to skip. Default: 0.
- **sort** – Prefix with "-" for descending order, otherwise in ascending order. Default: sort by matching scores in decending order.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **fetch_all** – if True, return a generator to all query results (unsorted). This can provide a very fast return of all hits from a large query. Server requests are done in blocks of 1000 and yielded individually. Each 1000 block of results must be yielded within 1 minute, otherwise the request will expire at server side.

Returns a dictionary with returned variant hits or a pandas DataFrame object (when **as_dataframe** is True) or a generator of all hits (when **fetch_all** is True)

Ref http://docs.mychem.info/en/latest/doc/chem_query_service.html.

Example:

```
>>> mc.query('drugbank.name:monobenzone')
>>> mc.query('drugbank.targets.uniprot:P07998')
>>> mc.query('drugbank.targets.uniprot:P07998 AND _exists_:unii')
>>> mc.query('chebi.mass:[300 TO 500]')
>>> mc.query('sider.side_effect.name:anaemia', size=5)
>>> mc.query('sider.side_effect.name:anaemia', fetch_all=True)
```

Hint: By default, **query** method returns the first 10 hits if the matched hits are >10. If the total number of hits are less than 1000, you can increase the value for **size** parameter. For a query returns more than 1000 hits, you can pass "fetch_all=True" to return a generator of all matching hits (internally, those hits are requested from the server-side in blocks of 1000).

querymany (*qterms*, *scopes=None*, ***kwargs*)

Return the batch query result. This is a wrapper for POST query of "/query" service.

Parameters

- **qterms** – a list/tuple/iterable of query terms, or a string of comma-separated query terms.
- **scopes** – specify the type (or types) of identifiers passed to **qterms**, either a list or a comma-separated fields to specify type of input qterms, e.g. "dbsnp.rsid", "clinvar.rcv_accession", ["dbsnp.rsid", "cosmic.cosmic_id"]. See here for full list of supported fields.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned. See here for all available fields.
- **returnall** – if True, return a dict of all related data, including dup. and missing qterms
- **verbose** – if True (default), print out information about dup and missing qterms
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of matching variant objects or a pandas DataFrame object.

Ref http://docs.myvariant.info/en/latest/doc/variant_query_service.html for available fields, extra *kwargs* and more.

Example:

```
>>> mc.querymany(["ZRALSGWEFCBTJO-UHFFFAOYSA-N", "RRUDCFGSUDOHDG-UHFFFAOYSA-N
↪"])
>>> mc.querymany(["DB00536", 'DB00533'], scopes='drugbank.id')
>>> mc.querymany(["CHEBI:95222", 'CHEBI:45924', 'CHEBI:33325'], scopes='chebi.
↪id')
>>> mc.querymany(["CHEMBL1555813", 'CHEMBL22', 'CHEMBL842'], scopes='chembl.
↪molecule_chembl_id')
>>> mc.querymany(["DB00536", '4RZ82L2GY5'], scopes='drugbank.id,unii.unii')
>>> mc.querymany(["DB00536", '4RZ82L2GY5'], scopes=['drugbank.id', 'unii.unii
↪'])
>>> mc.querymany(["DB00536", '4RZ82L2GY5'], scopes=['drugbank.id', 'unii.unii
↪'], fields='drugbank,unii')
>>> mc.querymany(["DB00536", '4RZ82L2GY5'], scopes=['drugbank.id', 'unii.unii
↪'],
...              fields='drugbank.name,unii',as_dataframe=True)
```

Hint: `querymany()` is perfect for query variants based different ids, e.g. rsid, clinvar ids, etc.

Hint: Just like `getvariants()`, passing a large list of ids (>1000) to `querymany()` is perfectly fine.

Hint: If you need to pass a very large list of input qterms, you can pass a generator instead of a full list, which is more memory efficient.

set_caching(*cache_db=None*, *verbose=True*, ***kwargs*)

Installs a local cache for all requests.

cache_db is the path to the local sqlite cache database.

stop_caching()
Stop caching.

5.4.5 MyDiseaseInfo

class biothings_client.**MyDiseaseInfo**(*url=None*)

clear_cache()
Clear the globally installed cache.

get_fields(*search_term=None*, *verbose=True*)
Wrapper for /metadata/fields

search_term is a case insensitive string to search for in available field names. If not provided, all available fields will be returned.

Hint: This is useful to find out the field names you need to pass to **fields** parameter of other methods.

getdisease(*_id*, *fields=None*, ***kwargs*)
Return the object given id. This is a wrapper for GET query of the biothings annotation service.

Parameters

- **_id** – an entity id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.

Returns an entity object as a dictionary, or None if _id is not found.

getdiseases(*ids*, *fields=None*, ***kwargs*)
Return the list of annotation objects for the given list of ids. This is a wrapper for POST query of the biothings annotation service.

Parameters

- **ids** – a list/tuple/iterable or a string of ids.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **as_generator** – if True, will yield the results in a generator.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of objects or a pandas DataFrame object (when **as_dataframe** is True)

Hint: A large list of more than 1000 input ids will be sent to the backend web service in batches (1000 at a time), and then the results will be concatenated together. So, from the user-end, it's exactly the same as passing a shorter list. You don't need to worry about saturating our backend servers.

