
BioServices

Release 1.10.0

Thomas Cokelaer, Lea M. Harder, Jordi Serra-Musach, Dennis Pul

Jul 01, 2022

CONTENTS

| | | |
|----------|----------------------------|------------|
| 1 | Contributors | 3 |
| 2 | Quick example | 5 |
| 3 | Current services | 7 |
| 4 | Bioservices command | 9 |
| 5 | Changelog | 11 |
| 6 | Installation | 13 |
| 6.1 | User guide | 13 |
| | Python Module Index | 219 |
| | Index | 221 |

**Python_version_available**

BioServices is tested for Python 3.6, 3.7, 3.8, 3.9

Contributions

Please join <https://github.com/cokelaer/bioservices> and share your notebooks <https://github.com/bioservices/notebooks/>

Issues

Please use <https://github.com/cokelaer/bioservices/issues>

How to cite

Cokelaer et al. *BioServices: a common Python package to access biological Web Services programmatically* *Bioinformatics* (2013) 29 (24): 3241-3242

Documentation

[RTD documentation.](#)

Bioservices is a Python package that provides access to many Bioinformatics Web Services (e.g., UniProt) and a framework to easily implement Web Services wrappers (based on WSDL/SOAP or REST protocols).

The primary goal of **BioServices** is to use Python as a glue language to provide a programmatic access to several Bioinformatics Web Services. By doing so, elaboration of new applications that combine several of the wrapped Web Services is fostered.

One of the main philosophy of **BioServices** is to make use of the existing biological databases (not to re-invent new databases) and to alleviate the needs for expertise in Web Services for the developers/users.

BioServices provides access to about 40 Web Services.

CONTRIBUTORS

Maintaining BioServices would not have been possible without users and contributors. Each contribution has been an encouragement to pursue this project. Thanks to all:

QUICK EXAMPLE

Here is a small example using the UniProt Web Service to search for the zap70 specy in human organism:

```
>>> from bioservices import UniProt
>>> u = UniProt(verbose=False)
>>> data = u.search("zap70+and+taxonomy:9606", frmt="tab", limit=3,
...                 columns="entry name,length,id, genes")
>>> print(data)
```

| Entry name | Length | Entry | Gene names |
|--------------|--------|--------|---------------|
| ZAP70_HUMAN | 619 | P43403 | ZAP70 SRK |
| B4E0E2_HUMAN | 185 | B4E0E2 | |
| RHOH_HUMAN | 191 | Q15669 | RHOH ARHH TTF |

More examples and tutorials are available in the [On-line documentation](#)

CURRENT SERVICES

Here is the list of services available and their testing status.

| Service | CI testing |
|----------------|------------|
| arrayexpress | |
| bigg | |
| biocarta | |
| biocontainers | |
| biodbnet | |
| biogrid | |
| biomart | |
| biomodels | |
| chebi | |
| chembl | |
| cog | |
| dbfetch | |
| ena | |
| ensembl | |
| eutils | |
| eva | |
| hgnc | |
| intact_complex | |
| kegg | |
| muscle | |
| mygeneinfo | |
| ncbiblast | |
| omicsdi | |
| omnipath | |
| panther | |
| pathwaycommons | |
| pdb | |
| pdbe | |
| pfam | |
| pride | |
| psicquic | |
| pubchem | |
| quickgo | |
| reactome | |
| rhea | |

continues on next page

Table 1 – continued from previous page

| Service | CI testing |
|-------------|------------|
| rnaseq_ebi | |
| seqret | |
| unichem | |
| uniprot | |
| wikipathway | |

Note: Contributions to implement new wrappers are more than welcome. See [BioServices github page](#) to join the development, and the Developer guide on how to implement new wrappers.

BIOSERVICES COMMAND

In version 1.8.2, we included a bioservices command. For now it has only one subcommand to download a NCBI accession number:

```
bioservices download-accession --accession K01711.1
```


CHANGELOG

| Version | Description |
|---------|--|
| 1.10.0 | <ul style="list-style-type: none">• Update uniprot to use the new API (june 2022) |
| 1.9.0 | <ul style="list-style-type: none">• Update unichem to reflect new API |
| 1.8.4 | <ul style="list-style-type: none">• biomodels: Fix #208• KEGG: fixed #204 #202 and #203 |
| 1.8.3 | <ul style="list-style-type: none">• Eutils: remove warning due to unreachable URL. Set REST as attribute rather and inheritance.• NEW biocontainers module• KEGG: add save_pathway method. Fix parsing of structure/pdb entry• remove deprecated function from Reactome |
| 1.8.2 | <ul style="list-style-type: none">• Fix suds package in code and requirements |
| 1.8.1 | <ul style="list-style-type: none">• Integrated a change made in KEGG service (DEFINITON was changed to ORG_CODE)• for developers: applied black on all modules• switch suds-jurko to new suds community |
| 1.8.0 | <ul style="list-style-type: none">• add main standalone application.• moved chemspider and clinvitae to the attic• removed picr service, not active anymore |

Warning: Some of the services may be down. BioServices developers are not responsible for the maintenance or failure of underlying services. Generally speaking (and by experience) the services are up most of the time but failure may occur because the site is under maintenance or too many requests have been sent. Another common reason is the fact that the API of the web services has changed: If so, BioServices need to be updated. You may contribute or report such API changes on our [Issue](#) page

INSTALLATION

BioServices is available on [PyPi](#), the Python package repository. The following command should install **BioServices** and its dependencies automatically provided you have **pip** on your system:

```
pip install bioservices
```

If not, please see the external [pip installation page](#) or [pip installation](#) entry. You may also find information in the [troubleshootings page](#) section about known issues.

Regarding the dependencies, **BioServices** depends on the following packages: **BeautifulSoup4** (for parsing XML), **SOAPpy** and **suds** (to access to SOAP/WSDL services; suds is used by ChEBI only for which SOAPpy fails to correctly fetch the service) and **easydev**. All those packages should be installed automatically when using **pip** installer. Since version 1.6.0, we also make use of **pandas** and **matplotlib** to offer some extra functionalities.

6.1 User guide

6.1.1 Quick Start

Introduction

BioServices provides access to several Web Services. Each service requires some expertise on its own. In this Quick Start section, we will neither cover all the services nor all their functionalities. However, it should give you a good overview of what you can do with **BioServices** (both from the user and developer point of views).

Before starting, let us remind what are Web Services. There provide an access to databases or applications via a web interface based on the SOAP/WSDL or the REST technologies. These technologies allow a programmatic access, which we take advantage in **BioServices**.

The REST technology uses URLs so there is no external dependency. You simply need to build a well-formatted URL and you will retrieve an XML document that you can consume with your preferred technology platform.

The SOAP/WSDL technology combines SOAP (Simple Object Access Protocol), which is a messaging protocol for transporting information and the WSDL (Web Services Description Language), which is a method for describing Web Services and their capabilities.

What methods are available for a given service

Usually most of the service functionalities have been wrapped and we try to keep the names as close as possible to the API. On top of the service methods, each class inherits from the `BioService` class (REST or WSDL). For instance REST service have the useful request method. Another nice function is the `onWeb`.

See also:

[REST](#), [WSDLService](#)

What about the output ?

Outputs depend on the service and functionalities of the service. It can be heterogeneous. However, output are mostly XML formatted or in tabulated separated column format (TSV). When XML is returned, it is usually parsed via the BeautifulSoup package (for instance you can get all children using `getchildren()` function). Sometimes, we also convert output into dictionaries. So, it really depends on the service/functionality you are using.

Let us look at some of the Web Services wrapped in **BioServices**.

UniProt service

Let us start with the `UniProt` class. With this class, you can access to uniprot services. In particular, you can map an ID from a database to another one. For instance to convert the UniProtKB ID into KEGG ID, use:

```
>>> from bioservices.uniprot import UniProt
>>> u = UniProt(verbose=False)
>>> u.mapping(fr="ACC+ID", to="KEGG_ID", query='P43403')
{'P43403': ['hsa:7535']}
```

Note that the returned response from uniprot web service is converted into a list. The first two elements are the databases used for the mapping. Then, alternance of the queried element and the answer populates the list.

You can also search for a specific UniProtKB ID to get exhaustive information:

```
>>> print(u.search("P43403", frmt="txt"))
ID   ZAP70_HUMAN                Reviewed;              619 AA.
AC   P43403; A6NFP4; Q6PIA4; Q8IXD6; Q9UBS6;
DT   01-NOV-1995, integrated into UniProtKB/Swiss-Prot.
DT   01-NOV-1995, sequence version 1.
...
```

To obtain the FASTA sequence, you can use `searchUniProtId()`:

```
>>> res = u.searchUniProtId("P09958", frmt="xml")
>>> print(u.searchUniProtId("P09958", frmt="fasta"))
sp|P09958|FURIN_HUMAN Furin OS=Homo sapiens GN=FURIN PE=1 SV=2
MELRPWLLWVVAATGTLVLLAADAQQQKVFNTWAVRIPGGPAVANSVARKHGFLNLGQI
FGDYHFWHRGVTKRSLSPHRPHSRLQREPQVQWLEQQVAKRRTRKRDVYQEPDTPKFPQ
QWYLSGVTQRDLNVKAAWAQGYTGHGIVVSILDDGIEKNHPDLAGNYDPGASFVNDQDP
DPQPRYTQMNDNRHGTRCAGEVA AVANNGVCGVG VAYNARIGGVRMLDGEVTD AVEARSL
GLNPNHIHIYSASWGPEDDGKTV DGPRLAE EAFR GVSQGRGGLGSIFVWASGNGGREH
DSCNCDGYTNSIYTLSSISATQFGNVPWYSEACSTLATTYSSGNQNEKQIVTTDLRQKC
TESHTGTSASAPLAAGIIALTLEANKNLTWRDMQHLVVQTSKPAHLNANDWATNGVGRKV
SHSYGYGLLDAGAMVALAQNWTTPVAPQRKCIIDILTEPKDIGKRLEVRKTVTACLGEPNH
```

(continues on next page)

(continued from previous page)

```
ITRLEHAQARLTLSYNRRGDLAIHLVSPMGTRSTLLAARPHDYSADGFNDWAFMTTHSWD
EDPSGEWVLEIENTSEANNYGTLTKFTLVLYGTAPEGLPVPPESSGCKTLTSSQACVVCE
EGFSLHQBKSCVQHCPPGFAPQVLDTHYSTENDVETIRASVCAPCHASCATCQGPALTDCL
SCPSHASLDPVEQTCRSRQSQRSPPPQQPPRLPPEVEAGQRLRAGLLPSHLPEVVAGL
SCAFIVLVFVTVFLVLQLRSGFSFRGVKYVTMDRGLISYKGLPPEAWQEECPDSEDEG
RGERTAFIKDQSAL
```

See also:

Reference guide of *bioservices.uniprot.UniProt* for more details

KEGG service

The KEGG interface is similar but contains more methods. The tutorial presents the KEGG interface in details, but let us have a quick overview. First, let us start a KEGG instance:

```
from bioservices import KEGG
k = KEGG(verbose=False)
```

KEGG contains biological data for many organisms. By default, no organism is set, which can be checked in the following attribute

```
k.organism
```

We can set it to human using KEGG terminology for homo sapiens:

```
k.organis = 'hsa'
```

You can use the *dbinfo()* to obtain statistics on the **pathway** database:

```
>>> print(k.info("pathway"))
pathway      KEGG Pathway Database
path         Release 65.0+/01-15, Jan 13
              Kanehisa Laboratories
              218,277 entries
```

You can see the list of valid databases using the *databases* attribute. Each of the database entry can also be listed using the *list()* method. For instance, the organisms can be retrieved with:

```
k.list("organism")
```

However, to extract the IDs extra processing is required. So, we provide aliases to retrieve the organism IDs easily:

```
k.organismIds
```

The human organism is coded as “hsa”. You can also get its T number instead:

```
>>> k.code2Tnumber("hsa")
'T01001'
```

Every element is referred to with a KEGG ID, which may be difficult to handle at first. There are methods to retrieve the IDs though. For instance, get the list of pathways IDs for the current organism as follows:

```
k.pathwayIds
```

For a given gene, you can get the full information related to that gene by using the method `get()`:

```
print(k.get("hsa:3586"))
```

or a pathway:

```
print(k.get("path:hsa05416"))
```

See also:

Reference guide of *bioservices.kegg.KEGG* for more details

See also:

KEGG Tutorial for more details

See also:

Reference guide of *bioservices.kegg.KEGGParser* to parse a KEGG entry into a dictionary

QuickGO

To access the GO interface, simply create an instance and look for an entry using the `bioservices.quickgo.QuickGO.Term()` method:

```
>>> from bioservices import QuickGO
>>> g = QuickGO(verbose=False)
>>> print(g.Term("GO:0003824", frmt="obo"))
[Term]
id: GO:0003824
name: catalytic activity
def: "Catalysis of a biochemical reaction at physiological temperatures. In
biologically catalyzed reactions, the reactants are known as substrates, and the
catalysts are naturally occurring macromolecular substances known as enzymes.
Enzymes possess specific binding sites for substrates, and are usually composed
wholly or largely of protein, but RNA that has catalytic activity (ribozyme) is
often also regarded as enzymatic."
synonym: "enzyme activity" exact
xref: InterPro:IPR000183
...
```

See also:

Reference guide of *bioservices.quickgo.QuickGO* for more details

PICR service

PICR, the Protein Identifier Cross Reference service provides 2 services in WSDL and REST protocols. When it is the case, we arbitrary chose one of the available protocol. In the PICR case, we implemented only the REST interface. The methods available in the REST service are very similar to those available via SOAP except for one major difference: only one accession or sequence can be mapped per request.

The following example returns a XML document containing information about the protein P29375 found in two specific databases:

```
>>> from bioservices.picr import PICR
>>> p = PICR()
>>> res = p.getUPIForAccession("P29375", ["IPI", "ENSEMBL"])
```

See also:

Reference guide of `bioservices.picr.PICR` for more details

BioModels service

You can access the biomodels service and obtain a model as follows:

```
>>> from bioservices import biomodels
>>> b = biomodels.BioModels()
>>> model = b.get_model('BIOMD0000000299')
```

Then you can play with the SBML file with your favorite SBML tool.

In order to get the model IDs, you can look at the full list:

```
>>> b.get_models()
```

See also:

Reference guide of `bioservices.biomodels.BioModels` for more details

See also:

[Biomodels tutorial](#) for more details

Rhea service

Create a [Rhea](#) instance as follows:

```
from bioservices import Rhea
r = Rhea()
```

Rhea provides only 2 type of requests with a REST interface that are available with the `search()` and `query()` methods. Let us first find information about the chemical product **caffeine** using the `search()` method:

```
response = r.search("caffeine*")
```

The output is a JSON file that we convert in BioServices into a Pandas dataframe.

The previous request returns more than 10,000 entries. Here are the first two entries:

| Reaction identifier | Equation | |
|---------------------|---|----------------------------------|
| ↪ ChEBI name | Cross-reference (KEGG) | Cross-reference (Reactome) |
| 0 RHEA:47148 | a ubiquinone + caffeine + H2O = | 1,3,7-trimethy... ↪ |
| ↪ MetaCyc:RXN-11523 | KEGG:R07980 | NaN |
| 1 RHEA:10280 | 1,7-dimethylxanthine + S-adenosyl-L-methionine... | ↪ |
| ↪ MetaCyc:RXN-7601 | KEGG:R07921 | NaN |

The second method provided is the `query()` method. Given an Id, you can query the Rhea database using Id found earlier (e.g., 10280). This is finally a filtering method as compared to the search method. If you know what you are looking for (the rhea-id) use this method instead of the search method:

```
info = r.query("10280", columns="rhea-id,equation", limit=10)
```

See also:

Reference guide of [bioservices.rhea.Rhea](#) for more details

Other services

There are many other services provided within **BioServices** and the reference guide should give you all the information available with examples to start to play with any of them. The home page of the services themselves is usually a good starting point as well.

Services that are not available in **BioServices** can still be accessed to quite easily as demonstrated in the [Developer Guide](#) section.

6.1.2 Tutorials

This section presents the KEGG and BioModels services in more details. The Protein test case study illustrates how several services can be used to get lots of information about a specific protein. New Contributions to this section are welcomed.

Contents

- [KEGG Tutorial](#)
 - [Introduction](#)
 - [Searching for an organism](#)
 - [Look for pathways \(by name\)](#)
 - [Look for pathway \(by genes i.e., IDs or usual name\)](#)
 - [Introspecting a pathway](#)
 - [Building a histogram of all relations in human pathways](#)

KEGG Tutorial

Introduction

Start a kegg interface (default organism is human, that is called **hsa**):

```
from bioservices.kegg import KEGG
k = KEGG()
```

KEGG has many databases. The list can be found in the attribute `bioservices.kegg.KEGG.databases`. Each database can be queried with the `bioservices.kegg.KEGG.list()` method:

```
k.list("organism")
```

The output contains Id of the organism and some other information. To retrieve the Ids, you will need to process the output. However, we provide an alias:

```
print(k.organismIds)
```

In general, methods require an access to the on-line KEGG database therefore it takes time. For instance, the command above takes a couple of seconds. However, some are buffered so next time you call it, it will be much faster.

Another useful alias is the **pathwayIds** to retrieve all pathway Ids. However, you must first specify the organism you are interested in. From the command above we know that **hsa** (human) is valid organism Id, so let us set it and then get the list of pathways:

```
k.organism = "hsa"
k.pathwayIds
```

Another function provided by the KEGG API is the `bioservices.kegg.KEGG.get()` one that query a specific entry. Here we are interested into the human gene with the code 7535:

```
k.get("hsa:7535")    #hsa:7535 is also known as ZAP70
```

It is quite verbose and is a single string, which may be tricky to handle. We provide a tool to ease the parsing (see below and `bioservices.kegg.KEGG.parse()`) returned by `bioservices.kegg.KEGG.parse()`.

Searching for an organism

The method `bioservices.kegg.KEGG.find()` is quite convenient to search for entries in different database. For instance, if you want to know the code of an entry for the gene called ZAP70 in the human organism, type:

```
>>> s.find("hsa", "zap70")
'hsa:7535\tZAP70, SRK, STCD, STD, TZK, ZAP-70; zeta-chain (TCR) associated protein,
↪kinase 70kDa (EC:2.7.10.2); K07360 tyrosine-protein kinase ZAP-70 [EC:2.7.10.2]\n'
```

It is quite powerful and more examples will be shown. However, it has some limitations. For example, what about searching for the organism Ids that correspond to any *Drosophila*? It does not look like it is possible. BioServices provides a method to search for an organism Id using `lookfor_organism()` given the name (or part of it):

```
>>> k.lookfor_organism("droso")
['T00030 dme Drosophila melanogaster (fruit fly) Eukaryotes;Animals;Arthropods;Insects',
'T01032 dpo Drosophila pseudoobscura pseudoobscura Eukaryotes;Animals;Arthropods;Insects
↪',
```

(continues on next page)

(continued from previous page)

```
'T01059 dan Drosophila ananassae Eukaryotes;Animals;Arthropods;Insects',
'T01060 der Drosophila erecta Eukaryotes;Animals;Arthropods;Insects',
'T01063 dpe Drosophila persimilis Eukaryotes;Animals;Arthropods;Insects',
'T01064 dse Drosophila sechellia Eukaryotes;Animals;Arthropods;Insects',
'T01065 dsi Drosophila simulans Eukaryotes;Animals;Arthropods;Insects',
'T01067 dwi Drosophila willistoni Eukaryotes;Animals;Arthropods;Insects',
'T01068 dya Drosophila yakuba Eukaryotes;Animals;Arthropods;Insects',
'T01061 dgr Drosophila grimshawi Eukaryotes;Animals;Arthropods;Insects',
'T01062 dmo Drosophila mojavensis Eukaryotes;Animals;Arthropods;Insects',
'T01066 dvi Drosophila virilis Eukaryotes;Animals;Arthropods;Insects']
```

Look for pathways (by name)

Searching for pathways is quite similar. You can use the **find** method as above:

```
>>> print(s.find("pathway", "B+cell"))
path:map04112    Cell cycle - Caulobacter
path:map04662    B cell receptor signaling pathway
path:map05100    Bacterial invasion of epithelial cells
path:map05120    Epithelial cell signaling in Helicobacter pylori infection
path:map05217    Basal cell carcinoma
```

Note that without the + sign, you get all pathway that contains *B* or *cell*. Yet, we have 5 results, which do not necessarily fit our request. Alternatively you can use one of BioServices method:

```
>>> k.lookfor_pathway("B cell")
['path:map04662 B cell receptor signaling pathway']
```

You can also search for a pathway knowing some gene names but first we need to introspect the pathway to get the genes IDs.

Look for pathway (by genes i.e., IDs or usual name)

Imagine you want to find the pathway that contains **ZAP70**. As we have seen earlier you can get its gene Id as follows:

```
>>> s.find("hsa", "zap70")
hsa:7535
```

The following commands do not help:

```
>>> s.find("pathway", "zap70")
>>> s.find("pathway", "hsa:7535")
>>> s.find("pathway", "7535")
```

We provide a method to search for pathways that contain the required gene Id. You can search by KEGG Id or gene name:

```
>>> res = s.get_pathway_by_gene("7535", "hsa")
>>> s.get_pathway_by_gene("zap70", "hsa")
['path:hsa04064', 'path:hsa04650', 'path:hsa04660', 'path:hsa05340']
```

This commands first search for the gene Id in the KEGG database and then parse the output to retrieve the pathways.

Introspecting a pathway

Let us focus on one pathway (**path:hsa04660**). You can use the `get()` command to obtain information about the pathway.

```
print(s.get("hsa04660"))
```

The output is a single string where you can recognise different fields such as NAME, GENE, DESCRIPTION and so on. This is quite limited. In BioServices, we provide a convenient parser that converts the output of the previous command into a dictionary:

```
>>> s = KEGG()
>>> data = s.get("hsa04660")
>>> dict_data = s.parse(data)
>>> print(dict_data['GENE'])
'10000': 'AKT3; v-akt murine thymoma viral oncogene homolog 3 (protein kinase B, gamma)
↳ [KO:K04456] [EC:2.7.11.1]',
'10125': 'RASGRP1; RAS guanyl releasing protein 1 (calcium and DAG-regulated) [KO:K04350]
↳ ',
'1019': 'CDK4; cyclin-dependent kinase 4 [KO:K02089] [EC:2.7.11.22]',
...
```

This is fine if we just want the name of the genes but what about their relations? Actually, there is an option with the `get` method where you can specify the output format. In particular you can request the pathway to be returned as a kgml file:

```
res = s.get("hsa04660", "kgml")
```

This file can be parsed to extract the relations. We provide a tool to do that:

```
res = s.parse_kgml_pathway("hsa04660")
```

The variable returned is a dictionary with 2 keys: “entries” and “relations”.

You can extract the relations as follows:

```
res['relations']
```

It is a list of relations, each relation being a dictionary:

```
>>> res['relations'][0]
{'entry1': u'61',
 'entry2': u'63',
 'link': u'PPrel',
 'name': u'binding/association',
 'value': u'---'}
```

Here entry1 and 2 are Ids. The Ids can be found in

```
res['entries']
```

From there you should consult `bioservices.kegg.KEGG.parse_kgml_pathway()` and the KEGG document for more information. You may also look at `bioservices.kegg.KEGG.pathway2sif()` method that extract only protein-protein interactions with activation and inhibition links only.


Building a histogram of all relations in human pathways

Scanning all relations of the Human organism takes about 5-10 minutes. You can look at a subset by setting Nmax to a small value (e.g., Nmax=10).

```
from pylab import *
# extract all relations from all pathways
from bioservices.kegg import KEGG
s = KEGG()
s.organism = "hsa"

# retrieve more than 260 pathways so it takes time
results = [s.parse_kgml_pathway(x) for x in s.pathwayIds]
relations = [x['relations'] for x in results]

hist([len(r) for r in relations], 20)
xlabel('number of relations')
ylabel('\#')
title("number of relations per pathways")
grid(True)
```



all_relations.png

You can then extract more information such as the type of relations:

```
>>> # scan all relations looking for the type of relations
>>> import collections # for python 2.7.0 and above

>>> # we extract from all pathways, all relations, where we retrieve the type of
>>> # relation (name)
>>> data = list(flatten([x['name'] for x in rel] for rel in relations]))

>>> counter = collections.Counter(data)
>>> print(counter)
Counter({u'compound': 5235, u'activation': 3265, u'binding/association': 1087,
u'phosphorylation': 940, u'inhibition': 672, u'indirect effect': 559,
u'expression': 542, u'dephosphorylation': 93, u'missing interaction': 80,
u'dissociation': 78, u'ubiquitination': 48, u'state change': 24, u'repression':
12, u'methylation': 2})
```

See also:

bioservices.biomodels.BioModels for the full reference guide.

Biomodels tutorial

Start a biomodels interface:

```
>>> from bioservices import BioModels
>>> s = BioModels()
```

look at the list of model identifiers:

```
models = s.get_all_models()
```

If you have a specific model identifier, then it is easy. You can retrieve the model itself:

```
model = s.get_model("BIOMD0000000100")
```

and get its name or other types of information:

```
>>> model['name']
Rozi2003_GlycogenPhosphorylase_Activation
```

In particular, description, author and files associated with this model. Here, we can see the files and in particular a PNG image called **BIOMD0000000100.png**. You can get it as follows:

```
s.get_model_download("BIOMD0000000100", filename="BIOMD0000000100.png")
```

or just download the whole bundle:

```
s.get_model_download("BIOMD0000000100")
```

saved into **BIOMD0000000100.zip**.

Protein test case study

Application: retrieving information about a given protein

This section uses BioServices to demonstrate the interest of combining several services together within a single framework using the Python language as a glue language.

In this tutorial we are interested in using **BioServices** to obtain information about a specific protein. Let us focus on ZAP70 protein (homo sapiens).

Get a unique identifier and gene names from a name

From **Uniprot**, we can obtain the unique accession number of ZAP70, which may be useful later on. Let us try to use the `search()` method:

```
>>> from bioservices import *
>>> u = UniProt(verbose=False)
>>> u.search("ZAP70_HUMAN") # could be lower case
```

The default format of the returned answer is “tabulated”:

```
>>> res = u.search("ZAP70_HUMAN", frmt="tab")
>>> print(res)
Entry   Entry name   Status   Protein names   Gene names   Organism   Length
P43403  ZAP70_HUMAN   reviewed   Tyrosine-protein kinase ZAP-70 (EC 2.7.10.2) (70 kDa,
↳ zeta-chain associated protein) (Syk-related tyrosine kinase)   ZAP70 SRK Homo sapiens,
↳ (Human)       619
```

It is better, but let us simplify even further. In **BioServices**, the output of the tabulated format contains several columns but we can select only a subset such as the Entry (accession number) and the gene names, which are coded as “id” and “genes” in uniprot database:

```
>>> res = u.search("ZAP70_HUMAN", frmt="tab", columns="id,genes")
>>> print(res)
Entry   Gene names
P43403  ZAP70 SRK
```

So here we got the Entry P43403. Entry and Gene names can be saved in two variables as follows:

```
>>> res = u.search("ZAP70_HUMAN", frmt="tab", columns="id,genes")
>>> entry, gene_names = res.split("\n")[1].split("\t")
```

Getting the fasta sequence

It is then straightforward to obtain the FASTA sequence of ZAP70 using another method from the UniProt class called `retrieve()`:

```
>>> sequence = u.retrieve("P43403", "fasta")
>>> print(sequence)
>sp|P43403|ZAP70_HUMAN Tyrosine-protein kinase ZAP-70 OS=Homo sapiens GN=ZAP70 PE=1 SV=1
MPDPAAHLPPFFYGSISRAEAEHLKLAGMADGLFLLRQCLRS�GGYVLSLVHDVRFHHFP
IERQLNGTYAIAGGKAHCGPAELCEFYSRDPDGLPCNLRKPCNRPSGLEPQPGVFDCLRD
AMVRDYVRQTWKLEGEALEQAIISQAPQVEKLIATTAHERMPWYHSSLTREEAERKLYSG
AQTDGKFLLRPRKEQGTAYALSLIYGKTVYHYLISQDKAGKYCIPEGTKFDTLWQLVEYLK
LKADGLIYCLKEACPNSSASNASGAAAPTLPAHPSTLTHPQRRIDTLNSDGYTPEPARIT
SPDKPRPMPMDTSVYESPYSDPEELKDKKLFLKRDNLLIADIELGCGNFGSVRQGVYRMR
KKQIDVAIKVLKQGTEKADTEEMMREAQIMHQLDNPIVRLIGVCQAEALMLVMEMAGGG
PLHKFLVGKREEIPVSNVAELLHQVSMGMKYLEEKNFVHRDLAARNVLLVNRHYAKISDF
GLSKALGADDSYYTARSAGWPLKWYAPECINFRKFSSRSDVWSYGVMTWEALSYGQKPY
KKMKGPEVMAFIEQGKRMECPPECPPELYALMSDCWIYKWEDRPDLTVEQRMACYYSL
ASKVEGPPGSTQKAEAAACA
```

Note: There are many services that provides access to the FASTA sequence. We chose uniprot but you could use the Entrez utilities as well as other services.

Using BLAST on the sequence

You can then analyse this sequence with your favorite tool. As an example, within **BioServices** you can use NCIBlast but first let us extract the sequence itself (without the header):

```
sequence = sequence.split("\n", 1)[1].strip("\n")
```

then,

```
>>> s = NCIBlast(verbose=False)
>>> jobid = s.run(program="blastp", sequence=sequence, stype="protein", \
...     database="uniprotkb", email="cokelaer@ebi.ac.uk")
>>> print(s.getResult(jobid, "out")[0:1000])
BLASTP 2.2.26 [Sep-21-2011]
```

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.

Query= EMBOSS_001
(619 letters)

Database: uniprotkb
32,727,302 sequences; 10,543,978,207 total letters

(continues on next page)

(continued from previous page)

```
Searching.....done
```

| Sequences producing significant alignments: | Score (bits) | E Value |
|---|-----------------|------------|
| SP:ZAP70_HUMAN P43403 Tyrosine-protein kinase ZAP-70 OS=Homo sap... | 1279 | 0.0 |
| TR:H2QIE3_PANTR H2QIE3 Tyrosine-protein kinase OS=Pan troglodyte... | 1278 | 0.0 |
| TR:G3QGN8_GORGO G3QGN8 Tyrosine-protein kinase OS=Gorilla gorill... | 1278 | 0.0 |
| TR:G1QLX3_NOMLE G1QLX3 Tyrosine-protein kinase OS=Nomascus leuco... | 1249 | 0.0 |
| TR:F6SWY7_CALJA F6SWY7 Tyrosin | | |

The last command waits for the job to be finished before printing the results, which may be quite long. We could look at the beginning of the reported results and select only HUMAN sequences to see that the best sequence found is indeed ZAP70_HUMAN as expected:

```
>>> [x for x in s.getResult(jobid, "out").split("\n") if "HUMAN" in x]
['SP:ZAP70_HUMAN P43403 Tyrosine-protein kinase ZAP-70 OS=Homo sap... 1279 0.0 ',
 'SP:KSYK_HUMAN P43405 Tyrosine-protein kinase SYK OS=Homo sapiens... 691 0.0 ',
 'TR:A8K4G2_HUMAN A8K4G2 Tyrosine-protein kinase OS=Homo sapiens P... 691 0.0 ',
 ...]
```

Searching for relevant pathways

The KEGG services provides pathways, so let try to find pathways that contains our targetted protein. First we need to know the KEGG Id that corresponds to ZAP70. We can use the **find** method form KEGG service:

```
>>> from bioservices import KEGG
>>> k = KEGG(verbose=False)
>>> k.find("hsa", "zap70") # "hsa" stands for homo sapiens
hsa:7535 ZAP70, SRK, STCD, STD, TZK, ZAP-70; zeta-chain (TCR) associated protein kinase,
↪70kDa (EC:2.7.10.2); K07360 tyrosine-protein kinase ZAP-70 [EC:2.7.10.2]
```

There are other ways to perform this conversion using the `bioservices.uniprot.UniProt.mapping()` or `bioservices.KEGG.conv()` methods (e.g., `textit{k.conv("hsa", "up:P43403")}`).

Now, let us get the pathways that contains this ID:

```
>>> k.get_pathway_by_gene("7535", "hsa")
{'hsa04064': 'NF-kappa B signaling pathway',
 'hsa04650': 'Natural killer cell mediated cytotoxicity',
 'hsa04660': 'T cell receptor signaling pathway',
 'hsa05340': 'Primary immunodeficiency'}
```

We can look at the first pathway in a browser (highlighting the ZAP70 node):

```
>>> k.show_pathway("hsa04064", keggid={"7535": "red"})
```

Searching for binary Interactions

For this purpose, we could use PSICQUIC services to find the interactions that involve ZAP70 in the **mint** database:

```
>>> from bioservices import PSICQUIC
>>> s = PSICQUIC(verbose=False)
>>> data = s.query("mint", "ZAP70 AND species:9606")
```

where 9606 is the taxonomy Id for homo sapiens. We could also figure out how many interactions could be found in each database for this particular query:

```
>>> s.getInteractionCounter("zap70 AND species:9606")
{'apid': 82,
 'bar': 0,
 'bind': 4,
 'bindingdb': 29,
 'biogrid': 73,
 'chembl': 161,
 'dip': 0,
 'i2d-imex': 0,
 'innatedb': 13,
 'innatedb-imex': 0,
 'intact': 11,
 'interoporc': 0,
 'irefindex': 273,
 'matrixdb': 0,
 'mbinfo': 0,
 'mint': 34,
 'molcon': 0,
 'mpidb': 0,
 'reactome': 0,
 'reactome-fis': 134,
 'spike': 47,
 'string': 319,
 'topfind': 0,
 'uniprot': 0}
```

We see for instance that the **mint** database has 34 interactions. Coming back to the interactions returned by `s.query`, we find indeed 34 interactions between ZAP70 and another component:

```
>>> len(data)
34
```

Let us look at the first one:

```
>>> for x in data[0]: print(x)
uniprotkb:P15498
uniprotkb:P43403
-
-
uniprotkb:VAV1(gene name)|uniprotkb:VAV(gene name synonym)
uniprotkb:ZAP70(gene name)|uniprotkb:SRK(gene name synonym)|uniprotkb:70 kDa
zeta-associated protein(gene name synonym)|uniprotkb:Syk-related tyrosine
kinase(gene name synonym)
```

(continues on next page)

(continued from previous page)

```
psi-mi:"MI:0019"(coimmunoprecipitation)
-
pubmed:9151714
taxid:9606(Homo sapiens)
taxid:9606(Homo sapiens)
psi-mi:"MI:0914"(association)
psi-mi:"MI:0471"(mint)
mint:MINT-8035351
mint-score:0.28(free-text)|homomint-score:0.28(free-text)'intact-miscore:0.60']
```

The First two elements are the entries for specy A and B. The last element is the score. The 11th element is the type of interaction and so on.

What could be useful is to convert these elements into uniprot ID only. With mint database it is irrelevant for this particular entry but with other DBs or entries, it may be useful (e.g., biogrid).

BioServices provides such a function called `convert()`:

```
>>> data = s.query("biogrid", "ZAP70 AND species:9606")
>>> data2 = s.convert(data, "biogrid")
```

Warning: some databases may be offline. If so, try we another database. Type “s.activeDBs”.

convert method converts all entries from data into uniprot ID. If this is not possible, the entry is removed. The **query** and **convert** works on a single database but you we could query all or a subset of all databases using the `queryAll` and `convertAll` functions:

```
>>> data = s.queryAll("ZAP70 AND species:9606", databases=["mint", "biogrid"])
>>> data2 = s.convertAll(data)
```

However, extra cleaning is required to remove entries that are not relevant (no match to uniprot ID, redundant, not a protein, self interactions, ...). In order to ease this tast, the `psicquic.AppsPPI` class is very useful.

```
from bioservices import psicquic
s = psicquic.AppsPPI()
s.queryAll("ZAP70 AND species:9606", databases=["mint", "biogrid", "intact", "reactome-
fis"])
s.summary()
s.show_pie()
```

The `summary` function print a useful summary about the number of found interactions and overlap between databases:

```
>>> s.summary()
Found 8 interactions within intact database
Found 124 interactions within reactome-fis database
Found 19 interactions within mint database
Found 67 interactions within biogrid database
-----
Found 152 interactions in 1 common databases
Found 14 interactions in 2 common databases
Found 0 interactions in 3 common databases
Found 1 interactions in 4 common databases
```


This may be different depending on the available databases. Finally, you can obtain the relation that was found in the 4 databases:

```
>>> s.relevant_interactions[4]
[['LCK_HUMAN', 'ZAP70_HUMAN']]
```

What's next ?

There are lots of other services that could be useful. An example is the [Wikipathway](#) (see [Wikipathway](#)) to retrieve even more pathways that include the ZAP70 protein. Another example is the [BioMart](#) portal. You could use it to retrieve pathways from REACTOME (see [BioMart](#)). You can also retrieve target from ChEMBL given the uniprot ID (`get_target_by_uniprotId("P43403")`) and so on.

Manipulating compound identifiers

Application: retrieving information about a compound

This section uses BioServices to demonstrate the interest of combining several services together within a single framework using the Python language as a glue language

Retrieve a compound identifier from KEGG, ChEBI and ChEMBL

Let us look at a compound called **Geldanamycin** that inhibits Hsp90. Let us search for information about that compound in several databases and manipulate the different identifiers.

First, let us retrieve information on KEGG database:

```
>>> from bioservices import *
>>> k = KEGG(verbose=False)
```

KEGG compounds have links to other databases. It is not systematic but the ChEBI database is often referenced. So we will want to convert the KEGG identifier to a ChEBI identifier. Later, we can convert a ChEBI to a ChEMBL identifier using another Web Service such as UniChem.

We can get a mapping dictionary from the KEGG compound to ChEBI as follows:

```
>>> map_kegg_chebi = k.conv("chebi", "compound")
>>> len(map_kegg_chebi)
15845

>>> print(k.find("compound", "geldanamycin"))
cpd:C11222  Geldanamycin
cpd:C15823  Progeldanamycin
```

Let us look at the first one (KEGG id cpd:C11222). We can get lots of information from KEGG already by using:

```
>>> print(k.get("C11222"))
```

From which, there is a link to other databases in particular ChEBI (ChEBI:5292). We could use the mapping dictionary created above:

```
>>> map_kegg_chebi['cpd:C11222']
'chebi:5292'
```

Unfortunately, there is no mapping function from KEGG to ChEMBL in KEGG Web Service.

However, BioServices provides access to the *bioservices.unichem* service. This service provides a useful mapping function from kegg to chembl:

```
>>> uni = UniChem()
>>> mapping = uni.get_mapping("kegg_ligand", "chembl")
>>> mapping['C11222']
'CHEMBL278315'
```

For sanity check, let us see that the ChEBI is indeed 5292 as given within the KEGG database:

```
>>> uni = UniChem()
>>> mapping = uni.get_mapping("kegg_ligand", "chebi")
>>> mapping['C11222']
'5292'
```

(2) In order to convert KEGG gene names into uniprot gene name, we can also use the UniProt web service from BioServices as follows:

```
>>> from bioservices import *
>>> u = UniProt()
>>> u.mapping(fr='ID', to='KEGG_ID', query="ZAP70_HUMAN")
{'ZAP70_HUMAN': 'hsa:7535'}
```

You can get accession number or protein name identifier from the KEGG identifier as follows:

```
>>> u.mapping(fr='KEGG_ID', to='ID', query='hsa:7535')
{'hsa7535': 'ZAP70_HUMAN'}
>>> u.mapping(fr='KEGG_ID', to='ACC', query='hsa:7535')
{'hsa7535': 'P43403'}
```

Mapping identifiers

There are quite a few functions from different Web Services that can help to map identifiers from one database to the other. This tutorial presents some of them.

Convert from KEGG ID to ChEBI (compound)

```
>>> from bioservices import *
>>> k = KEGG(verbose=False)
>>> map_kegg_chebi = k.conv("chebi", "compound")
>>> map_kegg_chebi['cpd:C11222']
'chebi:5292'
```

you could also use *bioservices.unichem.UniChem* (see below).

Convert from KEGG ID to ChEMBL (compound)

```
>>> from bioservices import UniChem
>>> uni = UniChem()
>>> mapping = uni.get_mapping("kegg_ligand", "chembl")
>>> mapping["C11222"]
'CHEMBL278315'
```

convert from KEGG ID to UniProt and vice versa (gene)

In order to convert KEGG gene names into uniprot gene name, we can also use the UniProt web service from BioServices as follows:

```
>>> from bioservices import *
>>> u = UniProt()
>>> u.mapping(fr='ID', to='KEGG_ID', query="ZAP70_HUMAN")
['From:ID', 'To:KEGG_ID', 'ZAP70_HUMAN', 'hsa:7535']
```

You can get accession number or protein name identifier from the KEGG identifier as follows:

```
>>> u.mapping(fr='KEGG_ID', to='ID', query='hsa:7535')
'ZAP70_HUMAN'
>>> u.mapping(fr='KEGG_ID', to='ACC', query='hsa:7535')
'P43403'
```

One can also use the `bioservices.kegg.KEGG.conv()` method:

```
>>> k = KEGG()
>>> mapping_kegg_uniprot = k.conv("hsa", "uniprot")
```

BioMart service

BioMart provides a uniform interface to many services such as Cosmic, Ensembl and many more. In BioMart terminology a service is called a **mart**. As an example, we will consider the COSMIC interface provided by BioMart (see [COSMIC](#)). You can play with the interface itself to get an idea of what can be selected (e.g., datasets, filters, attributes). To help you, let us give a simple example that consists in converting the ensemble identifiers into entrez identifiers.

First you create an instance. There are lots of services behind the scene. The `ENSEMBL_MART_ENSEMBL` provides the conversion we are looking for.

```
from bioservices import BioMart
b = BioMart()
datasets = b.get_datasets("ENSEMBL_MART_ENSEMBL")
```

In datasets, there is a `hsapiens_gene_ensembl` database. Let us add it to the request that will be send:

```
b.add_dataset_to_xml(dataset)
```

We want to extract only the to following attributes:

```
b.add_attribute_to_xml("ensemble_gene_id")
b.add_attribute_to_xml("entrezgene_id")
```

If you are interested in a set of identifiers, provide it as a list (here below the queries:

```
queries = ["", ""]
b.add_filter_to_xml("ensemble_gene_id", queries)
```

and finally do the query itself:

```
xml = b.get_xml()
res = b.query(xml)
```

You can obtain the attributes and filters of a dataset as follows:

```
dataset = 'hsapiens_gene_ensembl'
attributes = b.attributes(dataset)
filters = b.filters(dataset)
```

Here is another example with cosmic.

Note: the cosmic mart was available at the time of 1.0 but not during release 1.4.1 . This is not a BioServices issue but the COSMIC mart being down. Hopefully, it will be available again soon. meanwhile this example should help you get a feeling of what can be done with a MART.

In **BioServices**, you can create a biomart request (which is a XML document) but first we need to figure out what are the datasets associated with the COSMIC mart. The tricky part is to know the names of the datasets/attributes/filters. BioServices provides a function that ease this task. First let create an instance of BioMart:

```
>>> from bioservices import *
>>> s = BioMart()
```

Then, let us use the `lookfor()` as follows:

```
>>> s.lookfor("cosmic")
Candidate:
  database: cosp
  MART name: CosmicMart
  displayName: COSMIC (SANGER UK)
  hosts: www.sanger.ac.uk
```

From the previous command, only one mart has been found. It is called CosmicMart, from which we can retrieve the datasets:

```
>>> s.datasets("CosmicMart")
['COSMIC67', 'COSMIC68', 'COSMIC66']
```

There are lots of entries in such datasets and we want to restrict our request using filters and attributes. Let us use the “COSMIC60” dataset. The following commands can help you in figuring out what are the valid names of attributes and filters to be used:

```
>>> s.attributes("COSMIC67")
>>> s.filters("COSMIC67")
```

They return list of dictionaries that provide the identifiers (keys of the dictionary) and information about the identifier (e.g. descriptive name).

For instance, if you want to add the gene name in the list of attributes, you will need to know its identifier. If you look at the dictionary you will find the “gene_name” key that contains:

```
>>> s.attributes("COSMIC67")["gene_name"]
['Gene Name',
 '',
 'naive_attributes',
 'html,txt,csv,tsv,xls',
 'COSMIC67__MART__MAIN',
 'gene_name']
```

So if you want to add the **Gene Name** attribute, you must use the **gene_name** identifier. Similarly for filters. In order to use a filter you must use the identifier as well as a value. Values are contained in the dictionary returned by filters(). For instance, the “Mutated Sample” filter given by the “samp_gene_mutated” identifier returns a list, which second element contains the list of valid values (here y or n character):

```
>>> s.filters("COSMIC67")
['Mutated Sample',
 '[y,n]',
 '',
 'naive_filters',
 'list',
 '=',
 'COSMIC67__MART__MAIN',
 'samp_gene_mutated']
```

So, there is a little bit of work for the user to figure out the identifiers of the attributes and filters. This could be a good exercise but let us give the list of relevant identifiers and there names that we want to use in this tutorial:

| category | name | identifier |
|-----------|-------------------|------------------------------|
| filter | Mutated Sample | samp_gene_mutated (y) |
| filter | Primary Site | site_primary (breast) |
| filter | Validation Status | validation_status (verified) |
| Attribute | Cosmic Sample ID | id_sample |
| Attribute | Sample Name | sample_name |
| Attribute | Sample Source | sample_source |
| Attribute | Tumour source | tumour_source |
| Attribute | Gene Name | gene_name |
| Attribute | Accession Number | accession_number |
| Attribute | Cosmi Mutation ID | id_mutation |
| Attribute | Gene ID | id_gene |

It is now time to create the XML request by adding attributes/filters and the dataset:

```
>>> # add the dataset
>>> s.add_dataset_to_xml("COSMIC67")

>>> # add the attributes
>>> s.add_attribute_to_xml("id_sample")
>>> s.add_attribute_to_xml("sample_name")
>>> s.add_attribute_to_xml("sample_source")
>>> s.add_attribute_to_xml("tumour_source")
>>> s.add_attribute_to_xml("gene_name")
```

(continues on next page)

(continued from previous page)

```
>>> s.add_attribute_to_xml("accession_number")
>>> s.add_attribute_to_xml("id_mutation")
>>> s.add_attribute_to_xml("id_gene")

>>> # add the filters
>>> s.add_filter_to_xml("samp_gene_mutated", "y")
>>> s.add_filter_to_xml("site_primary", "breast")
>>> s.add_filter_to_xml("validation_status", "verified")
```

You can create the XML request that will be send:

```
>>> xml = s.get_xml()
```

And finally send the request:

```
>>> res = s.query(xml)
```

GeneProf tutorial

GeneProf tutorial

New in version 1.2.0.

Section author: Thomas Cokelaer, Dec 2013

GeneProf is a web-based, graphical software suite that allows users to analyse data produced using high-throughput sequencing platforms (RNA-seq and ChIP-seq; “Next-Generation Sequencing” or NGS): Next-gen analysis for next-gen data

BioServices uses the GeneProf Web Services to enable programmatic access to the public data stored in GeneProf’s databases via Python.

Note: GeneProf services is quite versatile and contains many resources and examples. For any technical or scientific questions related to the service itself, please see [GeneProf About&Help](#).

Here below you will find a couple of examples related to GeneProf.

Histogram expression data

Reference

<https://www.geneprof.org/GeneProf/media/bpsm-2013/>

In the example below, we use geneprof to

1. search for Gene identifiers related to an organism (mouse) and keyword (nanog).
2. From the gene identifiers, retrieve the gene expression values for a given gene in all experiments
3. plot histogram of the log values found above.

Note: broken example on June 2017. Service should be fix soon. Issue reported to the author.

```
>>> from bioservices import GeneProf
>>> g = GeneProf(verbose=True)
>>> res = g.search_gene_ids("nanog", "mouse")
>>> print(res)
{10090: [29640, 14899]}
>>> expr1 = g.get_expression("mouse", 29640)['values']
>>> expr2 = g.get_expression("mouse", 14899)['values']

>>> import math
>>> values1 = [math.log(x["RPKM"]+1, 2.) for x in expr1]
>>> values2 = [math.log(x["RPKM"]+1, 2.) for x in expr2]

>>> from pylab import clf, subplot, hist
>>> clf()
>>> subplot(2,1,1)
>>> hist(values1)
>>> subplot(2,1,2)
>>> hist(values2)
```

Transcription factor network of stem cells

References

<https://www.geneprof.org/GeneProf/media/recomb-2013/>

Another example, here below consists in retrieving the binding targets of transcription factors (about 70) in mouse embryonic stem cells, and generate a SIF network that could be open and visualised in Cytoscape.

The example below can probably be simplified and make use of tools such as networkx to manipulate and visualise the final network. Please use with care:

```
>>> # first import and create a GenProf instance
>>> from bioservices import GeneProf
>>> g = GeneProf(verbose=False)
>>>

>>> # find all pubic experimental mouse samples in geneprof
>>> samples = g.get_list_experiment_samples("mouse")['samples']
>>> # look at entries that contains "Gene"
>>> graph = {}
>>> mapgene = {}
>>> for i, entry in enumerate(samples):
...     print("progress %s/%s" % (i+1, len(samples)))

...     # keep only entries that have cell type "embryonic stem cell" in the celltype
...     if "Gene" in entry.keys() and "Cell_Type" in entry.keys() and entry["Cell_Type"] == "embryonic stem cell":
...
...         # aliases
...         sampleId = entry['sample_id']
...         gene = entry["Gene"]
```

(continues on next page)

(continued from previous page)

```
...     # get gene id and save mapping in a dictionary to be used later
...     geneId = g.get_gene_id("mouse", "C_NAME", gene)['ids']
...     mapgene[geneId[0]] = gene

...     # get targets and print them
...     targets = g.get_targets_by_experiment_sample("mouse", sampleId)

...     # could be simplified inside the geneprof.py module
...     if 'targets' in targets.keys():
...         targets = targets['targets']

...     # print the results
...     for x in targets:
...         print gene, geneId[0], " ", x['feature_id']
...     graph[gene] = [x['feature_id'] for x in targets]

>>> # The graph saved in the graph variables is quite large. Let us simplified keeping
↳target that
>>> # are in the list of genes only
>>> simple_graph = {}
>>> for k, v in graph.iteritems():
...     simple_graph[k] = [mapgene[x] for x in v if x in mapgene.keys()]
>>> len(simple_graph.keys())
72
>>> sum([len(simple_graph[x]) for x in simple_graph.keys()])
2137
```

Finally, you can look at the graph with your favorite tool such as Cytoscape, Gephi.

Here below, I'm using a basic graph visualisation tool implemented in [CellNOpt](#), which is not dedicated for Network visualisation but contains a small interface to graphviz useful in this context (it has a python interface):

```
>>> from cno import CNOGraph
>>> c = CNOGraph()
>>> for k in simple_graph.keys():
...     for v in simple_graph[k]:
...         c.add_edge(k, v, link="+")
>>> c.centralty_degree()
>>> c.graph['graph'] = {"splines":"true", "size):(20,20),
... "dpi":200, "fixedsize":True}
>>> c.graph['node'] = {"width":.01, "height":.01,
... "size":0.01, "fontsize":8}
>>> c.plotdot(prog="fdp", node_attribute="degree")
```

geneprof_network.png

Integrating expression data in pathways

References

<https://www.geneprof.org/GeneProf/media/recomb-2013/>

This is another example from the reference above but based on tools available in bioservices so as to overlaid high-throughput gene expression onto pathways and models from KEGG database.