Hint: If you need to pass a very large list of input ids, you can pass a generator instead of a full list, which is more memory efficient.

metadata(*verbose=True*, ***kwargs*)

Return a dictionary of Biothing metadata.

query(*q*, ***kwargs*)

Return the query result. This is a wrapper for GET query of biothings query service.

Parameters

- **q** – a query string.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **size** – the maximum number of results to return (with a cap of 1000 at the moment). Default: 10.
- **skip** – the number of results to skip. Default: 0.
- **sort** – Prefix with "-" for descending order, otherwise in ascending order. Default: sort by matching scores in decending order.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **fetch_all** – if True, return a generator to all query results (unsorted). This can provide a very fast return of all hits from a large query. Server requests are done in blocks of 1000 and yielded individually. Each 1000 block of results must be yielded within 1 minute, otherwise the request will expire at server side.

Returns a dictionary with returned variant hits or a pandas DataFrame object (when **as_dataframe** is True) or a generator of all hits (when **fetch_all** is True)

Hint: By default, **query** method returns the first 10 hits if the matched hits are >10. If the total number of hits are less than 1000, you can increase the value for **size** parameter. For a query that returns more than 1000 hits, you can pass "fetch_all=True" to return a generator of all matching hits (internally, those hits are requested from the server in blocks of 1000).

querymany(*qterms*, *scopes=None*, ***kwargs*)

Return the batch query result. This is a wrapper for POST query of "/query" service.

Parameters

- **qterms** – a list/tuple/iterable of query terms, or a string of comma-separated query terms.
- **scopes** – specify the type (or types) of identifiers passed to **qterms**, either a list or a comma-separated fields to specify type of input qterms.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **returnall** – if True, return a dict of all related data, including dup. and missing qterms
- **verbose** – if True (default), print out information about dup and missing qterms
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json

- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of matching objects or a pandas DataFrame object.

Hint: Passing a large list of ids (>1000) to `querymany()` is perfectly fine.

Hint: If you need to pass a very large list of input qterms, you can pass a generator instead of a full list, which is more memory efficient.

---

set_caching(*cache_db=None*, *verbose=True*, ***kwargs*)

Installs a local cache for all requests.

cache_db is the path to the local sqlite cache database.

stop_caching()

Stop caching.

5.4.6 MyTaxonInfo

class biothings_client.**MyTaxonInfo**(*url=None*)

clear_cache()

Clear the globally installed cache.

get_fields(*search_term=None*, *verbose=True*)

Wrapper for /metadata/fields

search_term is a case insensitive string to search for in available field names. If not provided, all available fields will be returned.

Hint: This is useful to find out the field names you need to pass to **fields** parameter of other methods.

---

gettaxa(*ids*, *fields=None*, ***kwargs*)

Return the list of annotation objects for the given list of ids. This is a wrapper for POST query of the biothings annotation service.

Parameters

- **ids** – a list/tuple/iterable or a string of ids.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **as_generator** – if True, will yield the results in a generator.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of objects or a pandas DataFrame object (when **as_dataframe** is True)

Hint: A large list of more than 1000 input ids will be sent to the backend web service in batches (1000 at a time), and then the results will be concatenated together. So, from the user-end, it's exactly the same as passing a shorter list. You don't need to worry about saturating our backend servers.

Hint: If you need to pass a very large list of input ids, you can pass a generator instead of a full list, which is more memory efficient.

gettaxon(*_id*, *fields=None*, ***kwargs*)

Return the object given id. This is a wrapper for GET query of the biothings annotation service.

Parameters

- **_id** – an entity id.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.

Returns an entity object as a dictionary, or None if _id is not found.

metadata(*verbose=True*, ***kwargs*)

Return a dictionary of Biothing metadata.

query(*q*, ***kwargs*)

Return the query result. This is a wrapper for GET query of biothings query service.

Parameters

- **q** – a query string.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **size** – the maximum number of results to return (with a cap of 1000 at the moment). Default: 10.
- **skip** – the number of results to skip. Default: 0.
- **sort** – Prefix with "-" for descending order, otherwise in ascending order. Default: sort by matching scores in decending order.
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **fetch_all** – if True, return a generator to all query results (unsorted). This can provide a very fast return of all hits from a large query. Server requests are done in blocks of 1000 and yielded individually. Each 1000 block of results must be yielded within 1 minute, otherwise the request will expire at server side.

Returns a dictionary with returned variant hits or a pandas DataFrame object (when **as_dataframe** is True) or a generator of all hits (when **fetch_all** is True)

Hint: By default, **query** method returns the first 10 hits if the matched hits are >10. If the total number of hits are less than 1000, you can increase the value for **size** parameter. For a query that returns more than 1000 hits, you can pass "fetch_all=True" to return a generator of all matching hits (internally, those hits are requested from the server in blocks of 1000).

querymany(*qterms*, *scopes=None*, ***kwargs*)

Return the batch query result. This is a wrapper for POST query of "/query" service.