Fold changes in lymphoma vs. kidney on selected KEGG pathways

```
>>> from bioservices import KEGG, GeneProf, UniProt
>>> import StringIO
>>> import pandas
>>> g = GeneProf()
>>> k = KEGG()
>>> u = UniProt()

>>> # load ENCODE RNA-seq into a DataFrame for later
>>> data = g.get_data("11_683_28_1", "txt")
>>> rnaseq = pandas.read_csv(StringIO.StringIO(data), sep="\t")
>>> gene_names = rnaseq['Ensembl Gene ID']

>>> # get a pathway diagram for the KEGG path hsa05202 ("Transcriptional
>>> # misregulation in cancers")
>>> res = k.parse(k.get("hsa05202"))
>>> # extract KEGG identifiers corresponding to the genes found in the pathway
>>> keggids = ["hsa:"+x for x in res['GENE'].keys()]

>>> # we need to map the KEGG Ids to Ensembl Ids. We will use KEGG mapping and uniprot_
↳ mapping
>>> # for cases where the former does not have associated mapping.
>>> ensemblids = {}
>>> for id_ in keggids:
...     res = k.parse(k.get(id_))['DBLINKS']
...     if 'Ensembl' in res.keys():
...         print id_, res['Ensembl']
...         ensemblids[id_] = res['Ensembl']
...     else:
...         if "UniProt" in res.keys():
...             ids = res['UniProt'].split()[0]
...             m = u.mapping("ACC", "ENSEMBL_ID", query=ids)
...             if len(m): ensemblids[id_] = m[ids][0]
...         pass # no links to ensembl DB found

>>> # what are the KEGG id transformed into Ensembl Ids that are in the ENCODE data set ?
>>> found = [x for x in ensemblids.values() if x in [str(y) for y in gene_names]]
>>> indices = [i for i, x in enumerate(rnaseq['Ensembl Gene ID']) if x in found]
>>>

>>> # now, we can pick out the log2 fold change values for visualization:
>>> data = rnaseq.ix[indices][['Ensembl Gene ID', 'log2FC Lymphoma / EmbryonicKidney']]
>>> # and keep only those that have a negative or positive value
>>> mid = 1.5
>>> low = data[data['log2FC Lymphoma / EmbryonicKidney'] < -mid]
>>> geneid_low = list(low['Ensembl Gene ID'])
```

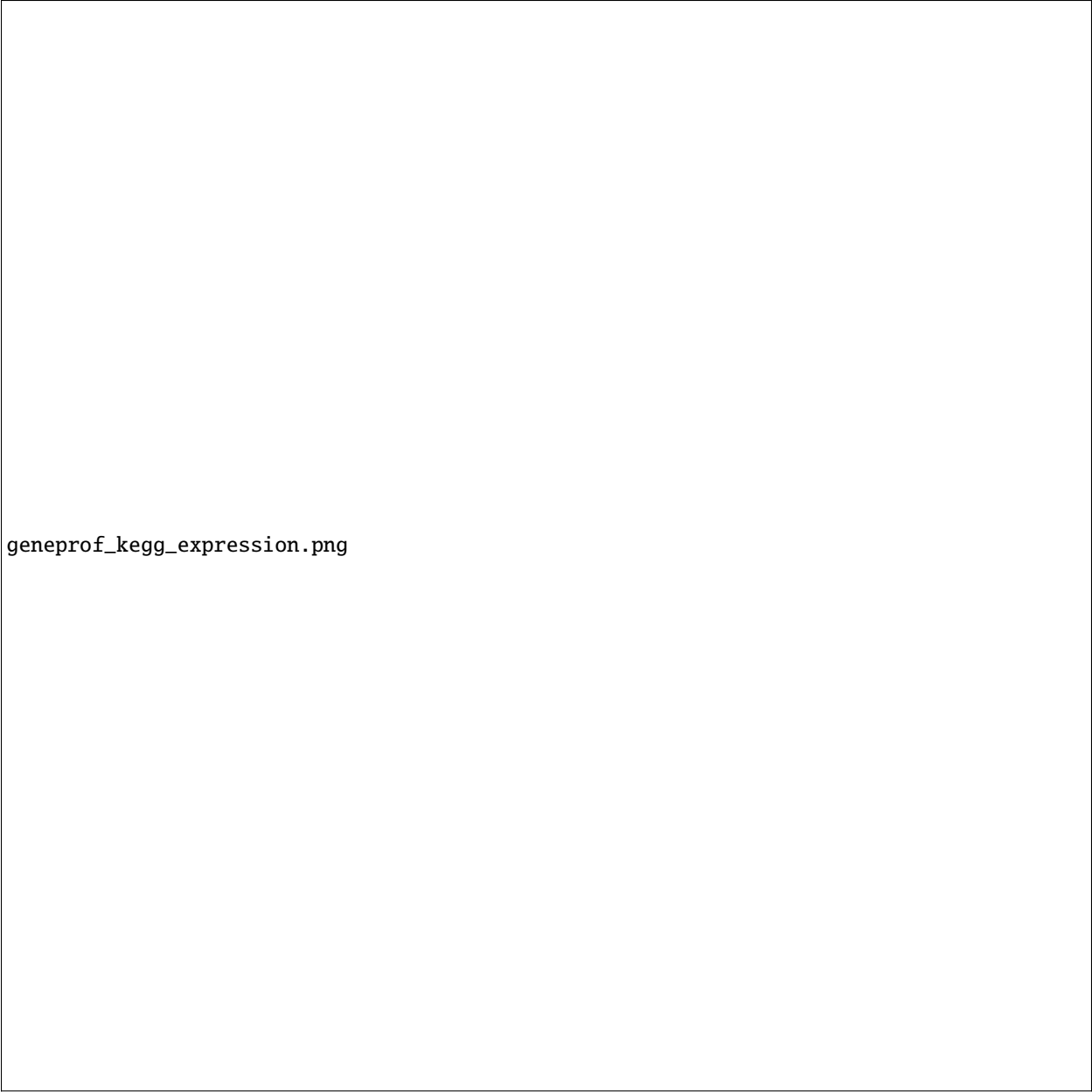
(continues on next page)

(continued from previous page)

```
>>> up = data[data['log2FC Lymphoma / EmbryonicKidney']>mid]
>>> geneid_up = list(up['Ensembl Gene ID'])
>>> mid = data[abs(data['log2FC Lymphoma / EmbryonicKidney'])<mid]
>>> geneid_mid = list(mid['Ensembl Gene ID'])

>>> # now that we have the genes (in ensembl format), we need the kegg id
>>> keggid_low = [this for this in keggids if ensemblids[this] in geneid_low]
>>> keggid_mid = [this for this in keggids if ensemblids[this] in geneid_mid]
>>> keggid_up = [this for this in keggids if ensemblids[this] in geneid_up]
>>> # it is now time to look at the expression on the diagram
>>> colors = {}
>>> for id_ in keggids: colors[id_[4:]] = "gray,"
>>> for id_ in keggid_low: colors[id_[4:]] = "blue,"
>>> for id_ in keggid_up: colors[id_[4:]] = "orange,"
>>> for id_ in keggid_mid: colors[id_[4:]] = "yellow,"
>>> k.show_pathway("hsa05202", dcolor="white", keggid=colors)
```

The last command will popup the KEGG diagram with the expression data on top of the diagram, as shown in the following picture:



geneprof_kegg_expression.png

6.1.3 Combining BioServices with external tools

Contents

- *Combining BioServices with external tools*
 - *PYMOL*
 - *BioPython*
 - *Galaxy*

This section shows how to use BioServices as an intermediate tool that fetch data to be used by third-party soft-

ware/application.

The external applications used in this section are not part of BioServices therefore we do not provide instructions for the installation. Reader should refer to the application web site instead (URLs are provided here below). However, we indicate the way we installed them.

PYMOL

URL

<http://www.pymol.org/>

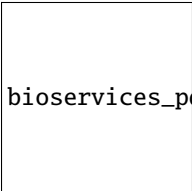
This example below uses the external software called PyMOL. We have installed it without trouble by downloading the source file from their website. Then, we typed those commands in a shell:

```
bunzip pymol-v1.6alpha1.tar.bz2
tar xvf pymol-v1.6alpha1.tar
cd pymol
python setup.py install
```

You may need to install library if requested. Tested under Fedora 15.

The following code uses BioServices to get the PDB Identifier of a protein called ZAP70. To do so, we use *bioservices.uniprot.UniProt* to get its accession number (P43403) and its PDB identifier. Then, we use *bioservices.pdb.PDB* to get the 3D structure in PDB format.

The script above uses PyMOL in a script manner to save the 3D graphical representation of the protein (here below) but you could also use PyMOL in an interactive mode.



bioservices_pdb.png

BioPython

URL

<http://biopython.org/DIST/docs/tutorial/Tutorial.html#chapter:Bio.AlignIO>

BioPython provides many tools for IO, algorithms and access to Web services. BioServices provides access to many web services. This example shows how (i) to use BioServices to retrieve FASTA files and (ii) BioPython to play with the sequences.

Note: We assume you have installed BioPython (pip install biopython)

First, let us retrieve two FASTA sequences and save them in 2 files:

```
from bioservices import UniProt
u = UniProt()
akt1 = u.retrieve("P31749", "fasta")
akt2 = u.retrieve("P31751", "fasta")
```

(continues on next page)

(continued from previous page)

```
fh = open("akt1.fasta", "w")
fh.write(akt1)
fh.close()

fh = open("akt2.fasta", "w")
fh.write(akt2)
fh.close()
```

Now, on the BioPython side, we read the 2 sequences and introspect them:

```
>>> from Bio import AlignIO
>>> record1 = SeqIO.read("akt1.fasta", "fasta")
>>> record2 = SeqIO.read("akt2.fasta", "fasta")
>>> record1 += "-" # this is to have 2 sequences on same length as requested by the
↳ following function

>>> alignment = AlignIO.MultipleSeqAlignment([])
>>> alignment.append(record1)
>>> alignment.append(record2)

>>> for record in alignment:
>>>     print(description)
sp|P31749|AKT1_HUMAN RAC-alpha serine/threonine-protein kinase OS=Homo sapiens GN=AKT1
↳ PE=1 SV=2
sp|P31751|AKT2_HUMAN RAC-beta serine/threonine-protein kinase OS=Homo sapiens GN=AKT2
↳ PE=1 SV=2
```

You are ready to play with BioPython multiple alignment tools. Please consult BioPython documentation for more examples.

Galaxy

URL

<http://wiki.galaxyproject.org/FrontPage>

Date

Aug 2013

Galaxy is an open, web-based platform for accessible, reproducible, and transparent computational biomedical research. It provides workflows and plugins to many web resources.

This tutorial shows how to link bioservices and galaxy. Our tutorial will provide a plugin to Galaxy so that a user can retrieve a FASTA file via BioServices and the wrapping of UniProt Web Services.

We assume that you installed Galaxy on your system via the source code:

```
hg clone https://bitbucket.org/galaxy/galaxy-dist/
cd galaxy-dist
hg update stable
```

The tree directory should therefore contains a directory called **tools/** and in the main directory, an XML file called **conf_tools.py**

We will first create a plugin for bioservices. This is done by adding a directory called bioservices in ./tools:

```
mkdir tools/bioservices
```

In this directory, we will create two files called **uniprot.py** that will contain the actual code that calls bioservices and a second XML file that will allow us to design the plugin layout.

Let us start with the plugin. It is very simple since only the UniProt Entry is required. The output will simply be the FASTA file that would have been fetched.

The XML file is:

```
<tool id="bioservices_uniprot" name="Get FASTA" version="1.1.0">
  <description>from UniProt via Bioservices</description>
  <requirements>
    <requirement type="package">bioservices</requirement>
  </requirements>
  <command interpreter="python">uniprot.py $uniprot_id $output</command>
  <inputs>
    <param name="uniprot_id" type="text" label="UniProt ID" size="40" help="Provide a
    ↪ valid UniProt Entry (e.g. P43403)" />
  </inputs>
  <outputs>
    <data format="fasta" name="output" />
  </outputs>
  <help>
Fetch a FASTA file using UniProt via BioServices. Simply provide a valid Uniprot Entry.
    ↪ (e.g., P43403)
  </help>
</tool>
```

The python code will take as an input the UniProt ID and create a file that contains the FASTA data:

```
import sys

def __main__():
    ids = sys.argv[1]
    filename = sys.argv[2]
    # TODO: check the validity and format ?
    try:
        from bioservices import UniProt
        u = UniProt(verbose=False)
        u.debugLevel = "ERROR"
    except ImportError:
        print("Could not import bioservices ? Check that it is installed. Try 'pip
        ↪ install bioservices'")

    try:
        fasta = u.searchUniProtId(ids, "fasta")
    except:
        print("An error occurred while fetching the FASTA file from uniprot")

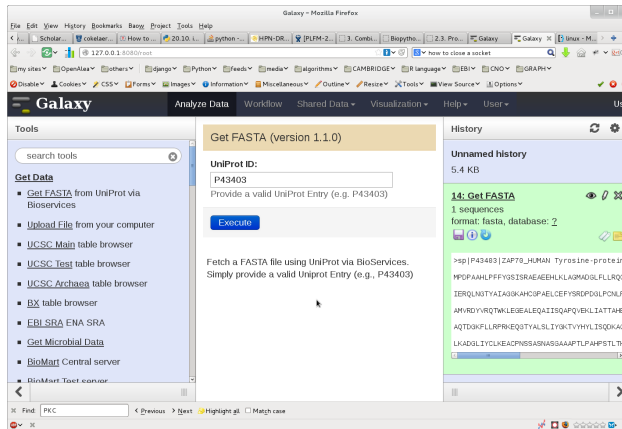
    try:
        fh = open(filename, "w")
        fh.write(fasta)
```

finally, you need to make Galaxy aware of this new plugin. this is done in the file called conf_tool.xml. Add bioservices

plugin. The beginning of the file should look like:

```
<?xml version="1.0"?>
<toolbox>
  <section name="Get Data" id="gettext">
    <tool file="bioservices/uniprot.xml"/>
    <tool file="data_source/upload.xml"/>
  ...
```

Once done, start your galaxy server. The following image shows the outcome: in the left hand side, you can select the bioservices plugin. Then, in the center, you can enter a uniprot entry. Press the execute button and the new file should appear in the right hand side. From there you can use Galaxy other tools to analyse the file.



This example shows that it is possible to link Galaxy and BioServices to access various Web Services that are available through Bioservices.

PyMOL

URL

<http://www.pymol.org/>


This example below uses the external software called PyMOL. We have installed it without trouble by downloading the source file from their website. Then, we typed those commands in a shell:

```
bunzip pymol-v1.6alpha1.tar.bz2
tar xvf pymol-v1.6alpha1.tar
cd pymol
python setup.py install
```

You may need to install library if requested. Tested under Fedora 15.

The following code uses BioServices to get the PDB Identifier of a protein called ZAP70. To do so, we use *bioservices.uniprot.UniProt* to get its accession number (P43403) and its PDB identifier. Then, we use *bioservices.pdb.PDB* to get the 3D structure in PDB format.

The script above uses PyMOL in a script manner to save the 3D graphical representation of the protein (here below) but you could also use PyMOL in an interactive mode.

bioservices_pdb.png

BioPython

URL

<http://biopython.org/DIST/docs/tutorial/Tutorial.html#chapter:Bio.AlignIO>

BioPython provides many tools for IO, algorithms and access to Web services. BioServices provides access to many web services. This example shows how (i) to use BioServices to retrieve FASTA files and (ii) BioPython to play with the sequences.

Note: We assume you have installed BioPython (pip install biopython)

First, let us retrieve two FASTA sequences and save them in 2 files:

```
from bioservices import UniProt
u = UniProt()
akt1 = u.retrieve("P31749", "fasta")
akt2 = u.retrieve("P31751", "fasta")

fh = open("akt1.fasta", "w")
fh.write(akt1)
fh.close()

fh = open("akt2.fasta", "w")
fh.write(akt2)
fh.close()
```

Now, on the BioPython side, we read the 2 sequences and introspect them:

```
>>> from Bio import AlignIO
>>> record1 = SeqIO.read("akt1.fasta", "fasta")
>>> record2 = SeqIO.read("akt2.fasta", "fasta")
>>> record1 += "-" # this is to have 2 sequences on same length as requested by the
↳ following function

>>> alignment = AlignIO.MultipleSeqAlignment([])
>>> alignment.append(record1)
>>> alignment.append(record2)

>>> for record in alignment:
>>>     print(description)
sp|P31749|AKT1_HUMAN RAC-alpha serine/threonine-protein kinase OS=Homo sapiens GN=AKT1
↳ PE=1 SV=2
sp|P31751|AKT2_HUMAN RAC-beta serine/threonine-protein kinase OS=Homo sapiens GN=AKT2
↳ PE=1 SV=2
```

You are ready to play with BioPython multiple alignment tools. Please consult BioPython documentation for more

examples.

Galaxy

URL

<http://wiki.galaxyproject.org/FrontPage>

Date

Aug 2013

Galaxy is an open, web-based platform for accessible, reproducible, and transparent computational biomedical research. It provides workflows and plugins to many web resources.

This tutorial shows how to link bioservices and galaxy. Our tutorial will provide a plugin to Galaxy so that a user can retrieve a FASTA file via BioServices and the wrapping of UniProt Web Services.

We assume that you installed Galaxy on your system via the source code:

```
hg clone https://bitbucket.org/galaxy/galaxy-dist/
cd galaxy-dist
hg update stable
```

The tree directory should therefore contains a directory called **tools/** and in the main directory, an XML file called **conf_tools.py**

We will first create a plugin for bioservices. This is done by adding a directory called bioservices in ./tools:

```
mkdir tools/bioservices
```

In this directory, we will create two files called **uniprot.py** that will contain the actual code that calls bioservices and a second XML file that will allows us to design the plugin layout.

Let us start with the plugin. It is very simple since only the UniProt Entry is required. The output will simply be the FASTA file that would have been fetched.

The XML file is:

```
<tool id="bioservices_uniprot" name="Get FASTA" version="1.1.0">
  <description>from UniProt via Bioservices</description>
  <requirements>
    <requirement type="package">bioservices</requirement>
  </requirements>
  <command interpreter="python">uniprot.py $uniprot_id $output</command>
  <inputs>
    <param name="uniprot_id" type="text" label="UniProt ID" size="40" help="Provide a
    ↪ valid UniProt Entry (e.g. P43403) "/>
  </inputs>
  <outputs>
    <data format="fasta" name="output" />
  </outputs>
  <help>
Fetch a FASTA file using UniProt via BioServices. Simply provide a valid Uniprot Entry.
    ↪ (e.g., P43403)
  </help>
</tool>
```

The python code will take as an input the UniProt ID and create a file that contains the FASTA data:

```
import sys

def __main__():
    ids = sys.argv[1]
    filename = sys.argv[2]
    # TODO: check the validity and format ?
    try:
        from bioservices import UniProt
        u = UniProt(verbose=False)
        u.debugLevel = "ERROR"
    except ImportError:
        print("Could not import bioservices ? Check that it is installed. Try 'pip_
        ↪install bioservices'")

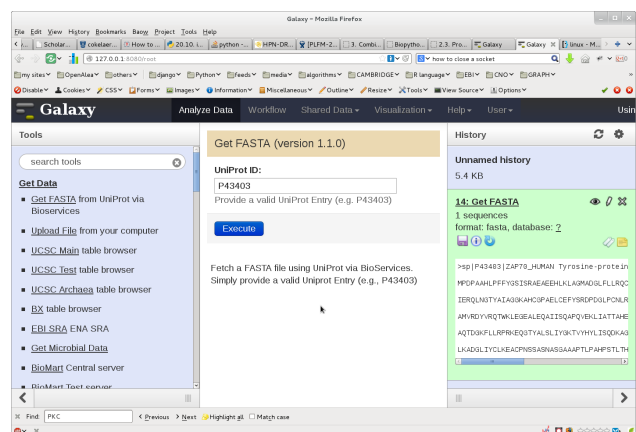
    try:
        fasta = u.searchUniProtId(ids, "fasta")
    except:
        print("An error occured while fetching the FASTA file from uniprot")

    try:
        fh = open(filename, "w")
        fh.write(fasta)
```

finally, you need to make Galaxy aware of this new plugin. this is done in the file called conf_tool.xml. Add bioservices plugin. The beginning of the file should look like:

```
<?xml version="1.0"?>
<toolbox>
  <section name="Get Data" id="gettext">
    <tool file="bioservices/uniprot.xml"/>
    <tool file="data_source/upload.xml"/>
  ...
```

Once done, start your galaxy server. The following image shows the outcome: in the left hand side, you can select the bioservices plugin. Then, in the center, you can enter a uniprot entry. Press the execute button and the new file should appear in the right hand side. From there you can use Galaxy other tools to analyse the file.



This example shows that it is possible to link Galaxy and BioServices to access to various Web Services that are available through Bioservices.

6.1.4 Developer Guide

Naming convention

To add a web services in BioServices, decide on a name for the python module. By convention we have the module name in lower case. Internally, class uses standard Python convention (Upper case for first letter).

The module name (e.g. uniprot) should be use to name the module (uniprot.py).

It will also be used to add a test or the continuous integration

Creating a service class (REST case)

You can test directly a SOAP/WSDL or REST service in a few lines. For instance, to access to the biomart REST service, type:

```
>>> s = REST("BioMart" , "http://www.biomart.org/biomart/martservice")
```

The first parameter is compulsory but can be any word. You can retrieve the base URL by typing:

```
>>> s.url
'http://www.biomart.org/biomart/martservice'
```

and then send a request to retrieve registry information for instance (see www.biomart.org/martservice.html for valid request:

```
>>> s.http_get("?type=registry")
<bioservices.xmltools.easyXML at 0x3b7a4d0>
```

The request method available from RESTService class concatenates the url and the parameter provided so it request the “<http://www.biomart.org/biomart/martservice>” URL.

As a developer, you should ease the life of the user by wrapping up the previous commands. An example of a BioMart class with a unique method dedicated to the registry would look like:

```
>>> class BioMart(REST):
...     def __init__(self):
...         url = "http://www.biomart.org/biomart/martservice"
...         super(BioMart, self).__init__("BioMart", url=url)
...     def registry(self):
...         ret = self.request("?type=registry")
...         return ret
```

and you would use it as follows:

```
>>> s = BioMart()
>>> s.registry()
<bioservices.xmltools.easyXML at 0x3b7a4d0>
```

Creating a service class (WSDL case)

If a web service interface is not provided within bioservices, you can still easily access its functionalities. As an example, let us look at the [Ontology Lookup service](#), which provides a WSDL service. In order to easily access this service, use the WSDLService class as follows:

```
>>> from bioservices import WSDLService
>>> ols = WSDLService("OLS", "http://www.ebi.ac.uk/ontology-lookup/OntologyQuery.wsdl")
```

You can now see which methods are available:

```
>>> ols.wsdl_methods
```

and call one (getVersion) using the bioservices.services.WSDLService.serv():

```
>>> ols.serv.getVersion()
```

You can then look at something more complex and extract relevant information:

```
>>> [x.value for x in ols.serv.getOntologyNames()[0]]
```

Of course, you can add new methods to ease the access to any functionalities:

```
>>> ols.getOnlogyNames() # returns the values
```

Similarly to the previous case using REST, you can wrap this example into a proper class.

Others

When wrapper a WSDL services, it may be difficult to know what parameters to provide if the API doc is not clear. This can be known as follows using the suds factory. In this previous examples, we could use:

```
>>> ols.suds.factory.resolver.find('getTermById')
<Element:0xa848b50 name="getTermById" />
```

For eutils, this was more difficult:

```
m1 = list(e.suds.wsdl.services[0].ports[0].methods.values())[2]
m1.soap.input.body.parts[0]
the service is in m1.soap.input.body.parts[0] check for the element in the
root attribute
```

suds and client auth

<http://stackoverflow.com/questions/6277027/suds-over-https-with-cert>

How to include tests ?

We use pytest. There are many web services included in BioServices. Consequently there are many tests. It is common to have failed tests on Travis and the continuous integration.

Some tests are known to be long or failing from time to time (e.g. service is down).

When a test is known to fail sometimes, we can add this decorator:

```
@pytest.mark.flaky(max_runs=3, min_passes=1)
```

On travis we allows 8 failures.

For long tests, we allows 60s at most. You can mark a tests if you knw it will fail on travis (e.g. too long):

```
pytest.mark.xfail
```

Finally, we skip some tests for some conditions:

```
skiptravis = pytest.mark.skipif( "TRAVIS_PYTHON_VERSION" in os.environ,
    reason="On travis")
@skiptravis
def test():
    ...
```

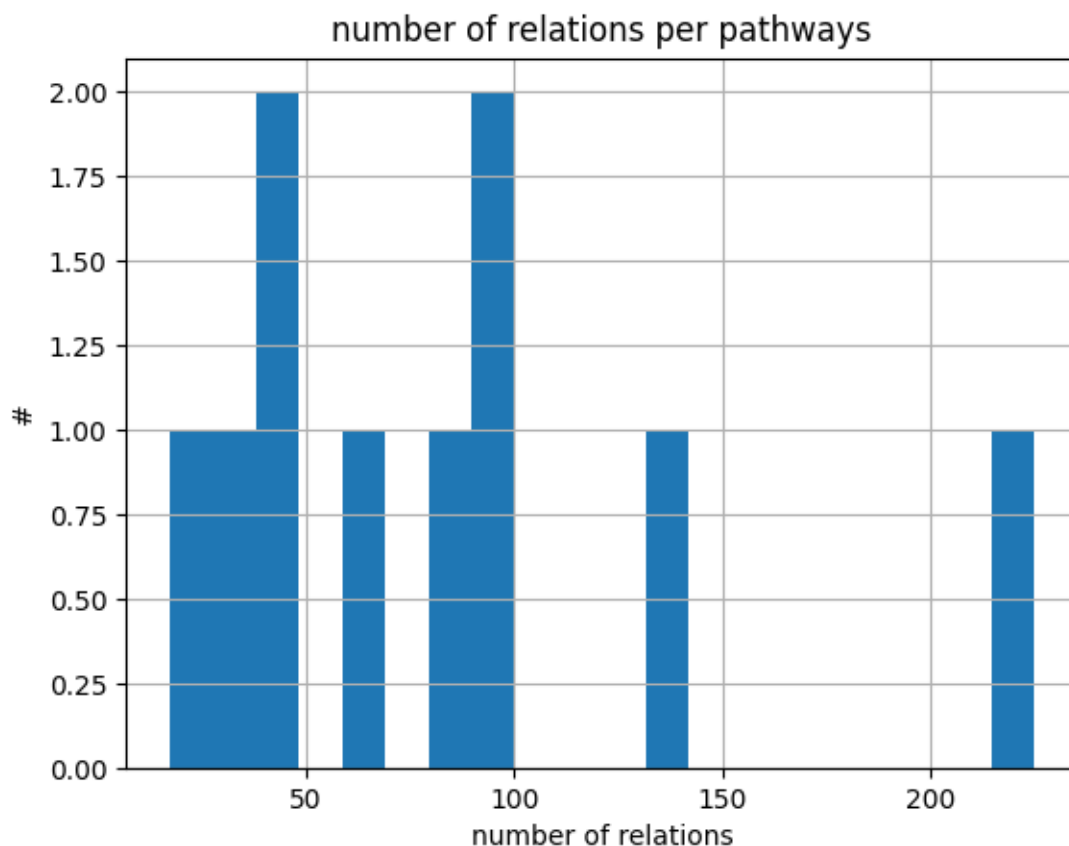
Continuous integration

1. add a test in ./test/webservices/test_**yourmodule**.py
2. add a continous integration file named after **yourmodule**.yaml. See example in .github/workflows/template.txt and replace **__name__** by your module name

6.1.5 Gallery

KEGG module example

Histogram of KEGG pathways relations



Out:

```

Creating directory /home/docs/.cache/bioservices
Welcome to Bioservices
=====
It looks like you do not have a configuration file.
We are creating one with default values in /home/docs/.config/bioservices/bioservices.
↪cfg .
Done
/home/docs/checkouts/readthedocs.org/user_builds/bioservices/envs/latest/lib/python3.7/
↪site-packages/bs4/builder/__init__.py:546: XMLParsedAsHTMLWarning: It looks like you
↪'re parsing an XML document using an HTML parser. If this really is an HTML document,
↪(maybe it's XHTML?), you can ignore or filter this warning. If it's XML, you should
↪know that using an XML parser will be more reliable. To parse this document as XML,
↪make sure you have the lxml package installed, and pass the keyword argument `features=
↪"xml"` into the BeautifulSoup constructor.
XMLParsedAsHTMLWarning.MESSAGE, XMLParsedAsHTMLWarning

```

```
from pylab import *
```

(continues on next page)

(continued from previous page)

```

# extract all relations from all pathways
from bioservices.kegg import KEGG
s = KEGG()
s.organism = "hsa"

# retrieve more than 260 pathways so it takes time
max_pathways = 10
results = [s.parse_kgml_pathway(x) for x in s.pathwayIds[0:max_pathways]]
relations = [x['relations'] for x in results]

# plot
hist([len(this) for this in relations], 20)
xlabel('number of relations')
ylabel('#')
title("number of relations per pathways")
grid(True)

```

Total running time of the script: (0 minutes 11.857 seconds)

6.1.6 NoteBooks

ipython notebook can be downloaded from this section. You can either see the results via nbviewer, or download and run them yourself. To do so, download the notebook and open a shell where is saved the file. Type:

```
ipython notebook
```

You should see the notebook name. Click on it and you are ready to try. Cels can be executed by typing CTRL+enter

UniProt

Here is a ipython notebook dedicated to UniProt, which can be downloaded [notebooks/UniProt.ipynb](#) or view its results on [uniprot nbviewer](#)

BioModels

Here is a ipython notebook dedicated to BioModels, which can be downloaded [notebooks/BioModels.ipynb](#) or view its results on [biomodels nbviewer](#)

ChEMBL

Here is a ipython notebook dedicated to ChEMBL, which can be downloaded [notebooks/ChEMBL.ipynb](#) or view its results on [chembl nbviewer](#)

Entrez/Eutils

Here is a ipython notebook dedicated to EUtils, which can be downloaded `notebooks/Entrez_EUtils.ipynb` or view its results on [eutil nbviewer](#)

KEGG

Here is a ipython notebook dedicated to KEGG, which can be downloaded `notebooks/KEGG.ipynb` or view its results on [kegg nbviewer](#)

MUSCLE

Here is a ipython notebook dedicated to MUSCLE, which can be downloaded `notebooks/MUSCLE.ipynb` or view its results on [muscle nbviewer](#)

NCBIBlast

Here is a ipython notebook dedicated to NCBIblast, which can be downloaded `notebooks/NCBIBlast.ipynb` or view its results on [ncbiblast nbviewer](#)

WikiPathway

Here is a ipython notebook dedicated to WikiPathway, which can be downloaded `notebooks/WikiPathway.ipynb` or view its results on [wiki pathway nbviewer](#)

Gene Mapping

Here is a ipython notebook dedicated to Gene Mapping, which can be downloaded `notebooks/Gene_Mapping.ipynb` or view its results on [gene mapping nbviewer](#)

BioMart

Here is a ipython notebook dedicated to BioMart, which can be downloaded `notebooks/BioMart` or view its results on [chembl biomiart](#)

Ensembl

Here is a ipython notebook dedicated to Ensembl, which can be downloaded `notebooks/Ensembl.ipynb` or view its results on [ensembl nbviewer](#)

Contents

- *Utilities*
 - *Service module (REST or WSDL)*
 - *xmltools module*

- *Services*
 - *ArrayExpress*
 - *Biocontainers*
 - *BiGG*
 - *BioDBnet*
 - *BioGrid*
 - *BioMart*
 - *BioModels*
 - *ChEBI*
 - *ChEMBL*
 - *COG*
 - *ENA*
 - *EUtils*
 - *GeneProf*
 - *QuickGO*
 - *Kegg*
 - * *Some terminology*
 - * *KEGG Databases Names and Abbreviations*
 - * *Database Entries*
 - *HGNC*
 - *Intact (complex)*
 - *MUSCLE*
 - *MyGeneInfo*
 - *NCBIblast*
 - *OmniPath Commons*
 - *Panther*
 - *Pathway Commons*
 - *PDB/PDBe modules*
 - *PRIDE module*
 - *PSICQUIC*
 - * *About queries*
 - * *About the MITAB output*
 - *Rhea*
 - *Reactome*
 - *Readseq*

- *UniChem*
- *UniProt*
- *DBFetch*
- *Wikipathway*
- *Applications and extra tools*
 - *Peptides*
 - *FASTA*

6.1.7 Utilities

Service module (REST or WSDL)

Modules with common tools to access web resources

exception `BioServicesError`(*value*)

class `REST`(*name*, *url=None*, *verbose=True*, *cache=False*, *requests_per_sec=3*, *proxies=[]*, *cert=None*, *url_defined_later=False*)

The ideas (sync/async) and code using requests were inspired from the chembl python wrapper but significantly changed.

Get one value:

```
>>> from bioservices import REST
>>> s = REST("test", "https://www.ebi.ac.uk/chemblws")
>>> res = s.get_one("targets/CHEMBL2476.json", "json")
>>> res['organism']
u'Homo sapiens'
```

The caching has two major interests. First one is that it speed up requests if you repeat requests.

```
>>> s = REST("test", "https://www.ebi.ac.uk/chemblws")
>>> s.CACHING = True
>>> # requests will be stored in a local sqlite database
>>> s.get_one("targets/CHEMBL2476")
>>> # Disconnect your wiki and any network connections.
>>> # Without caching you cannot fetch any requests but with
>>> # the CACHING on, you can retrieve previous requests:
>>> s.get_one("targets/CHEMBL2476")
```

Advantages of requests over urllib

requests length is not limited to 2000 characters <http://www.g-loaded.eu/2008/10/24/maximum-url-length/>

There is no need for authentication if the web services available in bioservices except for a few exception. In such case, the username and password are to be provided with the method call. However, in the future if a services requires authentication, one can set the attribute `authentication` to a tuple:

```
s = REST()
s.authentication = ('user', 'pass')
```

Note about headers and content type. The Accept header is used by HTTP clients to tell the server what content types they will accept. The server will then send back a response, which will include a Content-Type header telling the client what the content type of the returned content actually is. When using the `get__headers()`, you can see the User-Agent, the Accept and Content-Type keys. So, here the HTTP requests also contain Content-Type headers. In POST or PUT requests the client is actually sending data to the server as part of the request, and the Content-Type header tells the server what the data actually is. For a POST request resulting from an HTML form submission, the Content-Type of the request should be one of the standard form content types: application/x-www-form-urlencoded (default, older, simpler) or multipart/form-data (newer, adds support for file uploads)

Constructor

Parameters

- **name** (*str*) – a name for this service
- **url** (*str*) – its URL
- **verbose** (*bool*) – prints informative messages if True (default is True)
- **requests_per_sec** – maximum number of requests per seconds are restricted to 3. You can change that value. If you reach the limit, an error is raised. The reason for this limitation is that some services (e.g., NCBI) may black list you IP. If you need or can do more (e.g., ChEMBL does not seem to have restrictions), change the value. You can also have several instance but again, if you send too many requests at the same, your future requests may be restricted. Currently implemented for REST only

All instances have an attribute called `logging` that is an instance of the `logging` module. It can be used to print information, warning, error messages:

```
self.logging.info("informative message")
self.logging.warning("warning message")
self.logging.error("error message")
```

The attribute `debugLevel` can be used to set the behaviour of the logging messages. If the argument `verbose` is True, the `debugLevel` is set to INFO. If `verbose` is False, the `debugLevel` is set to WARNING. However, you can use the `debugLevel` attribute to change it to one of DEBUG, INFO, WARNING, ERROR, CRITICAL. `debugLevel=WARNING` means that only WARNING, ERROR and CRITICAL messages are shown.

property TIMEOUT

clear_cache()

```
content_types = {'bed': 'text/x-bed', 'default':
'application/x-www-form-urlencoded', 'fasta': 'text/x-fasta', 'gff3':
'text/x-gff3', 'gif': 'image/gif', 'jpeg': 'image/jpeg', 'jpg': 'image/jpeg',
'json': 'application/json', 'jsonp': 'text/javascript', 'nh': 'text/x-nh',
'phylip': 'text/x-phyloxml+xml', 'phyloxml': 'text/x-phyloxml+xml', 'png':
'image/png', 'seqxml': 'text/x-seqxml+xml', 'svg': 'image/svg', 'svg+xml':
'image/svg+xml', 'text': 'text/plain', 'txt': 'text/plain', 'xml':
'application/xml', 'yaml': 'text/x-yaml'}
```

debug_message()

delete_cache()

delete_one(*query*, *frmt*='json', ***kargs*)

getUserAgent()

get_async(*keys*, *frmt*='json', *params*={}, ***kargs*)

get_headers(*content*='default')

Parameters

content (*str*) – set to default that is application/x-www-form-urlencoded so that it has the same behaviour as urllib2 (Sept 2014)

get_one(*query*=None, *frmt*='json', *params*={}, ***kargs*)

if *query* starts with <http://> do not use self.url

get_sync(*keys*, *frmt*='json', ***kargs*)

http_delete(*query*, *params*=None, *frmt*='xml', *headers*=None, ***kargs*)

http_get(*query*, *frmt*='json', *params*={}, ***kargs*)

- *query* is the suffix that will be appended to the main url attribute.
- *query* is either a string or a list of strings.
- if list is larger than ASYNC_THRESHOLD, use asynchronous call.

http_post(*query*, *params*=None, *data*=None, *frmt*='xml', *headers*=None, *files*=None, *content*=None, ***kargs*)

post_one(*query*=None, *frmt*='json', ***kargs*)

property session

class Service(*name*, *url*=None, *verbose*=True, *requests_per_sec*=10, *url_defined_later*=False)

Base class for WSDL and REST classes

See also:

[REST](#), [WSDLService](#)

Constructor

Parameters

- **name** (*str*) – a name for this service
- **url** (*str*) – its URL
- **verbose** (*bool*) – prints informative messages if True (default is True)
- **requests_per_sec** – maximum number of requests per seconds are restricted to 3. You can change that value. If you reach the limit, an error is raise. The reason for this limitation is that some services (e.g., NCBI) may black list you IP. If you need or can do more (e.g., ChEMBL does not seem to have restrictions), change the value. You can also have several instance but again, if you send too many requests at the same, your future requests may be retracted. Currently implemented for REST only

All instances have an attribute called **logging** that is an instance of the [logging](#) module. It can be used to print information, warning, error messages:

```
self.logging.info("informative message")
self.logging.warning("warning message")
self.logging.error("error message")
```

The attribute `debugLevel` can be used to set the behaviour of the logging messages. If the argument `verbose` is `True`, the `debugLevel` is set to `INFO`. If `verbose` is `False`, the `debugLevel` is set to `WARNING`. However, you can use the `debugLevel` attribute to change it to one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`. `debugLevel=WARNING` means that only `WARNING`, `ERROR` and `CRITICAL` messages are shown.

property **CACHING**

easyXML(*res*)

Use this method to convert a XML document into an *easyXML* object

The *easyXML* object provides utilities to ease access to the XML tag/attributes.

Here is a simple example starting from the following XML

```
>>> from bioservices import *
>>> doc = "<xml> <id>1</id> <id>2</id> </xml>"
>>> s = Service("name")
>>> res = s.easyXML(doc)
>>> res.findAll("id")
[<id>1</id>, <id>2</id>]
```

property **easyXMLConversion**

If `True`, xml output from a request are converted to *easyXML* object (Default behaviour).

on_web(*url*)

Open a URL into a browser

pubmed(*Id*)

Open a pubmed Id into a browser tab

Parameters

Id – a valid pubmed Id in string or integer format.

The URL is a concatenation of the pubmed URL <http://www.ncbi.nlm.nih.gov/pubmed/> and the provided Id.

```
response_codes = {200: 'OK', 201: 'Created', 400: 'Bad Request. There is a
problem with your input', 404: 'Not found. The resource you requests does not
exist', 405: 'Method not allowed', 406: 'Not Acceptable. Usually headers issue',
410: 'Gone. The resource you requested was removed.', 415: 'Unsupported Media
Type', 500: 'Internal server error. Most likely a temporary problem', 503:
'Service not available. The server is being updated, try again later'}
```

some useful response codes

save_str_to_image(*data*, *filename*)

Save string object into a file converting into binary

property **url**

URL of this service

class `WSDLService(name, url, verbose=True, cache=False)`

Class dedicated to the web services based on WSDL/SOAP protocol.

See also:

`RESTService`, [Service](#)

Constructor

Parameters

- **name** (*str*) – a name e.g. Kegg, Reactome, ...
- **url** (*str*) – the URL of the WSDL service
- **verbose** (*bool*) – prints informative messages

The `serv` give access to all WSDL functionalities of the service.

The `methods` is an alias to `self.serv.methods` and returns the list of functionalities.

property `TIMEOUT`

wsdl_create_factory(*name, **kargs*)

property `wsdl_methods`

returns methods available in the WSDL service

wsdl_methods_info()

xmltools module

This module includes common tools to manipulate XML files

class `easyXML(data, encoding='utf-8')`

class to ease the introspection of XML documents.

This class uses the standard `xml` module as well as the package `BeautifulSoup` to help introspecting the XML documents.

```
>>> from bioservices import *
>>> n = ncbiblast.NCBIBlast()
>>> res = n.getParameters() # res is an instance of easyXML
>>> # You can retrieve XML from this instance of easyXML and print the content
>>> # in a more human-readable way.
>>> res.soup.findAll('id') # a BeautifulSoup instance is available
>>> res.root # and the root using xml.etree.ElementTree
```

There is a `getitem` so you can type:

```
res['id']
```

which is equivalent to:

```
res.soup.findAll('id')
```

There is also aliases `findAll` and `prettify`.

Constructor

Parameters

- **data** – an XML document format
- **fixing_unicode** – use only with HGNC service to fix issue with the XML returned by that particular service. No need to use otherwise. See [HGNC](#) documentation for details.
- **encoding** – default is utf-8 used. Used to fix the HGNC XML only.

The data parameter must be a string containing the XML document. If you have an URL instead, use [readXML](#)

getchildren()

returns all children of the root XML document

This is just an alias to self.soup.getchildren()

property soup

Returns the beautiful soup instance

class readXML(url, encoding='utf-8')

Read XML and converts to beautifulsoup data structure

easyXML accepts as input a string. This class accepts a filename instead inherits from easyXML

See also:

[easyXML](#)

Constructor

Parameters

- **data** – an XML document format
- **fixing_unicode** – use only with HGNC service to fix issue with the XML returned by that particular service. No need to use otherwise. See [HGNC](#) documentation for details.
- **encoding** – default is utf-8 used. Used to fix the HGNC XML only.

The data parameter must be a string containing the XML document. If you have an URL instead, use [readXML](#)

6.1.8 Services

ArrayExpress

Biocontainers

Interface to biocontainer

What is biocontainers

URL

<https://biocontainers.pro/>

Citation

BioContainers is an open-source project that aims to create, store, and distribute bioinformatics software containers and packages.

—From biocontainers (about), Jan 2021

class Biocontainers(*verbose=True, cache=False*)

Interface to Biocontainers service

```
>>> from bioservices import Biocontainers
>>> b = Biocontainers()
>>> b.get_tools()
```

Constructor

Parameters

verbose – set to False to prevent informative messages

get_stats()

Returns some stats about number of versions and tools

get_tools(limit=20000)

Returns all available tools.

get_versions_one_tool(tool)

Returns all versions of a given tool.

BiGG

Interface to the BiGG Models API Service

What is BiGG Models?

URL

<http://bigg.ucsd.edu>

REST

<http://bigg.ucsd.edu/api/v2>

“BiGG Models is a knowledgebase of genome-scale metabolic network reconstructions. BiGG Models integrates more than 70 published genome-scale metabolic networks into a single database with a set of standardized identifiers called BiGG IDs. Genes in the BiGG models are mapped to NCBI genome annotations, and metabolites are linked to many external databases (KEGG, PubChem, and many more).”

—BiGG Models Home Page, March 10, 2020.

class BiGG(*verbose=False, cache=False*)

Interface to the *BiGG Models* <<http://bigg.ucsd.edu/>> API Service.

```
>>> from bioservices import BiGG
>>> bigg = BiGG()
>>> bigg.search("e coli", "models")
[{'bigg_id': 'e_coli_core',
  'gene_count': 137,
  'reaction_count': 95,
```

(continues on next page)

(continued from previous page)

```
'organism': 'Escherichia coli str. K-12 substr. MG1655',
'metabolite_count': 72},
...
]
```

download(*model_id*, *format_*='json', *gzip*=True, *target*=None)

genes(*model_id*, *ids*=None)

metabolites(*model_id*=None, *ids*=None)

property models

reactions(*model_id*=None, *ids*=None)

search(*query*, *type_*)

property version

BioDBnet

This module provides a class *BioDBNet* to access to BioDBNet WS.

What is BioDBNet ?

URL

<http://biodbnet.abcc.ncifcrf.gov/>

Service

<http://biodbnet.abcc.ncifcrf.gov/webServices>

Citations

Mudunuri,U., Che,A., Yi,M. and Stephens,R.M. (2009) bioDBnet: the biological database network. *Bioinformatics*, 25, 555-556

“BioDBNet Database is a repository hosting computational models of biological systems. A large number of the provided models are published in the peer-reviewed literature and manually curated. This resource allows biologists to store, search and retrieve mathematical models. In addition, those models can be used to generate sub-models, can be simulated online, and can be converted between different representational formats. “

—From BioDBNet website, Dec. 2012

New in version 1.2.3.

Section author: Thomas Cokelaer, Feb 2014

class BioDBNet(*verbose*=True, *cache*=False)

Interface to the *BioDBNet* service

```
>>> from bioservices import *
>>> s = BioDBNet()
```

Most of the BioDBNet WSDL are available. There are functions added to the original interface such as *extra_getReactomeIds()*.

Use `db2db()` to convert from 1 database to some databases. Use `dbReport()` to get the conversion from one database to all databases.

Constructor

Parameters

verbose (*bool*) –

db2db(*input_db*, *output_db*, *input_values*, *taxon=9606*)

Retrieves models associated to the provided Taxonomy text.

Parameters

- **input_db** – input database.
- **output_db** – list of databases to map to.
- **input_values** – list of identifiers to map to the output databases

Returns

dataframe where index correspond to the input database identifiers. The columns contains the identifiers for each output database (see example here below).

```
>>> from bioservices import BioDBNet
>>> input_db = 'Ensembl Gene ID'
>>> output_db = ['Gene Symbol']
>>> input_values = ['ENSG00000121410', 'ENSG00000171428']
>>> df = s.db2db(input_db, output_db, input_values, 9606)
```

| | Gene Symbol |
|-----------------|-------------|
| Ensembl Gene ID | |
| ENSG00000121410 | A1BG |
| ENSG00000171428 | NAT1 |

dbFind(*output_db*, *input_values*, *taxon='9606'*)

dbFind method

dbFind can be used when you do not know the actual type of your identifiers or when you have a mixture of different types of identifiers. The tool finds the identifier type and converts them into the selected output if the identifiers are within the network.

Parameters

- **output_db** (*str*) – valid database name
- **input_values** (*list*) – list of identifiers to look for

Returns

a dataframe with index set to the input values.

```
>>> b.dbFind("Gene ID", ["ZMYM6_HUMAN", "NP_710159", "ENSP00000305919"])
```

| | Gene ID | Input Type |
|-----------------|---------|--------------------------|
| InputValue | | |
| ZMYM6_HUMAN | 9204 | UniProt Entry Name |
| NP_710159 | 203100 | RefSeq Protein Accession |
| ENSP00000305919 | 203100 | Ensembl Protein ID |

dbOrtho(*input_db*, *output_db*, *input_values*, *input_taxon*, *output_taxon*)

Convert identifiers from one species to identifiers of a different species

Parameters

- **input_db** – input database
- **output_db** – output database
- **input_values** – list of identifiers to retrieve
- **input_taxon** – input taxon
- **output_taxon** – output taxon

Returns

dataframe where index correspond to the input database identifiers. The columns contains the identifiers for each output database (see example here below)

```
>>> df = b.dbOrtho("Gene Symbol", "Gene ID", ["MYC", "MTOR", "A1BG"],
...               input_taxon=9606, output_taxon=10090)
      Gene ID InputValue
0    17869          MYC
1    56717          MTOR
2   117586          A1BG
```

dbReport(*input_db*, *input_values*, *taxon*=9606)

Same as [db2db\(\)](#) but returns results for all possible outputs.

Parameters

- **input_db** – input database
- **input_values** – list of identifiers to retrieve

Returns

dataframe where index correspond to the input database identifiers. The columns contains the identifiers for each output database (see example here below)

```
df = s.dbReport("Ensembl Gene ID", ['ENSG00000121410', 'ENSG00000171428'])
```

dbWalk(*db_path*, *input_values*, *taxon*=9606)

Walk through biological database network

dbWalk is a form of database to database conversion where the user has complete control on the path to follow while doing the conversion. When a input/node is added to the path the input selection gets updated with all the nodes that it can access directly.

Parameters

- **db_path** – path to follow in the databases
- **input_values** – list of identifiers

Returns

a dataframe with columns corresponding to the path nodes

A typical example is to get the Ensembl mouse homologs for Ensembl Gene ID's from human. This conversion is not possible through [db2db\(\)](#) as Homologene does not have Ensembl ID's and the input and output nodes to achieve this would both be 'Ensembl Gene ID'. It can however be run by using dbWalk as follows. Add Ensembl Gene ID to the path, then add Gene Id, Homolog - Mouse Gene ID and Ensembl Gene ID to complete the path.

```
db_path = "Ensembl Gene ID->Gene ID->Homolog - Mouse Gene ID->Ensembl Gene ID"
s.dbWalk(db_path, ["ENSG00000175899"])
```

Todo: check validity of the path

getDirectOutputsForInput(*input_db*)

Gets all the direct output nodes for a given input node

Gets all the direct output nodes for a given input node Outputs reachable by single edge connection in the bioDBnet graph.

```
b.getDirectOutputsForInput("genesymbol")
b.getDirectOutputsForInput("Gene Symbol")
b.getDirectOutputsForInput("pdbid")
b.getDirectOutputsForInput("PDB ID")
```

getInputs()

Return list of possible input database

```
s.getInputs()
```

getOutputsForInput(*input_db*)

Return list of possible output database for a given input database

```
s.getOutputsForInput("UniProt Accession")
```

BioGrid

This module provides a class BioGrid.

What is BioGrid ?

URL

<http://thebiogrid.org/>

Service

Via the PSICQUIC class

BioGRID is an online interaction repository with data compiled through comprehensive curation efforts. Our current index is version 3.2.97 and searches 37,954 publications for 638,453 raw protein and genetic interactions from major model organism species. All interaction data are freely provided through our search index and available via download in a wide variety of standardized formats.

—From BioGrid website, Feb. 2013

class BioGRID(*query=None, taxId=None, exP=None*)

Interface to BioGRID.

```
>>> from bioservices import BioGRID
>>> b = BioGRID(query=["map2k4","akt1"],taxId = "9606")
>>> interactors = b.biogrid.interactors
```

Examples:

```
>>> from bioservices import BioGRID
>>> b = BioGRID(query=["mtor", "akt1"], taxId="9606", exP="two hybrid")
>>> b.biogrid.interactors
```

One can also query an entire organism, by using the taxid as the query:

```
>>> b = BioGRID(query="6239")
```

BioMart

This module provides a class `BioModels` that allows an easy access to all the BioModel service.

What is BioMart ?

URL

<http://www.biomart.org/>

REST

<http://www.biomart.org/martservice.html>

The BioMart project provides free software and data services to the international scientific community in order to foster scientific collaboration and facilitate the scientific discovery process. The project adheres to the open source philosophy that promotes collaboration and code reuse.

—from BioMart March 2013

Note: SOAP and REST are available. We use REST for the wrapping.

class BioMart(*host=None, verbose=False, cache=False, secure=False*)

Interface to the [BioMart](#) service

BioMart is made of different views. Each view correspond to a specific **MART**. For instance the UniProt service has a [BioMart view](#).

The registry can help to find the different services available through BioMart.

```
>>> from bioservices import *
>>> s = BioMart()
>>> ret = s.registry() # to get information about existing services
```

The registry is a list of dictionaries. Some aliases are available to get all the names or databases:

```
>>> s.names          # alias to list of valid service names from registry
>>> "unimart" in s.names
True
```

Once you selected a view, you will want to select a database associated with this view and then a dataset. The datasets can be retrieved as follows:

```
>>> s.datasets("prod-intermart_1") # retrieve datasets available for this mart
```

The main issue is how to figure out the database name (here **prod-intermart_1**) ? Indeed, from the web site, what you see is the **displayName** and you must introspect the registry to get this information. In **BioServices**, we provide the *lookfor()* method to help you. For instance, to retrieve the database name of **interpro**, type:

```
>>> s = BioMart(verbose=False)
>>> s.lookfor("interpro")
Candidate:
  database: intermart_1
  MART name: prod-intermart_1
  displayName: INTERPRO (EBI UK)
  hosts: www.ebi.ac.uk
```

The display name (INTERPRO) correspond to the MART name prod-intermart_1. Let us use it to retrieve the datasets:

```
>>> s.datasets("prod-intermart_1")
['protein', 'entry', 'uniparc']
```

Now that we have the dataset names, we can select one and build a query. Queries are XML that contains the dataset name, some attributes and filters. The dataset name is one of the element returned by the datasets method. Let us suppose that we want to query **protein**, we need to add this dataset to the query:

```
>>> s.add_dataset_to_xml("protein")
```

Then, you can add attributes (one of the keys of the dictionary returned by attributes("protein")):

```
>>> s.add_attribute_to_xml("protein_accession")
```

Optional filters can be used:

```
>>> s.add_filter_to_xml("protein_length_greater_than", 1000)
```

Finally, you can retrieve the XML query:

```
>>> xml_query = s.get_xml()
```

and send the request to biomart:

```
>>> res = s.query(xml_query)
>>> len(res)
12801
# print the first 10 accession numbers
>>> res = res.split("\n")
>>> for x in res[0:10]: print(x)
['P18656',
 'Q81998',
 'O09585',
 'O77624',
 'Q9R3A1',
 'E7QZH5',
 'O46454',
 'Q9T3F4',
 'Q9TCA3',
 'P72759']
```

REACTOME example:

```
s.lookfor("reactome")
s.datasets("REACTOME")
['interaction', 'complex', 'reaction', 'pathway']

s.new_query()
s.add_dataset_to_xml("pathway")
s.add_filter_to_xml("species_selection", "Homo sapiens")
s.add_attribute_to_xml("pathway_db_id")
s.add_attribute_to_xml("_displayname")
xmlq = s.biomartQuery.get_xml()
res = s.query(xmlq)
```

Note: the biomart service is slow (in my experience, 2013-2014) so please be patient...