Parameters

- **qterms** – a list/tuple/iterable of query terms, or a string of comma-separated query terms.
- **scopes** – specify the type (or types) of identifiers passed to **qterms**, either a list or a comma-separated fields to specify type of input qterms.
- **fields** – fields to return, a list or a comma-separated string. If not provided or **fields="all"**, all available fields are returned.
- **returnall** – if True, return a dict of all related data, including dup. and missing qterms
- **verbose** – if True (default), print out information about dup and missing qterms
- **as_dataframe** – if True or 1 or 2, return object as DataFrame (requires Pandas). True or 1: using json_normalize 2 : using DataFrame.from_dict otherwise: return original json
- **df_index** – if True (default), index returned DataFrame by 'query', otherwise, index by number. Only applicable if as_dataframe=True.

Returns a list of matching objects or a pandas DataFrame object.

Hint: Passing a large list of ids (>1000) to `querymany()` is perfectly fine.

Hint: If you need to pass a very large list of input qterms, you can pass a generator instead of a full list, which is more memory efficient.

---

set_caching(*cache_db=None*, *verbose=True*, ***kwargs*)

Installs a local cache for all requests.

cache_db is the path to the local sqlite cache database.

stop_caching()

Stop caching.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

biothings_client, 18

Index

B

biothings_client (*module*), 18

C

clear_cache() (*biothings_client.MyChemInfo method*), 26
clear_cache() (*biothings_client.MyDiseaseInfo method*), 31
clear_cache() (*biothings_client.MyGeneInfo method*), 19
clear_cache() (*biothings_client.MyTaxonInfo method*), 33
clear_cache() (*biothings_client.MyVariantInfo method*), 22

F

findgenes() (*biothings_client.MyGeneInfo method*), 19
format_hgvs() (*biothings_client.MyVariantInfo method*), 22

G

get_client() (*in module biothings_client*), 18
get_fields() (*biothings_client.MyChemInfo method*), 26
get_fields() (*biothings_client.MyDiseaseInfo method*), 31
get_fields() (*biothings_client.MyGeneInfo method*), 19
get_fields() (*biothings_client.MyTaxonInfo method*), 33
get_fields() (*biothings_client.MyVariantInfo method*), 22
get_hgvs_from_vcf() (*biothings_client.MyVariantInfo method*), 23
getchem() (*biothings_client.MyChemInfo method*), 27
getchems() (*biothings_client.MyChemInfo method*), 27
getdisease() (*biothings_client.MyDiseaseInfo method*), 31
getdiseases() (*biothings_client.MyDiseaseInfo method*), 31
getdrug() (*biothings_client.MyChemInfo method*), 28
getdrugs() (*biothings_client.MyChemInfo method*), 28
getgene() (*biothings_client.MyGeneInfo method*), 19
getgenes() (*biothings_client.MyGeneInfo method*), 20
gettaxa() (*biothings_client.MyTaxonInfo method*), 33
gettaxon() (*biothings_client.MyTaxonInfo method*), 34
getvariant() (*biothings_client.MyVariantInfo method*), 23
getvariants() (*biothings_client.MyVariantInfo method*), 23

M

metadata() (*biothings_client.MyChemInfo method*), 29
metadata() (*biothings_client.MyDiseaseInfo method*), 32
metadata() (*biothings_client.MyGeneInfo method*), 20
metadata() (*biothings_client.MyTaxonInfo method*), 34
metadata() (*biothings_client.MyVariantInfo method*), 24
MyChemInfo (*class in biothings_client*), 26
MyDiseaseInfo (*class in biothings_client*), 31
MyGeneInfo (*class in biothings_client*), 19
MyTaxonInfo (*class in biothings_client*), 33
MyVariantInfo (*class in biothings_client*), 22

Q

query() (*biothings_client.MyChemInfo method*), 29
query() (*biothings_client.MyDiseaseInfo method*), 32
query() (*biothings_client.MyGeneInfo method*), 20

query() (*biothings_client.MyTaxonInfo method*), 34
query() (*biothings_client.MyVariantInfo method*), 24
querymany() (*biothings_client.MyChemInfo method*), 29
querymany() (*biothings_client.MyDiseaseInfo method*), 32
querymany() (*biothings_client.MyGeneInfo method*), 21
querymany() (*biothings_client.MyTaxonInfo method*), 34
querymany() (*biothings_client.MyVariantInfo method*), 25

S

set_caching() (*biothings_client.MyChemInfo method*), 30
set_caching() (*biothings_client.MyDiseaseInfo method*), 33
set_caching() (*biothings_client.MyGeneInfo method*), 22
set_caching() (*biothings_client.MyTaxonInfo method*), 35
set_caching() (*biothings_client.MyVariantInfo method*), 26
stop_caching() (*biothings_client.MyChemInfo method*), 30
stop_caching() (*biothings_client.MyDiseaseInfo method*), 33
stop_caching() (*biothings_client.MyGeneInfo method*), 22
stop_caching() (*biothings_client.MyTaxonInfo method*), 35
stop_caching() (*biothings_client.MyVariantInfo method*), 26