Constructor

URL required to use biomart change quite often. Experience has shown that BioMart class in Bioservices may fail. This is not a bioservices issue but due to API changes on server side.

For that reason the host is not filled anymore and one must set it manually.

Let us take the example of the ensembl biomart. The host is

`www.ensembl.org`

Note that there is no prefix *http* and that the actual URL looked for internally is <http://www.ensembl.org/biomart/martview>

(It used to be martservice in 2012-2016)

Another reason to not set any default host is that servers may be busy or take lots of time to initialise (if many MARTS are available). Usually, one knows which MART to look at, in which case you may want to use a specific host (e.g., www.ensembl.org) that will speed up significantly the initialisation time.

Parameters

host (*str*) – a valid host (e.g. “www.ensembl.org”, gramene.org)

List of databases are available in this webpage <http://www.biomart.org/community.html>

add_attribute_to_xml(*name*, *dataset=None*)

add_dataset_to_xml(*dataset*)

add_filter_to_xml(*name*, *value*, *dataset=None*)

attributes(*dataset*)

to retrieve attributes available for a dataset:

Parameters

dataset (*str*) – e.g. `oanatinus_gene_ensembl`

configuration(*dataset*)

to retrieve configuration available for a dataset:

Parameters

dataset (*str*) – e.g. `oanatinus_gene_ensembl`

create_attribute(*name*, *dataset=None*)

create_filter(*name*, *value*, *dataset=None*)

custom_query(***args*)

property databases

list of valid datasets

datasets(*mart*, *raw=False*)

to retrieve datasets available for a mart:

Parameters

mart (*str*) – e.g. `ensembl`. see [names](#) for a list of valid MART names the mart is the database.
see `lookfor` method or `databases` attributes

```
>>> s = BioMart(verbose=False)
>>> s.datasets("prod-intermart_1")
['protein', 'entry', 'uniparc']
```

property displayName

list of valid datasets

filters(*dataset*)

to retrieve filters available for a dataset:

Parameters

dataset (*str*) – e.g. `oanatinus_gene_ensembl`

```
>>> s.filters("uniprot").split("\n")[1].split("\t")
>>> s.filters("pathway")["species_selection"]
[Arabidopsis thaliana,Bos taurus,Caenorhabditis elegans,Canis familiaris,Danio rerio,Dictyostelium discoideum,Drosophila melanogaster,Escherichia coli,Gallus gallus,Homo sapiens,Mus musculus,Mycobacterium tuberculosis,Oryza sativa,Plasmodium falciparum,Rattus norvegicus,Saccharomyces cerevisiae,Schizosaccharomyces pombe,Staphylococcus aureus N315,Sus scrofa,Taeniopygia guttata ,Xenopus tropicalis]
```

get_datasets(*mart*)

Retrieve datasets with description

get_xml()

property host

property hosts

list of valid hosts

lookfor(*pattern*, *verbose=True*)

property marts

list of marts

property names

list of valid datasets

new_query()

query(xmlq)

Send a query to biomart

The query must be formatted in a XML format which looks like (example from <https://gist.github.com/keithshep/7776579>):

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE Query>
    <Query virtualSchemaName="default" formatter="CSV" header="0"
    ↪uniqueRows="0" count="" datasetConfigVersion="0.6">
      <Dataset name="mmusculus_gene_ensembl" interface="default">
        <Filter name="ensembl_gene_id" value="ENSMUSG000000086981"/>
        <Attribute name="ensembl_gene_id"/>
        <Attribute name="ensembl_transcript_id"/>
        <Attribute name="transcript_start"/>
        <Attribute name="transcript_end"/>
        <Attribute name="exon_chrom_start"/>
        <Attribute name="exon_chrom_end"/>
      </Dataset>
    </Query>
```

Warning: the input XML must be valid. There is no validation made in this method.

registry()

to retrieve registry information

the XML contains list of children called MartURLLocation made of attributes. We parse the xml to return a list of dictionary. each dictionary correspond to one MART.

aliases to some keys are provided: names, databases, displayName

property valid_attributes

list of valid datasets

version(mart)

Returns version of a mart

Parameters

mart (*str*) – e.g. ensembl

BioModels

This module provides a class *BioModels* to access to BioModels WS.

What is BioModels ?**URL**

<http://www.ebi.ac.uk/biomodels/>

Service

<http://www.ebi.ac.uk/biomodels>

Citations

please visit <https://www.ebi.ac.uk/biomodels/citation> for details

“BioModels is a repository of mathematical models of biological and biomedical systems. It hosts a vast selection of existing literature-based physiologically and pharmaceutically relevant mechanistic models in standard formats. Our mission is to provide the systems modelling community with reproducible, high-quality, freely-accessible models published in the scientific literature.”

—From BioModels website, March 2020

class BioModels(*verbose=True*)

Interface to the [BioModels](#) service

```
from bioservices import BioModels
bm = BioModels()
model = bm.get_model('BIOMD0000000299')
```

Previous API had several functions such as *getAuthorsByModelId*. This is easy to mimic with the new API:

```
bm = BioModels()
models = bm.get_all_models()
[x['submitter'] for x in res if x[] == "MODEL1204280003"][0]
```

This is also true for *getDateLastModifByModelId* and *getModelNameById* if one use the field *lastModified* or *name*. There was the ability to search for models based on their CHEBI identifiers, which is not supported anymore; this concerns functions *getModelsIdByChEBI*, *getModelsIdByChEBIID*, *getSimpleModelsByChEBIIDs*, *getSimpleModelsRelatedWithChEBI*. For other searches related to Reactome, Uniprot identifiers or GO terms, the *search()* method should work:

```
bm.search("P10113")
bm.search("REACT_33")
bm.search("GO:0006919")
```

constructor

Parameters

verbose (*bool*) –

get_all_models(*chunk=100*)

Return all models

get_model(*model_id*, *frmt='json'*)

Fetch information about a given model at a particular revision.

get_model_download(*model_id*, *filename=None*, *output_filename=None*)

Download a particular file associated with a given model or all its files as a COMBINE archive.

Parameters

- **model_id** – a valid BioModels identifier
- **filename** (*str*) – this is the requested filename to be found in the model
- **output_filename** (*str*) – if you request a different output filename, use this parameter
- **frmt** – format of the output (json, xml, html)

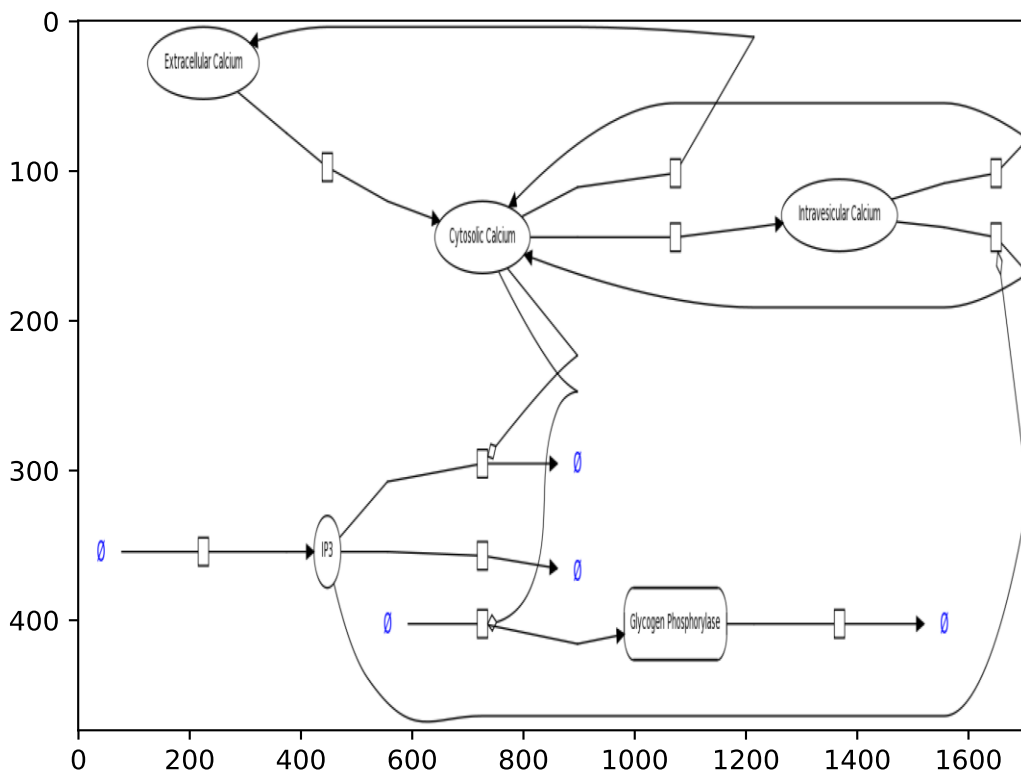
Returns

nothing. This function save the model into a ZIP file called after the model identifier. If parameter *filename* is specified, then the output file is the requested filename (if found)

```
bm.get_model_download("BIOMD00000000100", filename="BIOMD00000000100.png")
bm.get_model_download("BIOMD00000000100")
```

This function can retrieve all files in a ZIP archive or a single image. In the example below, we retrieve the PNG and plot it using matplotlib. Using your favorite image viewer, you should get a better resolution. Or just download the SVG version of the model.

```
from bioservices import BioModels
bm = BioModels()
from easydev import TempFile
with TempFile(suffix=".png") as fout:
    bm.get_model_download("BIOMD00000000100",
        filename="BIOMD00000000100.png",
        output_filename=fout.name)
from pylab import imshow, imread
imshow(imread(fout.name), aspect="auto")
```



get_model_files(model_id, frmt='json')

Extract metadata information of model files of a particular model

Parameters

- **model_id** – a valid BioModels identifier
- **frmt** – format of the output (json, xml)

get_p2m_missing(*frmt='json'*)

Retrieve all models in Path2Models that are now only available indirectly, through the representative model for the corresponding genus

Parameters

frmt (*str*) – the format of the result (xml, csv, json)

Returns

list of model identifiers

get_p2m_representative(*model, frmt='json'*)

Retrieve a representative model in Path2Models

Get the representative model identifier for a given missing model in Path2Models. This endpoint accepts as parameters a mandatory model identifier and an optional response format

Parameters

- **model** (*str*) – The identifier of a model of interest
- **frmt** (*str*) – the format of the result (xml, csv, json)

get_p2m_representatives(*models, frmt='json'*)

Find the replacement accessions for a set of Path2Models entries

Get the representative model identifiers of a set of given missing models in Path2Models. This end point expects a comma-separated list of model identifiers (without any surrounding whitespace) and an optional response format. Examples: BMID0000000112902, BMID0000000009880, BMID0000000027397.

Parameters

- **model** (*str*) – The model identifiers separated by commas, or as a list.
- **frmt** (*str*) – the format of the result (xml, csv, json)

```
from bioservices import BioModels
bm = BioModels()
bm.get_p2m_representatives("BMID0000000112902, BMID0000000009880, BMID0000000027397
→")
```

get_pdgsmm_missing(*frmt='json'*)

Retrieve the identifiers of all PDGSMM entries that are no longer directly accessible

Parameters

frmt (*str*) – the format of the result (xml, csv, json)

Returns

list of model identifiers

get_pdgsmm_representative(*model, frmt='json'*)

Retrieve a representative model in PDGSMM

Get the representative model identifier for a given missing model in PDGSMM. This endpoint accepts as parameters a mandatory model identifier and an optional response format.

Parameters

- **model** (*str*) – The identifier of a model of interest
- **frmt** (*str*) – the format of the result (xml, csv, json)

get_pdgsmm_representatives(*models*, *frmt*='json')

Find the replacement accessions for a set of PDFSSM

Get the representative model identifiers of a set of given missing models in PDGSMM. This end point expects a comma-separated list of model identifiers (without any surrounding whitespace) and an optional response format. Examples: MODEL1707110145,MODEL1707112456,MODEL1707115900.

Parameters

- **model** (*str*) – The model identifiers separated by commas, or as a list.
- **frmt** (*str*) – the format of the result (xml, csv, json)

search(*query*, *offset*=None, *numResults*=None, *sort*=None, *frmt*='json')

Search models of interest via keywords.

Examples: PUBMED:"27869123" to search models associated with the PubMed record identified by 27869123.

Parameters

- **query** (*str*) – search query. colon character must be escaped
- **offset** (*int*) – number of items to skip before starting to collect the result set
- **numResults** (*int*) – number of items to return
- **sort** (*str*) – sort criteria in {id-asc, relevance-asc, relevance-desc, first_author-asc, first_author, name-asc, name-desc, publication_year-asc, publication_year-desc}
- **frmt** (*str*) – format of the output (json, xml)

search_download(*models*, *output_filename*='models.zip', *force*=False)

Returns models (XML) corresponding to a list of model identifiers.

Parameters

- **models** (*str*) – list of model identifiers using comma to separate them. Could be a list of string (e.g 'BIOMD1,BIOMD2' or ['BIOMD1', 'BIOMD2'])
- **output_filename** (*str*) – file used to save the models. This is a zipped file. If the file exists, you must use the *force** parameter

Todo: if no models are found (typos), an error message is printed. if one model is not found, there is no warning or errors. Could be nice to have a warning by introspecting the number of models in the output file

search_parameter(*query*, *start*=0, *size*=10, *sort*=None, *frmt*='json')

Search for parameters of a model

Details BioModels Parameters is a resource that facilitates easy search and retrieval of parameter values used in the SBML models stored in the BioModels repository. Users can search for a model entity (e.g. a protein or drug) to retrieve the rate equations describing it; the associated parameter values and the initial concentration from the SBML models in BioModels. Although these data are directly extracted from the curated SBML models, they are not individually curated or validated; rather presented as such in the table below. Hence BioModels Parameters table will only provide a quick overview of available parameter values for guidance and original model should be referred to understand the complete context of the parameter usage.

Parameters

- **query** (*str*) – A query to search against the model parameter values.
- **start** (*int*) – if is the offset of the result set (default 0)
- **size** (*int*) – number of items to display per page
- **sort** (*str*) – model or entity
- **fmt** (*str*) – the format of the result (xml, csv, json)

```
bm.search_parameter("MAPK", size=100, sort="entity")
```

ChEBI

This module provides a class *ChEBI*

What is ChEBI

URL

<https://www.ebi.ac.uk/chebi/init.do>

WSDL

<http://www.ebi.ac.uk/webservices/chebi/2.0/webservice>

“The database and ontology of Chemical Entities of Biological Interest

—From ChEBI web page June 2013

class *ChEBI*(*verbose=False*)

Interface to ChEBI

```
>>> from bioservices import *
>>> ch = ChEBI()
>>> res = ch.getCompleteEntity("CHEBI:27732")
>>> res.smiles
CN1C(=O)N(C)c2ncn(C)c2C1=O
```

Constructor

Parameters

verbose (*bool*) –

conv(*chebiId*, *target*)

Calls *getCompleteEntity()* and returns the identifier of a given database

Parameters

- **chebiId** (*str*) – a valid ChEBI identifier (string)
- **target** – the identifier of the database

Returns

the identifier

```
>>> ch.conv("CHEBI:10102", "KEGG COMPOUND accession")
['C07484']
```

getAllOntologyChildrenInPath(*chebiId*, *relationshipType*, *onlyWithChemicalStructure=False*)

Retrieves the ontology children of an entity including the relationship type

Parameters

- **chebiId** (*str*) – a valid ChEBI identifier (string)
- **relationshipType** (*str*) – one of “is a”, “has part”, “has role”, “is conjugate base of”, “is conjugate acid of”, “is tautomer of”, “is enantiomer of”, “has functional parent”, “has parent hybriide” “is substituent group of”

```
>>> ch.getAllOntologyChildrenInPath("CHEBI:27732", "has part")
```

getCompleteEntity(*chebiId*)

Retrieves the complete entity including synonyms, database links and chemical structures, using the ChEBI identifier.

param str chebiId

a valid ChEBI identifier (string)

return

an object containing fields such as mass, names, smiles

```
>>> from bioservices import *
>>> ch = ChEBI()
>>> res = ch.getCompleteEntity("CHEBI:27732")
>>> res.mass
194.19076
```

The returned structure is the raw object returned by the API. You can extract names from other sources for instance:

```
>>> [x[0] for x in res.DatabaseLinks if x[1].startswith("KEGG")]
[C07481, D00528]
>>> [x[0] for x in res.DatabaseLinks if x[1].startswith("ChEMBL")]
[116485]
```

See also:

[*conv\(\)*](#), [*getCompleteEntity\(\)*](#)

getCompleteEntityByList(*chebiIdList=[]*)

Given a list of ChEBI accession numbers, retrieve the complete Entities.

The maximum size of this list is 50.

See also:

[*getCompleteEntity\(\)*](#)

getLiteEntity(*search*, *searchCategory='ALL'*, *maximumResults=200*, *stars='ALL'*)

Retrieves list of entities containing the ChEBI ASCII name or identifier

Parameters

- **search** – search string or category.
- **searchCategory** – filter with category. Can be ALL,
- **maximumResults** (*int*) – (default is 200)
- **stars** (*str*) – filters that can be set to “TWO ONLY”, “ALL”, “THREE ONLY”

The input parameters are a search string and a search category. If the search category is null then it will search under all fields. The search string accepts the wildcard character "*" and also unicode characters. You can get maximum results upto 5000 entries at a time.

```
>>> ch.getLiteEntity("CHEBI:27732")
[<LiteEntity>{
  chebiId = "CHEBI:27732"
  chebiAsciiName = "caffeine"
  searchScore = 4.77
  entityStar = 3
}]
>>> res = ch.getLiteEntity("caffeine")
>>> res = ch.getLiteEntity("caffeine", maximumResults=10)
>>> len(res)
10
```

See also:

[`getCompleteEntity\(\)`](#)

getOntologyChildren(*chebiId*)

Retrieves the ontology children of an entity including the relationship type

Parameters

chebiId (*str*) – a valid ChEBI identifier (string)

getOntologyParents(*chebiId*)

Retrieves the ontology parents of an entity including the relationship type

Parameters

chebiId (*str*) – a valid ChEBI identifier (string)

**getStructureSearch(*structure*, *mode*='MOLFILE', *structureSearchCategory*='SIMILARITY',
totalResults=50, *tanimotoCutoff*=0.25)**

Does a substructure, similarity or identity search using a structure.

Parameters

- **structure** (*str*) – the input structure
- **mode** (*str*) – type of input (MOLFILE, SMILES, CML) (note that the API uses type but this is a python keyword)
- **structureSearchCategory** (*str*) – category of the search. Can be "SIMILARITY", "SUBSTRUCTURE", "IDENTITY"
- **totalResults** (*int*) – limit the number of results to 50 (default)
- **tanimotoCuoff** – limit results to scores higher than this parameter

```
>>> ch = ChEBI()
>>> smiles = ch.getCompleteEntity("CHEBI:27732").smiles
>>> ch.getStructureSearch(smiles, "SMILES", "SIMILARITY", 3, 0.25)
```

getUpdatedPolymer(*chebiId*)

Returns the UpdatedPolymer object

Parameters

- **chebiId** (*str*) –

- **chebiId** – a valid ChEBI identifier (string)

Returns

an object with information as described below.

The object contains the updated 2D MolFile structure, GlobalFormula string containing the formulae for each repeating-unit, the GlobalCharge string containing the charge on individual repeating-units and the primary ChEBI ID of the polymer, even if the secondary Identifier was passed to the web-service.

ChEMBL

This module provides a class *ChEMBL*

What is ChEMBL**URL**

<https://www.ebi.ac.uk/chembl>

REST

<https://www.ebi.ac.uk/chembl/api/data>

“Using the ChEMBL web service API users can retrieve data from the ChEMBL database in a programmatic fashion. The following list defines the currently supported functionality and defines the expected inputs and outputs of each method.”

—From ChEMBL web page Dec 2012

```
class ChEMBL(verbose=False, cache=False)
```

New ChEMBL API bioservices 1.6.0

Resources

ChEMBL database is made of a set of resources. We recommend to look at <https://arxiv.org/pdf/1607.00378.pdf>

Here we first create an instance and retrieve the first 1000 molecules from the database using the **limit** parameter.

```
>>> from bioservices import ChEMBL
>>> c = ChEMBL()
>>> res = c.get_molecule(limit=1000)
```

The returned object is a list of 1000 records, each of them being a dictionary. The **molecule** resource is actually a very large one and one may want to skip some entries. This is possible using the **offset** parameter as follows:

```
# Retrieve 1000 molecules skipping the first 50
res = c.get_molecule(limit=1000, offset=50)
```

If you want to know all resources available and the number of entries in each resources, use:

```
status = c.get_status_resources()
```

For instance, you should be able to get the total number of entries in the *mechanism* resource is about 5,000:

```
print(status['mechanism'])
```

To retrieve all entries from the mechanism resource, you can either set limit to a value large enough:

```
res = c.get_mechanism(limit=1000000)
```

or simply set it to -1:

```
res = c.get_mechanism(limit=-1)
```

All resources methods behaves in the same way.

Those resources methods are: `get_activity()`, `get_assay()`, `get_atc_class()`, `get_binding_site()`, `get_biotherapeutic()`, `get_cell_line()`, `get_chembl_id_lookup()`, `get_compound_record()`, `get_compound_structural_alert()`, `get_document()`, `get_document_similarity()`, `get_document_term()`, `get_drug()`, `get_drug_indication()`, `get_go_slim()`, `get_mechanism()`, `get_metabolism()`, `get_molecule()`, `get_molecule_form()`, `get_protein_class()`, `get_source()`, `get_target()`, `get_target_component()`, `get_target_prediction()`, `get_target_relation()`, `get_tissue()`.

3 ways of getting items

1. Retrieve everything:

```
c.get_molecule(limit=-1)
```

2. Retrieve a specific entry:

```
c.get_molecule("CHEMBL24")
```

3. Retrieve a set of entries:

```
c.get_molecule(["CHEMBL24", "CHEMBL2"])
```

Filtering and Ordering

For ordering the results, we provide a simple method `order_by()` that allows to sort the dictionary according to values in a specific key.

Any data returned by a resource method (a list of dictionary) can be process through this method:

```
c = ChEMBL()
data = c.get_drug(limit=100)
ordered_data = c.order_by(data, 'chirality')
```

If you want to order using a key within a key, for instance order by molecular weight stored in the `molecular_properties` key, use the double underscore method as follows:

```
c = ChEMBL()
data = c.get_drug(limit=100)
ordered_data = c.order_by(data, 'molecular_properties__mw_freebase')
```

For filtering, it is possible to apply search filters to any resources. For example, it is possible to return all ChEMBL targets that contain the term 'kinase' in the `pref_name` attribute:

```
c.get_target(filters='pref_name__contains=kinase')
```

The pattern for applying a filter is as follows:

```
[field]__[filter_type]=[value]
```

where field has to be found by the user. Simply introspect the content of an item returned by the resource. For instance:

```
c.get_target(limit=1) # to get one entry
```

Let us consider the case of the **molecule** resource. You can retrieve the first 10 molecules using e.g.:

```
res = c.get_molecule(limit=10)
```

If you look at the first entry using `res[0]`, you will get about 38 keys. For instance **molecule_properties** or **molecule_chembl_id**.

You can filter the molecules to keep only the molecule_chembl_id that match either ChEMBL25 or ChEMBL1000 using:

```
res = c.get_molecule(filters='molecule_chembl_id__in=ChEMBL25,ChEMBL1000')
```

For **molecule_properties**, this is actually a dictionary. For instance, inside the **molecule_properties** field, you have the molecular weight (`mw_freebase`). So to apply this filter, you need to use the following code (to keep molecules with molecular weight greater than 300):

```
res = c.get_molecule(filters='molecule_properties__mw_freebase__gte=300')
```

Here are the different types of filtering:

| Filter Type | Description |
|----------------|---|
| exact (iexact) | Exact match with query |
| contains | wild card search with query |
| startswith | starts with query |
| endswith | ends with query |
| regex | regular expression query |
| gt (gte) | Greater than (or equal) |
| lt (lte) | Less than (or equal) |
| range | Within a range of values |
| in | Appears within list of query values |
| isnull | Field is null |
| search | Special type of filter allowing a full text search based on Solr queries. |

Several filters can be applied at the same time using a list:

```
filters = ['molecule_properties__mw_freebase__gte=300']
filters += ['molecule_properties__alogp__gte=3']
res = c.get_molecule(filters)
```

Use Cases: (inspired from ChEMBL documentation)

Search molecules by synonym:

```
>>> from bioservices import ChEMBL
>>> c = ChEMBL()
>>> res = c.search_molecule('aspirin')
```

or SMILE, or InChIKey, or ChEMBLID:

```
>>> res = c.get_molecule("CC(=O)Oc1ccccc1C(=O)O")
>>> res = c.get_molecule("BSYNRYMUTXBXSQ-UHFFFAOYSA-N")
>>> res = c.get_molecule('CHEMBL25')
```

Several molecules at the same time can also be retrieved using lists:

```
>>> res = c.get_molecule(['CHEMBL25', 'CHEMBL2'])
```

Search target by gene name:

```
>>> res = c.search_target("GABRB2")
>>> len(res['targets'])
18
```

or directly in the target synonym field:

```
>>> res = c.get_target(filters='target_synonym__icontains=GABRB2')
```

Note: Not sure what is the difference between `icontains` vs `contains`. It looks like `icontains` is more permissive (you get more entries with `icontains`).

Having a list of molecules ChEMBL IDs in a list, get uniprot accession numbers that map to those compounds:

```
# First, get some IDs of approved drugs (about 2000 molecules)
c = ChEMBL()
drugs = c.get_approved_drugs()
IDs = [x['molecule_chembl_id'] for x in drugs]

# we jump from compounds to targets through activities
# Here this is a one to many mapping so we initialise a default
# dictionary.
compound2target = defaultdict(set)

filter = "molecule_chembl_id__in={}"
for i in range(0, len(IDs), 50):
    activities = c.get_activity(filter.format(IDs[i:i+50]))
    # get target ChEMBL IDs from activities
    for act in activities:
        compound2target[act['molecule_chembl_id']].add(act['target_chembl_id'])

# What we need is to get targets for all targets found in the previous
# step. For each compound/drug there are hundreds of targets though. And
# we will call the get_target for each list of hundreds targets. This
# will take forever. Instead, because there are *only* 12,000 targets,
# let us download all of them ! This took about 4 minutes on this test but
# if you use the cache, next time it will be much much quicker. This is
# not down at the activities level because there are too many entries

targets = c.get_target(limit=-1)

# identifies all target chembl id to easily retrieve the entry later on
target_names = [target['target_chembl_id'] for target in targets]
```

(continues on next page)

(continued from previous page)

```
# retrieve all uniprot accessions for all targets of each compound
for compound, targs in compounds2targets.items():
    accessions = set()
    for target in targs:
        index = target_names.index(target)
        accessions = accessions.union([comp['accession']
                                       for comp in targets[index]['target_components']])
    compounds2targets[compound] = accessions
```

In version 1.6.0 of bioservices, you can simply use:

```
res = c.compounds2targets(IDs)
```

Get Target type count for all targets:

```
import collections
collections.Counter([x['target_type'] for x in targets])
```

Find compounds similar to given SMILES query with similarity threshold of 85%:

```
>>> SMILE = "CN(CCCN)c1cccc2ccccc12"
>>> c.get_similarity(SMILE, similarity=70)
```

Find compounds similar to aspirin (ChEMBL25) with similarity threshold of 70%:

```
# search for aspirin in all molecules and from first hit
# get the ChEMBL ID
>>> molecules = c.search_molecule("aspirin")['molecules']
>>> chembl_id = molecules[0]['molecule_chembl_id']
# now use the :meth:`get_similarity` given the ID
>>> res = c.get_similarity(chembl_id, similarity=70)
```

Perform substructure search using SMILES or ChEMBLID:

```
>>> res = c.get_substructure("CN(CCCN)c1cccc2ccccc12")
>>> res = c.get_substructure("CHEMBL25")
```

Obtain the pChEMBL value for compound:

```
res = c.get_activity(filters=['pchembl_value__isnull=False',
                             'molecule_chembl_id=CHEMBL25'])
```

Obtain the pChEMBL value for compound and target:

```
res = c.get_activity(filters=['pchembl_value__isnull=False',
                             'molecule_chembl_id=CHEMBL25',
                             'target_chembl_id=CHEMBL612545'])
```

Get all approved drugs:

```
c.get_approved_drugs(max_phase=4)
```

Get approved drugs for lung cancer

The ChEMBL API significantly changed in 2018 and the nez version of bioservices (1.6.0) had to change the API as well, which has been simplified.

Here below are some correspondances between the previous and the new API.

| bioservices before 1.6.0 | After 1.6.0 |
|--|------------------------|
| get_compounds_substructure | get_substructure |
| get_compounds_similar_to_SMILES | get_similarity(SMILE) |
| get_compounds_by_chemblId(ID) | get_similarity(ID) |
| get_individual_compounds_by_inChiKey | get_molecule(inchikey) |
| get_compounds_by_chemblId_form | get_molecule_form |
| get_compounds_by_chemblId_drug_mechanism | get_mechanism(ID) |
| get_target_by_chemblId(ID) | get_target(ID) |
| get_image_of_compounds_by_chemblId | get_image |
| etc | |

References

- <https://arxiv.org/pdf/1607.00378.pdf>
- <https://www.ebi.ac.uk/chembl/api/data/docs>

Constructor

Parameters

- **name** (*str*) – a name for this service
- **url** (*str*) – its URL
- **verbose** (*bool*) – prints informative messages if True (default is True)
- **requests_per_sec** – maximum number of requests per seconds are restricted to 3. You can change that value. If you reach the limit, an error is raise. The reason for this limitation is that some services (e.g., NCBI) may black list you IP. If you need or can do more (e.g., ChEMBL does not seem to have restrictions), change the value. You can also have several instance but again, if you send too many requests at the same, your future requests may be retracted. Currently implemented for REST only

All instances have an attribute called `logging` that is an instance of the `logging` module. It can be used to print information, warning, error messages:

```
self.logging.info("informative message")
self.logging.warning("warning message")
self.logging.error("error message")
```

The attribute `debugLevel` can be used to set the behaviour of the logging messages. If the argument `verbose` is True, the `debugLevel` is set to INFO. If `verbose` is False, the `debugLevel` is set to WARNING. However, you can use the `debugLevel` attribute to change it to one of DEBUG, INFO, WARNING, ERROR, CRITICAL. `debugLevel=WARNING` means that only WARNING, ERROR and CRITICAL messages are shown.

`compounds2accession(compounds)`

For each compound, identifies the target and corresponding UniProt accession number

This is not part of ChEMBL API

```
# we recommend to use cache if you use this method regularly
c = Chembl(cache=True)
drugs = c.get_approved_drugs()

# to speed up example
drugs = drugs[0:20]
IDs = [x['molecule_chembl_id'] for x in drugs]

c.compounds2accession(IDs)
```

get_ATC(*limit=20, offset=0, filters=None*)

WHO ATC Classification for drugs

`c.get_atc() c['atc']`

Note: `get_molecule` returns 'molecules' and likewise all methods return a dictionary whose key is the plural of the method name. This is quite consistent through the API except for that one because it is an acronym

get_activity(*query=None, limit=20, offset=0, filters=None*)

Activity values recorded in an Assay

get_approved_drugs(*max_phase=4, maxdrugs=1000000*)

Return all approved drugs

Parameters

max_phase – 4 by default for approved drugs.

get_assay(*query=None, limit=20, offset=0, filters=None*)

Assay details as reported in source Document/Dataset

```
>>> c.get_assay("CHEMBL1217643")
```

get_binding_site(*limit=20, offset=0, filters=None*)

Target binding site definition

get_biotherapeutic(*limit=20, offset=0, filters=None*)

Biotherapeutic molecules, which includes HELM notation and sequence data

get_cell_line(*limit=20, offset=0, filters=None*)

Cell line information

get_chembl_id_lookup(*query=None, limit=20, offset=0, filters=None*)

Look up ChEMBL Id entity type

get_compound_record(*query=None, limit=20, offset=0, filters=None*)

Occurence of a given compound in a specific document

get_compound_structural_alert(*query=None, limit=20, offset=0, filters=None*)

Indicates certain anomaly in compound structure

get_document(*query=None, limit=20, offset=0, filters=None*)

Document/Dataset from which Assays have been extracted

get_document_similarity(*query=None, limit=20, offset=0, filters=None*)

Provides documents similar to a given one

get_document_term(*query=None, limit=20, offset=0, filters=None*)

Provides keywords extracted from a document using the TextRank algorithm

get_drug(*query=None, limit=20, offset=0, filters=None*)

Approved drugs information, including (but not limited to) applicants, patent numbers and research codes

get_drug_indication(*query=None, limit=20, offset=0, filters=None*)

Joins drugs with diseases providing references to relevant sources

get_go_slim(*query=None, limit=20, offset=0, filters=None*)

GO slim ontology

get_image(*query, dimensions=500, format='png', save=True, view=True, engine='indigo'*)

Get the image of a given compound in PNG png format.

Parameters

- **query** (*str*) – a valid compound ChEMBLId or a list/tuple of valid compound ChEMBLIds.
- **format** – png, svg. json not supported
- **dimensions** (*int*) – size of image in pixels. An integer z ($1 \leq z \leq 500$)
- **save** –
- **view** (*bool*) –
- **engine** – Defaults to rdkit. can be rdkit or indigo
- **view** – show the image if set to True.

Returns

the path (list of paths) used to save the figure (figures) (different from Chembl API)

```
>>> from pylab import imread, imshow
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> res = s.get_image(31863)
>>> imshow(imread(res['filenames'][0]))
```

Todo: ignorecoords option

get_mechanism(*query=None, limit=20, offset=0, filters=None*)

Mechanism of action information for FDA-approved drugs

get_metabolism(*query=None, limit=20, offset=0, filters=None*)

Metabolic pathways with references

get_molecule(*query=None, limit=20, offset=0, filters=None*)

Returns some molecules

Parameters

- **limit** – number of molecules to retrieve
- **offset** – molecules to ignore before retrieving molecules.

Returns

a dictionary with keys *page_meta* and *molecules*.

There are 1,800,000 molecules (Jan 2019). You can only retrieve 1,000 molecule at most using the *limit* parameter. With a loop you can retrieve molecules in some range.

```
c.get_molecule('QFFGVLORLPOAEC-SNVBAGLBSA-N')
c.get_molecule("CC(=O)Oc1ccccc1C(=O)O")
```

get_molecule_form(*query=None, limit=20, offset=0, filters=None*)

Relationships between molecule parents and salts

```
>>> s.get_molecule_form("CHEMBL2")['molecule_forms']
[{'is_parent': 'True',
  'molecule_chembl_id': 'CHEMBL2',
  'parent_chembl_id': 'CHEMBL2'},
 {'is_parent': 'False',
  'molecule_chembl_id': 'CHEMBL1558',
  'parent_chembl_id': 'CHEMBL2'},
 {'is_parent': 'False',
  'molecule_chembl_id': 'CHEMBL1347191',
  'parent_chembl_id': 'CHEMBL2'}]
```

get_organism(*query=None, limit=20, offset=0, filters=None*)

get_protein_class(*query=None, limit=20, offset=0, filters=None*)

Protein family classification of TargetComponents

get_similarity(*structure, similarity=80, limit=20, offset=0, filters=None*)

Molecule similarity search

Parameters

- **structure** – provide a valid / existing substructure in SMILE format to look for in all molecules:
- **similarity** – must be an integer greater than 70 and less than 100

Returns

list of **molecules** corresponding to the search

```
>>> from bioservices import ChEMBL
>>> c = ChEMBL()
>>> res = c.get_similarity("CC(=O)Oc1ccccc1C(=O)O", 80)
>>> res['molecules']
```

Here are more examples:

```
# Similarity (80% cut off) search for against ChEMBL using
# aspirin SMILES string
c.get_similarity("CC(=O)Oc1ccccc1C(=O)O") # 80 by default

# Similarity (80% cut off) search for against ChEMBL using
# aspirin CHEMBL_ID
c.get_similarity("CHEMBL25")
```

(continues on next page)

(continued from previous page)

```
# Similarity (80% cut off) search for against ChEMBL
# using aspirin InChI Key
c.get_similarity("BSYNRYMUTXBXSQ-UHFFFAOYSA-N")
```

The 'Substructure' and 'Similarity' web service resources allow for the chemical content of ChEMBL to be searched. Similar to the other resources, these search based resources except filtering, paging and ordering arguments. These methods accept SMILES, InChI Key and molecule ChEMBL_ID as arguments and in the case of similarity searches an additional identity cut-off is needed. Some example molecule searches are provided in the table below.

Searching with InChI key is only possible for InChI keys found in the ChEMBL database. The system does not try and convert InChI key to a chemical representation.

get_source(*query=None, limit=20, offset=0, filters=None*)

Document/Dataset source

get_status()

Return version of the DB and number of entries

Returns the number of entries for activities, compound_records, distinct_compounds (molecule), publications (document), targets, etc...

See also:

[*get_status_resources\(\)*](#)

get_status_resources()

Return number of entries for all resources

Note: not in the ChEMBL API.

Changed in version 1.7.3: (removed target_prediction and document_term)

get_substructure(*structure, limit=20, offset=0, filters=None*)

Molecule substructure search

Parameters

structure – provide a valid / existing substructure in SMILE format to look for in all molecules:

Returns

list of molecules corresponding to the search

```
>>> from bioservices import ChEMBL
>>> c = ChEMBL()
>>> res = c.get_substructure("CC(=O)Oc1ccccc1C(=O)O")
```

Other examples:

```
# Substructure search for against ChEMBL using aspirin
# SMILES string
c.get_substructure("CC(=O)Oc1ccccc1C(=O)O")

# Substructure search for against ChEMBL using aspirin
# ChEMBL_ID
```

(continues on next page)

(continued from previous page)

```
c.get_substructure("ChEMBL25")

# Substructure search for against ChEMBL using aspirin
# InChIKey
c.get_substructure("BSYNRYMUTXBXSQ-UHFFFAOYSA-N")
```

The 'Substructure' and 'Similarity' web service resources allow for the chemical content of ChEMBL to be searched. Similar to the other resources, these search based resources except filtering, paging and ordering arguments. These methods accept SMILES, InChI Key and molecule ChEMBL_ID as arguments and in the case of similarity searches an additional identity cut-off is needed. Some example molecule searches are provided in the table below.

Searching with InChI key is only possible for InChI keys found in the ChEMBL database. The system does not try and convert InChI key to a chemical representation.

get_target(query=None, limit=20, offset=0, filters=None)

Targets (protein and non-protein) defined in Assay

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> resjson = s.get_targetd('ChEMBL240')
```

get_target_component(query=None, limit=20, offset=0, filters=None)

Target sequence information (A Target may have 1 or more sequences)

```
res = c.get_target_component(1)
res['sequence']
```

get_target_prediction(query=None, limit=20, offset=0, filters=None)

Predicted binding of a molecule to a given biological target

```
>>> res = c.get_target_prediction(1)
>>> res['molecule_chembl_id']
'ChEMBL2'
```

get_target_relation(query=None, limit=20, offset=0, filters=None)

Describes relations between targets

```
>>> c.get_target_relation('ChEMBL261')
{'related_target_chembl_id': 'ChEMBL2095180',
 'relationship': 'SUBSET OF',
 'target_chembl_id': 'ChEMBL261'}
```

get_tissue(query=None, limit=20, offset=0, filters=None)

Tissue classification

```
c.get_tissue(filters=['pref_name__contains=cervix'])
```

get_xref_source(query=None, limit=20, offset=0, filters=None)

order_by(data, name, ascending=True)

Ordering data

we use same API as ChEMBL API using the double underscore to indicate a hierarchy in the dictionary. So to access to d['a']['b'], we use a__b as the input **name** parameter. We only allows 3 levels e.g., a__b__c

```
data = c.get_molecules()
data1 = c.order_by(data, 'molecule_chembl_id')
data2 = c.order_by(data, 'molecule_properties__alogp')
```

Note: the ChEMBL API allows for ordering but we do not use that API. Instead, we provide this generic function.

search_activity(*query*, *limit=20*, *offset=0*)

Activity values recorded in an Assay

search_assay(*query*, *limit=20*, *offset=0*)

Assay details as reported in source document

search_chembl_id_lookup(*query*, *limit=20*, *offset=0*)

Look up ChEMBL Id entity type

search_document(*query*, *limit=20*, *offset=0*)

Document/Dataset from which Assays have been extracted

search_molecule(*query*, *limit=20*, *offset=0*)

search_protein_class(*query*, *limit=20*, *offset=0*)

search_target(*query*, *limit=20*, *offset=0*)

Targets (protein and non-protein) defined in Assay

COG

Interface to some part of the UniProt web service

What is COG service?

URL

<https://www.ncbi.nlm.nih.gov/research/cog/webservices/>

Citation

Database of Clusters of Orthologous Genes (COGs)

—From COG web site, Jan 2021

class COG(*verbose=False*, *cache=False*)

Interface to the COG service

from bioservices import COG c = COG() cogs = c.get_all_cogs() # This is a pandas dataframe

Constructor

get_all_cogs_definition()

Get all COG Definitions:

get_cog_definition_by_cog_id(*cog_id*)

Get specific COG Definitions by COG: COG0003

get_cog_definition_by_name(*cog*)

Get specific COG Definitions by name: Thiamin-binding stress-response protein YqgV, UPF0045 family

get_cogs(*page=1*)

Get COGs. Unfortunately, the API sends 10 COGS at a time given a specific page.

The dictionary returned contains the results, count, previous and next page.

get_cogs_by_assembly_id(*assembly_id*)

Filter COGs by assembly ID: GCA_000007185.1

get_cogs_by_category(*category*)

Filter COGs by Taxonomic Category: ACTINOBACTERIA

get_cogs_by_category_(*protein*)

Filter COGs by Protein name: AJP49128.1

get_cogs_by_category_id(*category*)

Filter COGs by Taxonomic Category taxid: 651137

get_cogs_by_gene(*gene*)

Filter COGs by gene tag: MK0280

get_cogs_by_id(*cog_id*)

Filter COGs by COG ID tag: COG0003

get_cogs_by_id_and_category(*cog_id*, *category*)

Filter COGs by COG id and Taxonomy Categories: COG0004 and CYANOBACTERIA

get_cogs_by_id_and_organism(*cog_id*, *organism*)

Filter COGs by COG id and organism: COG0004 and Escherichia_coli_K-12_sub_MG1655

get_cogs_by_organism(*name*)

Filter COGs by organism name: Nitrosopumilus_maritimus_SCM1

get_cogs_by_taxon_id(*taxon_id*)

Filter COGs by taxid: 1229908

get_taxonomic_categories()

Get all Taxonomic Categories:

get_taxonomic_category_by_name(*name*)

Get specific Taxonomic Category by name: ALPHAPROTEOBACTERIA

ENA

This module provides a class [ENA](#)

What is ENA**URL**

<https://www.ebi.ac.uk/ena>

The European Nucleotide Archive (ENA) provides a comprehensive record of the world's nucleotide sequencing information, covering raw sequencing data, sequence assembly information and functional annotation.

New in version 1.4.4.

class ENA(verbose=False, cache=False)

Interface to ChEMBL

Here is a quick example to retrieve a target given its ChEMBL Id

```
>>> from bioservices import ENQ
>>> s = ENA(verbose=False)
```

Retrieve read domain metadata in XML format:

```
print(e.get_data('ERA000092', 'xml'))
```

Retrieve assemble and annotated sequences in fasta format:

```
print(e.get_data('A00145', 'fasta'))
```

The range parameter can be used in combination to retrieve a subsequence from sequence entry A00145 from bases 3 to 63 using

```
e.get_data('A00145', 'fasta', fasta_range=[3,63])
```

Retrieve assembled and annotated subsequences in HTML format (same as above but in HTML page).

```
e.view_data('A00145')
```

Retrieve expanded CON records:

To retrieve expanded CON records use the expanded=true parameter. For example, the expanded CON entry AL513382 in flat file format can be obtained as follows:

```
e.get_data('AL513382', frmt='text', expanded=True)
```

Expanded CON records are different from CON records in two ways. Firstly, the expanded CON records contain the full sequence in addition to the contig assembly instructions. Secondly, if a CON record contains only source or gap features the expanded CON records will also display all features from the segment records.

Retrieve assembled and annotated sequence header in flat file format

To retrieve assembled and annotated sequence header in flat file format please use the header=true parameter, e.g.:

```
e.get_data('BN000065', 'text', header=True)
```

Retrieve assembled and annotated sequence records using sequence versions:

```
e.get_data('AM407889.1', 'fasta')
e.get_data('AM407889.2', 'fasta')
```

Constructor

Parameters

verbose – set to False to prevent informative messages

data_warehouse()

```
get_data(identifier, frmt, fasta_range=None, expanded=None, header=None, download=None)
```

:param frmt

[xml, text, fasta, fastq, html, embl but does depend on the] entry

Example:

```
get_data("/AL513382", "embl")
```

ENA API changed in 2020 but we tried to keep the same services in this method.

```
get_taxon(taxon)
```

```
url = 'http://www.ebi.ac.uk/ena/browser/api'
```

EUtils

Interface to the EUtils web Service.

What is EUtils ?

URL

<http://www.ncbi.nlm.nih.gov/books/NBK25499/>

URL

http://www.ncbi.nlm.nih.gov/books/NBK25500/#chapter1.Demonstration_Programs

The Entrez Programming Utilities (E-utilities) are a set of eight server-side programs that provide a stable interface into the Entrez query and database system at the National Center for Biotechnology Information (NCBI). The E-utilities use a fixed URL syntax that translates a standard set of input parameters into the values necessary for various NCBI software components to search for and retrieve the requested data. The E-utilities are therefore the structured interface to the Entrez system, which currently includes 38 databases covering a variety of biomedical data, including nucleotide and protein sequences, gene records, three-dimensional molecular structures, and the biomedical literature.

—from <http://www.ncbi.nlm.nih.gov/books/NBK25497/>, March 2013

```
class EUtils(verbose=False, email='unknown', cache=False, xmlparser='EUtilsParser')
```

Interface to [NCBI Entrez Utilities](#) service

Note: Technical note: the WSDL interface was dropped in July 2015 so we now use the REST service.

Warning: Read the [guidelines](#) before sending requests. No more than 3 requests per seconds otherwise your IP may be banned. You should provide your email by filling the [email](#) so that before being banned, you may be contacted.

There are a few methods such as [ELink\(\)](#), [EFetch\(\)](#). Here is an example on how to use [EFetch\(\)](#) method to retrieve the FASTA sequence of a given identifier (34577063):

```
>>> from bioservices import EUtils
>>> s = EUtils()
>>> print(s.EFetch("protein", "34577063", rettype="fasta"))
```

(continues on next page)

(continued from previous page)

```
>gi|34577063|ref|NP_001117.2| adenylosuccinate synthetase isozyme 2 [Homo sapiens]
MAFAETYPAASSLPNGDCGRPRARPGGNRVTVVLGAQWDEGKGKVVDLLAQDADIVCRCQGGNNAGHTV
VVDSVEYDFHLLPSGIINPNVTAFIGNGVVIHLPLGFEEAEKNVQKGKLEGWEKRLIISDRAHIVDFH
QAADGIQEQQRQEQAGKNLGTTKKGIGVPYSSKAARSGLRMCDLVSDFDGFSERFKVLANQYKSIYPTLE
IDIEGELQKLKGYMEKIKPMVRDGVYFLYEALHGPPKILVEGANAALLDIDFGTYPFVTSSNCTVGGVC
TGLGMPQNVGEVYGVVKAYTTRVGIGAFPTQDNEIGELLQTRGREGVTTGRKRRCGWLDLVLLKYAH
MINGFTALALTKLDILDMFTEIKVGVAYKLDGEIIPHIPANQEVLNKVEVQYKTLPGWNTDISNARAFKE
LPVNAQNYVRFIEDELQIPVKWIGVGKSRESMIQLF
```

Most of the methods take a database name as input. You can obtain the valid list by checking the *databases* attribute.

A few functions takes Identifier(s) as input. It could be a list of strings, list of numbers, or a string where identifiers are separated either by comma or spaces.

A few functions take an argument called **term**. You can use the **AND** keyword with spaces or + signs as separators:

```
Correct: term=biomol mrna[properties] AND mouse[organism]
Correct: term=biomol+mrna[properties]+AND+mouse[organism]
```

Other special characters, such as quotation marks (") or the # symbol used in referring to a query key on the History server, could be represented by their URL encodings (%22 for "; %23 for #) or verbatim .:

```
Correct: term=#2+AND+"gene in genomic"[properties]
Correct: term=%232+AND+%22gene+in+genomic%22[properties]
```

For information about retmode and retype, please see:

http://www.ncbi.nlm.nih.gov/books/NBK25499/table/chapter4.T._valid_values_of__retmode_and/?report=objectonly

ECitMatch(bdata, **kargs)

Parameters

bdata – Citation strings. Each input citation must be represented by a citation string in the following format:

```
journal_title|year|volume|first_page|author_name|your_key|
```

Multiple citation strings may be provided by separating the strings with a carriage return character (%0D) or simply \r or \n.

The your_key value is an arbitrary label provided by the user that may serve as a local identifier for the citation, and it will be included in the output.

all spaces must be replaced by + symbols and that citation strings should end with a final vertical bar |.

Only xml supported at the time of this implementation.

```
from bioservices import EUtils
s = EUtils()
print(s.ECitMatch("proc+natl+acad+sci+u+s+a|1991|88|3248|mann+bj|Art1|
↪%0Dscience|1987|235|182|palmenberg+ac|Art2|"))
```


EFetch(db, id, retmode='text', **kargs)

Access to the EFetch E-Utilities

Parameters

- **db** (*str*) – database from which to retrieve UIDs.
- **id** (*str*) – list of identifiers.
- **retmode** – default to text (could be xml but not recommended).
- **rettype** – could be fasta, summary, docsum

Returns

depends on retmode parameter.

Note: addition to NCBI: settings rettype to “dict” returns a dictionary

```
>>> ret = s.EFetch("omim", "269840") --> ZAP70
>>> ret = s.EFetch("taxonomy", "9606", retmode="xml")
>>> [x.text for x in ret.getchildren()[0].getchildren() if x.tag==
↳ "ScientificName"]
['Homo sapiens']

>>> s = eutils.EUtils()
>>> s.EFetch("protein", "34577063", retmode="text", rettype="fasta")
>gi|34577063|ref|NP_001117.2| adenylosuccinate synthetase isozyme 2 [Homo
↳ sapiens]
MAFAETYPAASSLPNGDCGRPRARPGGNRVTTVLGAQWGDEGKGKVVDDLLAQDADIVCRCQGGNNAGHTV
VVDSEYDFHLLPSGIINPNVTAFINGVVIHLPLGFEEAEKNVQKGKLEGWEKRLIISDRAHIVDFH
QAADGIEQQRQEQAGKNLGTTKKGIGPVYSSKAARSLRMCDLVSDFDGFSERFKVLANQYKSIYPTLE
IDIEGELQKLKGYMEKIKPMVRDGVYFLYEALHGPPKKILVEGANAALLDIDFGTYPFVTSSNCTVGGVC
TGLGMPPQNVGEVYGVVKAYTTRVGIGAFPTQDNEIGELLQTRGREFGVTTGRKRRCGWLDLVLLKYAH
MINGFTALALTKLDILDMFTEIKVGVAYKLDGEIIPHIPANQEVNLKVEVQYKTLPGWNTDISNARAFKE
LPVNAQNYVRFIEDELQIPVKWIGVGKSRESMIQLF
```

Identifiers could be provided as a single string with comma-separated values, or a list of strings, a list of integers, or just one string or one integer but no mixing of types in the list:

```
>>> e.EFetch("protein", "352, 234", retmode="text", rettype="fasta")
>>> e.EFetch("protein", 352, retmode="text", rettype="fasta")
>>> e.EFetch("protein", [352], retmode="text", rettype="fasta")
>>> e.EFetch("protein", [352, 234], retmode="text", rettype="fasta")
```

retmode should be xml or text depending on the database. For instance, xml for pubmed:

```
>>> e.EFetch("pubmed", "20210808", retmode="xml")
>>> e.EFetch('nucleotide', id=15, retmode='xml')
>>> e.EFetch('nucleotide', id=15, retmode='text', rettype='fasta')
>>> e.EFetch('nucleotide', 'NT_019265', rettype='gb')
```

Other special characters, such as quotation marks (") or the # symbol used in referring to a query key on the History server, should be represented by their URL encodings (%22 for “; %23 for #).

A useful command is the following one that allows to get back a GI identifier from its accession, which is common to NCBI/EMBL:

```
e.EFetch(db="nucore",id="AP013055", rettype="seqid", retmode="text")
```

Changed in version 1.5.0: instead of “xml”, retmode can now be set to dict, in which case an XML is retrieved and converted to a dictionary if possible.

EGQuery(term, ***kargs*)

Provides the number of records retrieved in all Entrez databases by a text query.

Parameters

term (*str*) – Entrez text query. Spaces may be replaced by ‘+’ signs. For very long queries (more than several hundred characters long), consider using an HTTP POST call. See the PubMed or Entrez help for information about search field descriptions and tags. Search fields and tags are database specific.

Returns

returns a json data structure

```
>>> ret = s.EGQuery("asthma")
>>> [(x.DbName, x.Count) for x in ret.eGQueryResult.ResultItem if x.Count!='0']

>>> ret = s.EGQuery("asthma")
>>> ret.eGQueryResult.ResultItem[0]
{'Count': '115241',
 'DbName': 'pmc',
 'MenuName': 'PubMed Central',
 'Status': 'Ok'}
```

EInfo(db=None, ***kargs*)

Provides information about a database (e.g., number of records)

Parameters

db (*str*) – target database about which to gather statistics. Value must be a valid Entrez database name. See [databases](#) or don’t provide any value to obtain the entire list

Returns

a json data structure that depends on the value of [databases](#) (default to json)

```
>>> all_database_names = s.EInfo()
>>> # specific info about one database:
>>> ret = s.EInfo("taxonomy")
>>> ret[0]['count']
u'1445358'
>>> ret = s.EInfo('pubmed')
>>> ret[0]['fieldlist'][2]['fullname']
'Filter'
```

You can use the *retmode* parameter to ‘xml’ as well. In that case, you will need a XML parser.

```
>>> ret = s.EInfo("taxonomy")
```

Note: Note that the name in the XML or json outputs differ (some have lower cases, some have upper cases). This is inherent to the output of EUtils.

ELink(*db=None, dbfrom=None, id=None, **kargs*)

The Entrez links utility

Responds to a list of UIDs in a given database with either a list of related UIDs (and relevancy scores) in the same database or a list of linked UIDs in another Entrez database;

Parameters

- **db** (*str*) – valid database from which to retrieve UIDs.
- **dbfrom** (*str*) – Database containing the input UIDs. The value must be a valid database name (default = pubmed). This is the origin database of the link operation. If db and dbfrom are set to the same database value, then ELink will return computational neighbors within that database. Computational neighbors have linknames that begin with dbname_dbname (examples: protein_protein, pcassay_pcassay_activityneighbor).
- **id** (*str*) – UID list. Either a single UID or a comma-delimited list Limited to 200 Ids
- **cmd** (*str*) – ELink command mode. The command mode specified which function ELink will perform. Some optional parameters only function for certain values of cmd (see <http://www.ncbi.nlm.nih.gov/books/NBK25499/#chapter4.ELink>). Examples are neighbor, prlinks.

```
>>> # Example: Find related articles to PMID 20210808
>>> ret = s.ELink("pubmed", id="20210808", cmd="neighbor_score")

>>> ret = s.parse_xml(ret, 'EUtilsParser')
>>> ret.eLinkResult.LinkSet.LinkSetDb[0].Link[1]
{'Id': '16539535'}

>>> s.ELink(dbfrom="nucleotide", db="protein",
            id="48819,7140345")
>>> s.ELink(dbfrom="nucleotide", db="protein",
            id="48819,7140345")
>>> s.ELink(dbfrom='nuccore', id='21614549,219152114',
            cmd='ncheck')
```

Convert GI number to Taxon identifiers:

```
>>> s.ELink(dbfrom='nuccore', db="taxonomy", id='21614549,219152114')
```

EPost(*db, id, **kargs*)

Accepts a list of UIDs from a given database,

stores the set on the History Server, and responds with a query key and web environment for the uploaded dataset.

Parameters

- **db** (*str*) – a valid database
- **id** – list of strings of strings

Returns

a dictionary with a Web Environment string and a QueryKey to be re-used in another EUtils.

ESearch(*db, term, **kargs*)

Responds to a query in a given database

The response can be used later in ESummary, EFetch or ELink, along with the term translations of the query.

Parameters

- **db** – a valid database
- **term** – an Entrez text query

Note: see `_get_esearch_params()` for the list of valid parameters.

```
>>> ret = e.ESearch('protein', 'human', RetMax=5)
>>> ret = e.ESearch('taxonomy', 'Staphylococcus aureus[all names]')
>>> ret = e.ESearch('pubmed', "cokelaer AND BioServices")

>>> ret = e.ESearch('protein', '15718680')
>>> # Let us show the first pubmed identifier in a browser
>>> identifiers = e.pubmed(ret['idlist'][0])
```

More complex requests can be used. We will not cover all the possibilities (see the NCBI website). Here is an example to tune the search term to look into PubMed for the journal PNAS Volume 16, and retrieve.:

```
>>> e.ESearch("pubmed", "PNAS[ta] AND 16[vi]")
```

You can then look more closely at a specific identifier using EFetch:

```
>>> e = EFetch("pubmed")
>>> e.Efetch(identifiers)
```

Note: valid parameters can be found by calling `_get_esearch_params()`

ESpell(db, term, **kargs)

Retrieve spelling suggestions for a text query in a given database.

Parameters

- **db** (*str*) – database to search. Value must be a valid Entrez database name (default = pubmed).
- **term** (*str*) – Entrez text query. All special characters must be URL encoded.

```
>>> ret = e.ESpell(db="pubmed", term="aasthma+OR+alergy")
>>> ret = ret['eSpellResult']
>>> ret['Query']          'asthmaa OR allergies'
>>> ret['CorrectedQuery']
'asthma or allergy'
>>> ret = e.ESpell(db="pubmed", term="biosservices")
>>> ret = ret['eSpellResult']
>>> ret['CorrectedQuery']
biosservices
```

ESummary(db, id=None, **kargs)

Returns document summaries for a list of input UIDs

Parameters

- **db** – a valid database
- **id** (*str*) – list of identifiers (or string comma separated). all of the UUIDs must be from the database specified by db. Limited to 200 identifiers

```
>>> from bioservices import *
>>> s = EUtils()
>>> ret = s.ESummary("snp", "7535")
>>> ret = s.ESummary("snp", "7535,7530")
>>> ret = s.ESummary("taxonomy", "9606,9913")
```

```
>>> proteins = e.ESearch("protein", "bacteriorhodopsin",
                        retmax=20)
>>> ret = e.ESummary("protein", 449301857)
>>> ret['result']['449301857']['extra']
'gi|449301857|gb|EMC97866.1|gnl|WGS:AEIF|BAUCODRAFT_31870'
```

property databases

Returns list of valid databases

email

fill this with your email address

help()

Open EUtils help page

parse_xml(*ret*, *method=None*)

snp_summary(*id*)

Alias to Efetch for the SNP database

Return

a json data structure.

```
>>> ret = s.snp("123")
```

taxonomy_summary(*id*)

Alias to EFetch for the taxonomy database

```
>>> s = EUtils()
>>> ret = s.taxonomy("9606")
>>> ret['9606']['species']
'sapiens'
>>> ret = s.taxonomy("9606,9605,111111111,9604")
>>> ret['9604']['taxid']
9604
```

class EUtilsParser(*xml*)

Convert xml returned by EUtils into a structure easier to manipulate

Used by *EUtils.EGQuery()*, *EUtils.ELink()*.

GeneProf

Currently removed from the main API from version 1.6.0 onwards. You can still get the code in earlier version or in the github repository in the attic/ directory

QuickGO

Interface to the quickGO interface

What is quickGO

URL

<http://www.ebi.ac.uk/QuickGO/>

Service

<http://www.ebi.ac.uk/QuickGO/WebServices.html>

“QuickGO is a fast web-based browser for Gene Ontology terms and annotations, which is provided by the UniProt-GOA project at the EBI. “

—from QuickGO home page, Dec 2012

```
class QuickGO(verbose=False, cache=False)
```

Interface to the QuickGO service

Retrieve information given a GO identifier:

```
>>> from bioservices import QuickGO
>>> go = QuickGO()
>>> res = go.get_go_terms("GO:0003824")
```

Changed in version we: use the new QuickGO API since version 1.5.0 To use the old API, please use version of bioservices below 1.5

Constructor

Parameters

verbose (*bool*) – print informative messages.

Annotation(*assignedBy=None, includeFields=None, limit=100, page=1, aspect=None, reference=None, geneProductId=None, evidenceCode=None, goId=None, qualifier=None, withFrom=None, taxonId=None, taxonUsage=None, goUsage=None, goUsageRelationships=None, evidenceCodeUsage=None, evidenceCodeUsageRelationships=None, geneProductType=None, targetSet=None, geneProductSubset=None, extension=None*)

Calling the Annotation service

Changed in version 1.4.18: due to service API changes, we refactored this method completely

Parameters

- **assignedBy** (*str*) – The database from which this annotation originates. Accepts comma separated values.E.g., BHF-UCL,Ensembl.
- **includeFields** (*str*) – Optional fields retrieved from external services. Accepts comma separated values. accepted values: goName, taxonName, name, synonyms.

- **limit** (*int*) – download limit (number of lines) (default 10,000 rows, which may not be sufficient for the data set that you are downloading. To bypass this default, and return the entire data set, specify a limit of -1).
- **page** (*int*) – results may be stored on several pages. You must provide this number. There is no way to retrieve more than 100 results without calling this function several times changing this parameter (default to 1).
- **aspect** (*char*) – use this to limit the annotations returned to a specific ontology or ontologies (Molecular Function, Biological Process or Cellular Component). The valid character can be F,P,C.
- **reference** (*str*) – PubMed or GO reference supporting annotation. Can refer to a specific reference identifier or category (for category level, use * after ref type). Can be 'PUBMED:*', 'GO_REF:0000002'.
- **geneProductId** (*str*) – The id of the gene product annotated with the GO term. Accepts comma separated values. E.g., URS00000064B1_559292.
- **evidenceCode** (*str*) – Evidence code indicating how the annotation is supported. Accepts comma separated values. E.g., ECO:0000255.
- **goId** (*str*) – The GO id of an annotation. Accepts comma separated values. E.g., GO:0070125.
- **qualifier** (*str*) – Aids the interpretation of an annotation. Accepts comma separated values. E.g., enables,involved_in.
- **withFrom** (*str*) – Additional ids for an annotation. Accepts comma separated values. E.g., P63328.
- **taxonId** (*str*) – The taxonomic id of the species encoding the gene product associated to an annotation. Accepts comma separated values. E.g., 1310605.
- **taxonUsage** (*str*) – Indicates how the taxonomic ids within the annotations should be used. E.g., exact.
- **goUsage** (*str*) – Indicates how the GO terms within the annotations should be used. Used in conjunction with 'goUsageRelationships' filter. E.g., descendants.
- **goUsageRelationships** (*str*) – The relationship between the 'goId' values found within the annotations. Allows comma separated values. E.g., is_a,part_of.
- **evidenceCodeUsage** (*str*) – Indicates how the evidence code terms within the annotations should be used. Is used in conjunction with 'evidenceCodeUsageRelationships' filter. E.g., descendants, exact<F12>
- **evidenceCodeUsageRelationships** (*str*) – The relationship between the provided 'evidenceCode' identifiers. Allows comma separated values. E.g., is_a,part_of.
- **geneProductType** (*str*) – The type of gene product. Accepts comma separated values. E.g., protein,RNA. can be protein, RNA and/or complex
- **targetSet** (*str*) – Gene product set. Accepts comma separated values. E.g., KRUK,BHF-UCL,Exosome.
- **geneProductSubset** (*str*) – A database that provides a set of gene products. Accepts comma separated values. E.g., TrEMBL.
- **extension** (*str*) – Extensions to annotations, where each extension can be: EXTENSION(DB:ID) / EXTENSION(DB) / EXTENSION.

Returns

a dictionary

```
>>> print(s.Annotation(protein='P12345', frmt='tsv', col="ref,evidence",
... reference='PMID:*'))
>>> print(s.Annotation(protein='P12345,Q4VCS5', frmt='tsv',
... col="ref,evidence",reference='PMID:,Reactome:'))
```

Annotation_from_gold(*gold*, *max_number_of_pages*=25, ***kargs*)

Returns a DataFrame containing annotation on a given GO identifier

Parameters

protein (*str*) – a GO identifier

Returns

all outputs are stored into a Pandas.DataFrame data structure.

All parameters from *Annotation* are also valid except **format** that is set to **tsv** and cols that is made of all possible column names.

gene_product_search(*query*, *taxonID*=None, *page*=1, *limit*=100, *type*=None, *dbSubSet*=None, *proteome*=None)

get_go_ancestors(*query*, *relations*='is_a,part_of,occurs_in,regulates')

get_go_chart(*query*)

```
res = go.get_chart("GO:0022804")
with open("temp.png", "wb") as fout:
    fout.write(res)
```

get_go_children(*query*)

get_go_paths(*_from*, *_to*, *relations*='is_a,part_of,occurs_in,regulates')

Retrieves the paths between two specified sets of ontology terms. Each path is formed from a list of (term, relationship, term) triples.

```
paths = go.go_terms_path("GO:0005215", "GO:0003674") # First path is found as the first item
in the "results" paths["results"][0]
```

get_go_terms(*query*, *max_number_of_pages*=None)

Get information on all terms and page through the result

go_search(*query*, *limit*=600, *page*=1)

Searches a simple user query, e.g., query=apopto

Parameters

- **query** (*str*) –
- **limit** (*int*) – max 600
- **page** (*int*) –

Kegg

This module provides a class [KEGG](#) to access to the REST KEGG interface. There are additional methods and functionalities added by **BioServices**.

Note: a previous interface to the KEGG WSDL service was designed but the WSDL closed in Dec 2012.

What is KEGG ?

URL

<http://www.kegg.jp/>

REST

<http://www.kegg.jp/kegg/rest/keggapi.html>

weblink

<http://www.genome.jp/kegg/rest/weblink.html>

dbentries

<http://www.genome.jp/kegg/rest/dbentry.html>

“KEGG is a database resource for understanding high-level functions and utilities of the biological system, such as the cell, the organism and the ecosystem, from molecular-level information, especially large-scale molecular datasets generated by genome sequencing and other high-throughput experimental technologies (See Release notes for new and updated features). “

—KEGG home page, Jan 2013

Some terminology

The following list is a simplified list of terminology taken from KEGG API pages.

- organisms (**org**) are made of a three-letter (or four-letter) code (e.g., **hsa** stands for Human Sapiens) used in KEGG (see [organismIds](#)).
- **db** is a database name. See [databases](#) attribute and *KEGG Databases Names and Abbreviations* section.
- **entry_id** is a unique identifier. It is a combination of the database name and the identifier of an entry joined by a colon sign (e.g. ‘embl:J00231’).

Here are some examples of entry Ids:

- **genes_id**: A KEGG organism and a gene name (e.g. ‘eco:b0001’).
- **enzyme_id**: ‘ec’ and an enzyme code. (e.g. ‘ec:1.1.1.1’). See [enzymeIds](#).
- **compound_id**: ‘cpd’ and a compound number (e.g. ‘cpd:C00158’). Some compounds also have ‘glycan_id’ and both IDs are accepted and converted internally. See [compoundIds](#).
- **drug_id**: ‘dr’ and a drug number (e.g. ‘dr:D00201’). See [drugIds](#).
- **glycan_id**: ‘gl’ and a glycan number (e.g. ‘gl:G00050’). Some glycans also have ‘compound_id’ and both IDs are accepted and converted internally. see [glycanIds](#) attribute.
- **reaction_id**: ‘rn’ and a reaction number (e.g.

- ‘rn:R00959’ is a reaction which catalyze cpd:C00103 into cpd:C00668). See [reactionIds](#) attribute.
- **pathway_id**: ‘path’ and a pathway number. Pathway numbers prefixed by ‘map’ specify the reference pathway and pathways prefixed by a KEGG organism specify pathways specific to the organism (e.g. ‘path:map00020’, ‘path:eco00020’). See [pathwayIds](#) attribute.
- **motif_id**: a motif database names (‘ps’ for prosite, ‘bl’ for blocks, ‘pr’ for prints, ‘pd’ for prodom, and ‘pf’ for pfam) and a motif entry name. (e.g. ‘pf:DnaJ’ means a Pfam database entry ‘DnaJ’).
- **ko_id**: identifier made of ‘ko’ and a ko number (e.g. ‘ko:K02598’). See [koIds](#) attribute.

KEGG Databases Names and Abbreviations

Here is a list of databases used in KEGG API with their name and abbreviation:

| Database Name | Abbrev | kid |
|---------------|--------|------------|
| pathway | path | map number |
| brite | br | br number |
| module | md | M number |
| disease | ds | H number |
| drug | dr | D number |
| environ | ev | E number |
| orthology | ko | K number |
| genome | genome | T number |
| genomes | gn | T number |
| genes | • | • |
| ligand | ligand | • |
| compound | cpd | C number |
| glycan | gl | G number |
| reaction | rn | R number |
| rpair | rp | RP number |
| rclass | rc | RC number |
| enzyme | ec | • |

Database Entries

Database entries can be written in on of the following ways:

```
<dbentries> = <dbentry>1[+<dbentry>2...]  
<dbentry> = <db:entry> | <kid> | <org:gene>
```

Each database entry is identified by:

```
db:entry
```

where “db” is the database name or its abbreviation shown above and “entry” is the entry name or the accession number that is uniquely assigned within the database. In reality “db” may be omitted, for the entry name called the KEGG object identifier (kid) is unique across KEGG.:

```
kid = database-dependent prefix + five-digit number
```

In the KEGG GENES database the db:entry combination must be specified. This is more specifically written as:

```
org:gene
```

where “org” is the three- or four-letter KEGG organism code or the T number genome identifier and “gene” is the gene identifier, usually locus_tag or ncbi GeneID, or the primary gene name.

class `KEGG(verbose=False, cache=False)`

Interface to the [KEGG](#) service

This class provides an interface to the KEGG REST API. The weblink tools are partially accesible. All dbentries can be parsed into dictionaries using the [KEGGParser](#)

Here are some examples. In order to retrieve the entry of the gene identifier 7535 of the **hsa** organism, type:

```
from bioservices import KEGG
s = KEGG()
print(s.get("hsa:7535"))
```

The output is the raw ouput sent by KEGG API. See [KEGGParser](#) to parse this output.

See also:

The [Database Entries](#) to know more about the db entries format.

Another example here below shows how to print the list of pathways of the human organism:

```
print(s.list("pathway", organism="hsa"))
```

Further post processing would allow you to retrieve the pathway Ids. However, we provide additional functions to the KEGG API so the previous code and post processing to extract the pathway Ids can be written as:

```
s.organism = "hsa"
s.pathwayIds
```

and similarly you can get all [databases\(\)](#) output and database Ids easily. For example, for the reaction database:

```
s.reaction # equivalent to s.list("reaction")
s.reactionIds
```

Other methods of interest are [conv\(\)](#), [find\(\)](#), [get\(\)](#).

See also:

[KEGG Databases Names and Abbreviations](#), [Database Entries](#), [Some terminology](#).

Constructor

Parameters

verbose (*bool*) – prints informative messages

Tnumber2code(*Tnumber*)

Converts organism T number to its code

```
>>> from bioservices import KEGG
>>> s = KEGG()
>>> s.Tnumber2code("T01001")
'hsa'
```

property briteIds

returns list of brite Ids.

See also:

[*list\(\)*](#)

code2Tnumber(*code*)

Converts organism code to its T number

```
>>> from bioservices import KEGG
>>> s = KEGG()
>>> s.code2Tnumber("hsa")
'T01001'
```

property compoundIds

returns list of compound Ids

See also:

[*list\(\)*](#)

conv(*target*, *source*)

convert KEGG identifiers to/from outside identifiers

Parameters

- **target** (*str*) – the target database (e.g., a KEGG organism).
- **source** (*str*) – the source database (e.g., uniprot) or a valid dbentries; see below for details.

Returns

a dictionary with keys being the source and values being the target.

Here are the rules to set the target and source parameters.

If the second argument is not a **dbentries**, source and target parameters can be of two types:

1. gene identifiers. If the target is a KEGG Id, then the source must be one of *ncbi-gi*, *ncbi-geneid* or *uniprot*.

Note: source and target can be swapped.

- chemical substance identifiers. If the target is one of the following kegg database: drug, compound, glycan then the source must be one of *pubchem* or *chebi*.

Note: again, source and target can be swapped

If the second argument is a **dbentries**, it can be again of two types:

- gene identifiers. The database used can be one ncbi-gi, ncbi-geneid, uniprot or any KEGG organism
- chemical substance identifiers. The database used can be one of drug, compound, glycan, pubchem or chebi only.

Note: if the second argument is a dbentries, target and dbentries cannot be swapped.

```
# conversion from NCBI GeneID to KEGG ID for E. coli genes
conv("eco", "ncbi-geneid")
# inverse of the above example
conv("eco", "ncbi-geneid")
#conversion from KEGG ID to NCBI GI
conv("ncbi-gi", "hsa:10458+ece:Z5100")
```

To make it clear by taking another example, you can either convert an entire database to another (e.g., from uniprot to KEGG Id all human gene IDs):

```
uniprot_ids, kegg_ids = s.conv("hsa", "uniprot")
```

or a subset by providing a valid **dbentries**:

```
s.conv("hsa", "up:Q9BV86+")
```

Warning: dbentries are not check and are supposed to be correct. See `check_idbentries()` to help you checking a dbentries.

Warning: call to this function may be long. `conv("hsa", "uniprot")` takes a minute supprisingly, `conv("uniprot", "hsa")` takes just a few seconds.

Changed in version 1.1: the output is now a dictionary, not a list of tuples

property databases

Returns list of valid KEGG databases.

dbinfo(*database='kegg'*)

Displays the current statistics of a given database

Parameters

database (*str*) – can be one of: kegg (default), brite, module, disease, drug, environ, ko, genome, compound, glycan, reaction, rpair, rclass, enzyme, genomes, genes, ligand or any *organismIds*.

```
from bioservices import KEGG
s = KEGG()
s.dbinfo("hsa") # human organism
s.dbinfo("T01001") # same as above
s.dbinfo("pathway")
```

Changed in version 1.4.1: renamed info method into `dbinfo()`, which clashes with Logging framework `info()` method.

property `drugIds`

returns list of drug Ids

See also:

`list()`

`entry(dbentries)`

Retrieve entry

There is a weblink service (see <http://www.genome.jp/kegg/rest/weblink.html>) Since it is equivalent to `get()`, we do not implement it for now

property `enzymeIds`

returns list of enzyme Ids

See also:

`list()`

`find(database, query, option=None)`

finds entries with matching query keywords or other query data in a given database

Parameters

- **database** (*str*) – can be one of pathway, module, disease, drug, environ, ko, genome, compound, glycan, reaction, rpair, rclass, enzyme, genes, ligand or an organism code (see `organismIds` attributes) or T number (see `organismTnumbers` attribute).
- **query** (*str*) – See examples
- **option** (*str*) – If option provided, database can be only ‘compound’ or ‘drug’. Option can be ‘formula’, ‘exact_mass’ or ‘mol_weight’

Note: Keyword search against brite is not supported. Use `/list/brite` to retrieve a short list.

```
# search for pathways that contain Viral in the definition
s.find("pathway", "Viral")
# for keywords "shiga" and "toxin"
s.find("genes", "shiga+toxin")
# for keywords "shiga toxin"
s.find("genes", "shiga toxin")
# for chemical formula "C7H10O5"
s.find("compound", "C7H10O5", "formula")
# for chemical formula containing "O5" and "C7"
s.find("compound", "O5C7", "formula")
# for 174.045 <= exact mass < 174.055
s.find("compound", "174.05", "exact_mass")
```

(continues on next page)

(continued from previous page)

```
# for 300 =< molecular weight =< 310
s.find("compound", "300-310", "mol_weight")
```

get(dbentries, option=None, parse=False)

Retrieves given database entries

param str dbentries

KEGG database entries involving the following database: pathway, brite, module, disease, drug, environ, ko, genome compound, glycan, reaction, rpair, rclass, enzyme or any organism using the KEGG organism code (see [organismIds](#) attributes) or T number (see [organismTnumbers](#) attribute).

param str option

one of: aaseq, ntseq, mol, kcf, image, kgml

Note:

you can add the option at the end of dbentries in which case
the parameter option must not be used (see example)

```
from bioservices import KEGG
s = KEGG()
# retrieves a compound entry and a glycan entry
s.get("cpd:C01290+gl:G000092")
# same as above
s.get("C01290+G000092")
# retrieves a human gene entry and an E.coli 0157 gene entry
s.get("hsa:10458+ece:Z5100")
# retrieves amino acid sequences of a human gene and an E.coli 0157 gene
s.get("hsa:10458+ece:Z5100/aaseq")
# retrieves the image file of a pathway map
s.get("hsa05130/image")
# same as above
s.get("hsa05130", "image")
```

Another example here below shows how to save the image of a given pathway:

```
res = s.get("hsa05130/image")
# same as : res = s.get("hsa05130", "image")
f = open("test.png", "w")
f.write(res)
f.close()
```

Note: The input is limited up to 10 entries (KEGG restriction).

get_pathway_by_gene(gene, organism)

Search for pathways that contain a specific gene

Parameters

- **gene** (*str*) – a valid gene Id
- **organism** (*str*) – a valid organism (e.g., hsa)

Returns

list of pathway Ids that contain the gene

```
>>> s.get_pathway_by_gene("7535", "hsa")
['path:hsa04064', 'path:hsa04650', 'path:hsa04660', 'path:hsa05340']
```

property glycanIds

Returns list of glycan Ids

See also:

[list\(\)](#)

isOrganism(org)

Checks if org is a KEGG organism

Parameters

org (*str*) –

Returns

True if org is in the KEGG organism list (code or Tnumber)

```
>>> from bioservices import KEGG
>>> s = KEGG()
>>> s.isOrganism("hsa")
True
```

property koIds

returns list of ko Ids

See also:

[list\(\)](#)

link(target, source)

Find related entries by using database cross-references

Parameters

- **target** (*str*) – the target KEGG database or organism (see below for the list).
- **source** (*str*) – the source KEGG database or organism (see below for the list) or a valid dbentries involving one of the database; see below for details.

The valid list of databases is pathway, brite, module, disease, drug, environ, ko, genome, compound, glycan, reaction, rpair, rclass, enzyme

```
# KEGG pathways linked from each of the human genes
s.link("pathway", "hsa")
# human genes linked from each of the KEGG pathways
s.link("hsa", "pathway")
# KEGG pathways linked from a human gene and an E. coli 0157 gene.
s.link("pathway", "hsa:10458+ece:Z5100")
```

list(query, organism=None)

Returns a list of entry identifiers and associated definition for a given database or a given set of database entries

Parameters

- **query** (*str*) – can be one of pathway, brite, module, disease, drug, environ, ko, genome, compound, glycan, reaction, rpair, rclass, enzyme, organism **or** an organism from the *organismIds* attribute **or** a valid dbentry (see below). If a dbentry query is provided, organism should not be used!
- **organism** (*str*) – a valid organism identifier that can be provided. If so, database can be only “pathway” or “module”. If not provided, the default value is chosen (*organism*)

Returns

A string with a structure that depends on the query

Here is an example that shows how to extract the pathways IDs related to the hsa organism:

```
>>> s = KEGG()
>>> res = s.list("pathway", organism="hsa")
>>> pathways = [x.split()[0] for x in res.strip().split("\n")]
>>> len(pathways) # as of Dec 2012
261
```

Note, however, that there are convenient aliases to some of the databases. For instance, the pathway IDs can also be retrieved as a list from the *pathwayIds* attribute (after defining the *organism* attribute).

Note: If you set the query to a valid organism, then the second argument rganism is irrelevant and ignored.

Note: If the query is not a database or an organism, it is supposed to be a valid dbentries string and the maximum number of entries is 100.

Other examples:

```
s.list("pathway")           # returns the list of reference pathways
s.list("pathway", "hsa")    # returns the list of human pathways
s.list("organism")          # returns the list of KEGG organisms with
↳ taxonomic classification
s.list("hsa")               # returns the entire list of human genes
s.list("T01001")           # same as above
s.list("hsa:10458+ece:Z5100") # returns the list of a human gene and an E.coli
↳ 0157 gene
s.list("cpd:C01290+gl:G00092") # returns the list of a compound entry and a
↳ glycan entry
s.list("C01290+G00092")     # same as above
```

lookfor_organism(query)

Look for a specific organism

Parameters

query (*str*) – your search term. upper and lower cases are ignored

Returns

a list of definition that matches the query

lookfor_pathway(query)

Look for a specific pathway

Parameters

query (*str*) – your search term. upper and lower cases are ignored

Returns

a list of definition that matches the query

property moduleIds

returns list of module Ids for the default organism.

organism must be set.

```
s = KEGG()
s.organism = "hsa"
s.moduleIds
```

property organism

returns the current default organism

property organismIds

Returns list of organism Ids

property organismTnumbers

returns list of organisms (T numbers)

See also:

list()

parse(entry)

See *KEGGParser* for details

Parse entry returned by *get()*

```
k = KEGG()
res = k.get("hsa04150")
d = k.parse(res)
```

parse_kgml_pathway(pathwayId, res=None)

Parse the pathway in KGML format and returns a dictionary (relations and entries)

Parameters

- **pathwayId** (*str*) – a valid pathwayId e.g. hsa04660
- **res** (*str*) – if you already have the output of the query *get(pathwayId)*, you can provide it, otherwise it is queried.

Returns

a dictionary with relations and entries as keys. Values of relations is a list of relations, each relation being dictionary with entry1, entry2, link, value, name. The list of entries is a list of dictionary as well. Entry contains contains more details about the entry found in the relation. See example below for details.

```
>>> res = s.parse_kgml_pathway("hsa04660")
>>> set([x['name'] for x in res['relations']])
>>> res['relations'][-1]
{'entry1': u'15',
 'entry2': u'13',
 'link': u'PPrel',
 'name': u'phosphorylation',
 'value': u'+p'}
```

(continues on next page)

(continued from previous page)

```
>>> set([x['link'] for x in res['relations']])
set([u'PPrel', u'PCrel'])

>>> # get information about an entry :
>>> res['entries'][4]
```

See also:

KEGG API

pathway2sif(*pathwayId*, *uniprot=True*)

Extract protein-protein interaction from KEGG pathway to a SIF format

Warning: experimental Not tested on all pathway. should be move to another package such as cellnopt

Parameters

- **pathwayId** (*str*) – a valid pathway Id
- **uniprot** (*bool*) – convert to uniprot Id or not (default is True)

Returns

a list of relations (A 1 B) for activation and (A -1 B) for inhibitions

This is longish due to the conversion from KEGGIds to UniProt.

This method can be useful to provide prior knowledge network to software such as CellNOpt (see <http://www.cellnopt.org>)

property pathwayIds

returns list of pathway Ids for the default organism.

organism must be set.

```
s = KEGG()
s.organism = "hsa"
s.pathwayIds
```

property reactionIds

returns list of reaction Ids

save_pathway(*pathId*, *filename*, *scale=None*, *keggid={}*, *params={}*)

Save KEGG pathway in PNG format

Parameters

- **pathId** – a valid pathway identifier
- **filename** (*str*) – output PNG file
- **params** – valid kegg params expected

show_entry(*entry*)

Opens URL corresponding to a valid entry

```
s.www_bget("path:hsa05416")
```

show_module(modId)

Show a given module inside a web browser

Parameters

modId (*str*) – a valid module Id. See [moduleIds\(\)](#)

Validity of modId is not checked but if wrong the URL will not open a proper web page.

show_pathway(pathId, scale=None, dcolor='pink', keggid={}, show=True)

Show a given pathway inside a web browser

Parameters

- **pathId** (*str*) – a valid pathway Id. See [pathwayIds\(\)](#)
- **scale** (*int*) – you can scale the image with a value between 0 and 100
- **dcolor** (*str*) – set the default background color of nodes
- **keggid** (*dict*) – set color of entries contained in the pathway as key/value pairs; can also be a list, in which case all nodes have the same default color (red)

Note: if scale is provided, dcolor and keggid are ignored.

```
# show a pathway in the browser
s.show_pathway("path:hsa05416", scale=50)

# Same as above but also highlights some KEGG Ids (red for all)
s.show_pathway("path:hsa05416", dcolor="white",
               keggid=['1525', '1604', '2534'])

# You can refine the colors using a dictionary:
s.show_pathway("path:hsa05416", dcolor="white",
               keggid={'1525':'yellow,red', '1604':'blue,green', '2534':"blue"})
```

class KEGGParser(verbose=False)

This is an extension of the [KEGG](#) class to ease parsing of dbentries

This class provides a generic method [parse\(\)](#) that will read the output of a dbentry returned by [KEGG.get\(\)](#) and converts it into a dictionary ready to use.

The [parse\(\)](#) method parses any entry. It can be a pathway, a gene, a compound...

```
from bioservices import *
s = KEGG()

# Retrieve a KEGG entry
res = s.get("hsa04150")

# parse it
d = s.parse(res)
```

As a pedagogical example, you can then further process this dictionary. Here below, we convert the gene Ids found in the pathway into UniProt Ids:

```
# Get the KEGG Ids in the pathway
kegg_geneIds = [x for x in d['GENE']]

# Convert them
db_up, db_kegg = s.conv("hsa", "uniprot")

# Get the corresponding uniprot Ids
indices = [db_kegg.index("hsa:%s" % x) for x in kegg_geneIds]
uniprot_geneIds = [db_up[x] for x in indices]
```

However, you could also have done it simply as follows:

```
kegg_geneIds = [x for x in d['gene']]
uprot_geneIds = [s.parse(s.get("hsa:"+str(e))['DBLINKS']['UniProt:']) for e in d[
    ↪ 'GENE']]
```

Note: The 2 outputs are slightly different.

See also:

<http://www.kegg.jp/kegg/rest/dbentry.html>

parse(res)

Parse to any outputs returned by *KEGG.get()*

Parameters

res (*str*) – output of a *KEGG.get()*.

Returns

a dictionary. Keys are those found in the KEGG entry (e.g., REACTION, ENTRY, EQUATION, ...). The format of each value is various. It could be a string, a list (of strings generally), a dictionary, a float depending on the key. Depending on the type of the entry (e.g., module, pathway), the type of the value may also differ (e.g., REACTION can be either a list of reactions or a dictionary depending on the content)

```
>>> # Parses a drug entry
>>> res = s.get("dr:D00001")
>>> # Parses a pathway entry
>>> res = s.get("path:hsa10584")
>>> # Parses a module entry
>>> res = s.get("md:hsa_M00554")
>>> # Parses a disease entry
>>> res = s.get("ds:H00001")
>>> # Parses a environ entry
>>> res = s.get("ev:E00001")
>>> # Parses Orthology entry
>>> res = s.get("ko:K00001")
>>> # Parses a Genome entry
>>> res = s.get('genome:T00001')
>>> # Parses a gene entry
>>> res = s.get("hsa:1525")
>>> # Parses a compound entry
>>> s.get("cpd:C00001")
```

(continues on next page)

(continued from previous page)

```

>>> # Parses a glycan entry
>>> res = s.get("gl:G000001")
>>> # Parses a reaction entry
>>> res = s.get("rn:R000001")
>>> # Parses a rpair entry
>>> res = s.get("rp:RP000001")
>>> # Parses a rclass entry
>>> res = s.get("rc:RC000001")
>>> # Parses an enzyme entry
>>> res = s.get('ec:1.1.1.1')

>>> d = s.parse(res)

```

HGNC

Interface to HUGO/HGNC web services

What is HGNC ?

URL

<http://www.genenames.org>

Citation

“The HUGO Gene Nomenclature Committee (HGNC) has assigned unique gene symbols and names to over 37,000 human loci, of which around 19,000 are protein coding. [genenames.org](http://www.genenames.org) is a curated online repository of HGNC-approved gene nomenclature and associated resources including links to genomic, proteomic and phenotypic information, as well as dedicated gene family pages.”

—From HGNC web site, July 2013

class HGNC(verbose=False, cache=False)

Wrapper to the [genenames](http://www.genenames.org) web service

See details at <http://www.genenames.org/help/rest-web-service-help>

fetch(database, query, frmt='json')

Retrieve particular records from a searchable fields

Returned object is a json object with fields as in `stored_field`, which is returned from `get_info()` method.

Only one query at a time. No wild cards are accepted.

```

>>> h = HGNC()
>>> h.fetch('symbol', 'ZNF3')
>>> h.fetch('alias_name', 'A-kinase anchor protein, 350kDa')

```

get_info(frmt='json')

Request information about the service

Fields are when the server was last updated (`lastModified`), the number of documents (`numDoc`), which fields can be queried using search and fetch (`searchableFields`) and which fields may be returned by fetch (`storedFields`).

search(database_or_query=None, query=None, frmt='json')

Search a searchable field (database) for a pattern

The search request is more powerful than fetch for querying the database, but search will only return the fields hgnc_id, symbol and score. This is because this tool is mainly intended to query the server to find possible entries of interest or to check data (such as your own symbols) rather than to fetch information about the genes. If you want to retrieve all the data for a set of genes from the search result, the user could use the hgnc_id returned by search to then fire off a fetch request by hgnc_id.

Parameters

database – if not provided, search all databases.

```
# Search all searchable fields for the term BRAF
h.search('BRAF')

# Return all records that have symbols that start with ZNF
h.search('symbol', 'ZNF*')

# Return all records that have symbols that start with ZNF
# followed by one and only one character (e.g. ZNF3)
# Nov 2015 does not work neither here nor in within in the
# official documentation
h.search('symbol', 'ZNF?')

# search for symbols starting with ZNF that have been approved
# by HGNC
h.search('symbol', 'ZNF*+AND+status:Approved')

# return ZNF3 and ZNF12
h.search('symbol', 'ZNF3+OR+ZNF12')

# Return all records that have symbols that start with ZNF which
# are not approved (ie entry withdrawn)
h.search('symbol', 'ZNF*+NOT+status:Approved')
```

class HGNCDeprecated(verbose=False, cache=False)

Interface to the HGNC service

```
>>> from bioservices import *
>>> # Fetch XML document for gene ZAP70
>>> s = HGNC()
>>> xml = s.get_xml("ZAP70")
>>> # You can fetch several gene names:
>>> xml = s.get_xml("ZAP70;INSR")
>>> # Wrong gene name request returns an empty list
>>> s.get_xml("wrong")
[]
```

For a single name, the following methods are available:

```
>>> # get the aliases of a given gene
>>> print(s.get_aliases("ZAP70"))
[u'ZAP-70', u'STD']
>>> # get UniProt accession code
```

(continues on next page)

(continued from previous page)

```
>>> s.get_xrefs("ZAP70")['UniProt']['xkey']
'P43403'
>>> # get XML link to a UniProt cross-reference
>>> s.get_xrefs("ZAP70", "xml")['UniProt']['link']
['http://www.uniprot.org/uniprot/P43403.xml']
```

You can access to the links of a cross reference as well:

```
values = s.get_xrefs("ZAP70")
s.on_web(values['EntrezGene']['link'][0])
```

References

<http://www.avatar.se/HGNC/doc/tutorial.html>

Warning: this maybe not the official.

Constructor

Parameters

- **name** (*str*) – a name for this service
- **url** (*str*) – its URL
- **verbose** (*bool*) – prints informative messages if True (default is True)
- **requests_per_sec** – maximum number of requests per seconds are restricted to 3. You can change that value. If you reach the limit, an error is raise. The reason for this limitation is that some services (e.g., NCBI) may black list you IP. If you need or can do more (e.g., ChEMBL does not seem to have restrictions), change the value. You can also have several instance but again, if you send too many requests at the same, your future requests may be retracted. Currently implemented for REST only

All instances have an attribute called `logging` that is an instance of the `logging` module. It can be used to print information, warning, error messages:

```
self.logging.info("informative message")
self.logging.warning("warning message")
self.logging.error("error message")
```

The attribute `debugLevel` can be used to set the behaviour of the logging messages. If the argument `verbose` is True, the `debugLevel` is set to INFO. If `verbose` is False, the `debugLevel` is set to WARNING. However, you can use the `debugLevel` attribute to change it to one of DEBUG, INFO, WARNING, ERROR, CRITICAL. `debugLevel=WARNING` means that only WARNING, ERROR and CRITICAL messages are shown.

`get_aliases(gene)`

Get aliases for a single gene name

`get_all_names()`

Returns all gene names

get_chromosome(*gene*)

Get chromosome for a single gene name

get_name(*gene*)

Get name for a single gene name

get_previous_names(*gene*)

Get previous names for a single gene name

get_previous_symbols(*gene*)

Get previous symbols for a single gene name

get_withdrawn_symbols(*gene*)

Get withdrawn symbols for a single gene name

get_xml(*gene*)

Returns XML of a single gene or list of genes

Parameters

gene (*str*) – a valid gene name. Several gene names can be concatenated with comma ; character (e.g., 'ZAP70;INSR')

```
>>> from bioservices import *
>>> s = HGNC()
>>> res = s.get_xml("ZAP70")
>>> res.findAll("alias")
>>> [x.text for x in res.findAll("alias")]
[u'ZAP-70', u'STD']
```

See also:

[`get_aliases\(\)`](#)

get_xrefs(*gene*, *keep*='html')

Get the cross references for a given single gene name

```
>>> databases = s.get_xrefs("ZAP70").keys()

>>> # get XML link to a UniProt cross-reference
>>> s.get_xrefs("ZAP70", "xml")['UniProt']['link']
['http://www.uniprot.org/uniprot/P43403.xml']
```

lookfor(*pattern*)

Finds all genes that starts with a given pattern

Parameters

pattern (*str*) – a string. Could be the wild character *

Returns

list of dictionary. Each dictionary contains the 'acc', 'xlink:href' and 'xlink:title' keys

```
>>> from bioservices import *
>>> s = HGNC()
>>> s.lookfor("ZAP")
[{'acc': 'HGNC:12858',
 'xlink:href': '/HGNC/wr/gene/ZAP70',
 'xlink:title': 'ZAP70'}]
```

This function may be used to count the number of entries:

```
len(s.lookfor('*'))
```

mapping(*value*)

maps an identifier from a database onto HGNC database

Parameters

value (*str*) – a valid DB:id string (e.g. “UniProt:P36888”)

Returns

a list of dictionary with the keys ‘acc’, ‘xlink:href’, ‘xlink:title’

```
>>> value = "UniProt:P43403"
>>> res = s.mapping(value)
>>> res[0]['xlink:title']
'ZAP70'
>>> res[0]['acc']
'HGNC:12858'
```

See also:

[*mapping_all\(\)*](#)

mapping_all(*entries=None*)

Retrieves cross references for more than one entry

Parameters

entries – list of values entries (e.g., returned by the [*lookfor\(\)*](#) method.) if not provided, this method looks for all entries.

Returns

list of dictionaries with keys being all entry names. Values is a dictionary of cross references.

Warning: takes 10 minutes

Intact (complex)

This module provides a class [*IntactComplex*](#)

What is Intact Complex ?

URL

<https://www.ebi.ac.uk/intact/complex/>

REST

<https://www.ebi.ac.uk/intact/complex-ws/details/>

“The Complex Portal is a manually curated, encyclopaedic resource of macromolecular complexes from a number of key model organisms.”

—From Intact web page Feb 2015

class IntactComplex(*verbose=False, cache=False*)

Interface to the [Intact](#) service

```
>>> from bioservices import IntactComplex
>>> u = IntactComplex()
```

Constructor IntactComplex

Parameters

verbose – set to False to prevent informative messages

details(*query*)

Return details about a complex

Parameters

query (*str*) – EBI-1163476

search(*query, frmt='json', facets=None, first=None, number=None, filters=None*)

Search for a complex inside intact complex.

Parameters

- **query** (*str*) – the query (e.g., ndc80)
- **frmt** (*str*) – Defaults to json (could be a Pandas data frame if Pandas is installed; set frmt to 'pandas')
- **facets** (*str*) – lists of facets as a string (separated by comma)
- **first** (*int*) –
- **number** (*int*) –
- **filter** (*str*) – list of filters. See examples here below.

```
s = IntactComplex()
# search for ndc80
s.search('ndc80')

# Search for ndc80 and facet with the species field:
s.search('ndc80', facets='species_f')

# Search for ndc80 and facet with the species and biological role fields:
s.search('ndc80', facets='species_f,pbiorole_f')

# Search for ndc80, facet with the species and biological role
# fields and filter the species using human:
s.search('Ndc80', first=0, number=10,
        filters='species_f:("Homo sapiens")',
        facets='species_f,ptype_f,pbiorole_f')

# Search for ndc80, facet with the species and biological role
# fields and filter the species using human or mouse:
s.search('Ndc80', first=0, number=10,
        filters='species_f:("Homo sapiens" "Mus musculus")',
        facets='species_f,ptype_f,pbiorole_f')

# Search with a wildcard to retrieve all the information:
```

(continues on next page)

(continued from previous page)

```
s.search('*')

# Search with a wildcard to retrieve all the information and facet
# with the species, biological role and interactor type fields:
s.search('*', facets='species_f,pbiorole_f,ptype_f')

# Search with a wildcard to retrieve all the information, facet with
# the species, biological role and interactor type fields and filter
# the interactor type using small molecule:
s.search('*', facets='species_f,pbiorole_f,ptype_f',
          filters='ptype_f("small molecule")')

# Search with a wildcard to retrieve all the information, facet with
# the species, biological role and interactor type fields and filter
# the interactor type using small molecule and the species using human:
s.search('*', facets='species_f,pbiorole_f,ptype_f',
          filters='ptype_f("small molecule"),species_f("Homo sapiens")')

# Search for GO:0016491 and paginate (first is for the offset and number
# is how many do you want):
s.search('GO:0016491', first=10, number=10)
```

The organism name used in the filter must be exact. Here is the list found by typing:

```
res = set(ci.search('*', frmt='pandas')['organismName'])
```

```
'Bos taurus; 9913',
'Caenorhabditis elegans; 6239',
'Canis familiaris; 9615',
'Drosophila melanogaster; 7227',
'Escherichia coli (strain K12); 83333',
'Gallus gallus; 9031',
'Homo sapiens; 9606',
'Mus musculus; 10090',
'Oryctolagus cuniculus; 9986',
'Rattus norvegicus; 10116',
'Saccharomyces cerevisiae (strain ATCC 204508 / S288c);559292',
'Schizosaccharomyces pombe (strain 972 / ATCC 24843);284812',
'Xenopus laevis; 8355'
```

MUSCLE

Interface to the MUSCLE web service

What is MUSCLE ?

URL

<http://www.drive5.com/muscle/>

service

http://www.ebi.ac.uk/Tools/webservices/services/msa/muscle_rest

“MUSCLE - (Multiple Sequence Comparison by Log-Expectation) 1)

is claimed to achieve both better average accuracy and better speed than ClustalW or T-Coffee, depending on the chosen options. Multiple alignments of protein sequences are important in many applications, including phylogenetic tree estimation, secondary structure prediction and critical residue identification.”

—from EMBL-EBI web page

class `MUSCLE(verbose=False)`

Interface to the `MUSCLE` service.

```
>>> from bioservices import *
>>> m = MUSCLE(verbose=False)
>>> sequencesFasta = open('filename', 'r')
>>> jobid = n.run(frmt="fasta", sequence=sequencesFasta.read(),
>>>               email="name@provider")
>>> s.getResult(jobid, "out")
```

Warning: It is very important to provide a real e-mail address as your job otherwise very likely will be killed and your IP, Organisation or entire domain black-listed.

Here is another similar example but we use `UniProt` class provided in bioservices to fetch the FASTA sequences:

```
>>> from bioservices import UniProt, MUSCLE
>>> u = UniProt(verbose=False)
>>> f1 = u.get_fasta("P18413")
>>> f2 = u.get_fasta("P18412")
>>> m = MUSCLE(verbose=False)
>>> jobid = m.run(frmt="fasta", sequence=f1+f2, email="name@provider")
>>> m.getResult(jobid, "out")
```

get_parameter_details(parameterId)

Get detailed information about a parameter.

Returns

An XML document providing details about the parameter or a list of values that can take the parameters if the XML could be parsed.

For example:

```
>>> n.get_parameter_details("format")
```

get_parameters()

List parameter names.

Returns

An XML document containing a list of parameter names.

```
>>> from bioservices import muscle
>>> n = muscle.Muscle()
>>> res = n.get_parameters()
>>> [x.text for x in res.findAll("id")]
```

See also:

[*parameters*](#) to get a list of the parameters without need to process the XML output.

get_result(*jobid*, *result_type*)

Get the job result of the specified type.

Parameters

- **jobid** (*str*) – a job identifier returned by [*run\(\)*](#).
- **resultType** (*str*) – type of result to retrieve. See [*getResultTypes\(\)*](#).

get_result_types(*jobid*)

Get available result types for a finished job.

Parameters

- **jobid** (*str*) – a job identifier returned by [*run\(\)*](#).
- **verbose** (*bool*) – print the identifiers together with their label, mediaTypes, description and filesuffix.

Returns

A dictionary, which keys correspond to the identifiers. Each identifier is itself a dictionary containing the label, description, file suffix and mediaType of the identifier.

get_status(*jobid*)

Get status of a submitted job

Parameters

- **jobid** (*str*) –
- **jobid** – a job identifier returned by [*run\(\)*](#).

Returns

A string giving the jobid status (e.g. FINISHED).

The values for the status are:

- RUNNING: the job is currently being processed.
- FINISHED: job has finished, and the results can then be retrieved.
- ERROR: an error occurred attempting to get the job status.
- FAILURE: the job failed.
- NOT_FOUND: the job cannot be found.

property parameters

run(*frmt=None*, *sequence=None*, *tree='none'*, *email=None*)

Submit a job with the specified parameters.

Compulsary arguments

Parameters

- **frmt** (*str*) – input format (e.g., fasta)
- **sequence** (*str*) – query sequence. The use of fasta formatted sequence is recommended.
- **tree** (*str*) – tree type ('none', 'tree1', 'tree2')
- **email** (*str*) – a valid email address. Will be checked by the service itself.

Returns

A jobid that can be analysed with `getResult()`, `getStatus()`, ...

The up to data values accepted for each of these parameters can be retrieved from the `get_parameter_details()`.

For instance,:

```
from bioservices import MUSCLE
m = MUSCLE()
m.parameterDetails("tree")
```

Example:

```
jobid = m.run(frmt="fasta",
             sequence=sequence_example,
             email="test@yahoo.fr")
```

frmt can be a list of formats:

```
frmt=['fasta', 'clw', 'clwstrict', 'html', 'msf', 'phyi', 'phys']
```

The returned object is a jobid, which status can be checked. It must be finished before analysing/geeting the results.

See also:

`getResult()`

wait(*jobId*, *checkInterval*=5, *verbose*=True)

This function checks the status of a jobid while it is running

Parameters

- **jobid** (*str*) – a job identifier returned by `run()`.
- **checkInterval** (*int*) – interval between requests in seconds.

MyGeneInfo

Interface to the mygeneinfo web Service.

What is MyGeneInfo ?

URL

<https://mygene.info>

REST

<https://mygeneinfo/v3.api/>

MyGene.info provides simple-to-use REST web services to query/retrieve gene annotation data. It's designed with simplicity and performance emphasized. You can use it to power a web application which requires querying genes and obtaining common gene annotations. For example, MyGene.info services are used to power BioGPS; or use it in an analysis pipeline to retrieve always up-to-date gene annotations.

—mygene.info home page, June 2020

```
class MyGeneInfo(verbose=False, cache=False)
```

Interface to mygene.info service

```
>>> from bioservices import MyGeneInfo
>>> s = MyGeneInfo()
```

Constructor

Parameters

verbose (*bool*) – prints informative messages (default is off)

```
get_genes(ids, fields='symbol,name,taxid,entrezgene,ensemblgene', species=None, dotfield=True,
          email=None)
```

Get matching gene objects for a list of gene ids

Parameters

- **ids** – list of geneinfo IDs
- **fields** (*str*) – a comma-separated fields to limit the fields returned from the matching gene hits. The supported field names can be found from any gene object (e.g. <http://mygene.info/v3/gene/1017>). Note that it supports dot notation as well, e.g., you can pass “refseq.rna”. If “fields=all”, all available fields will be returned. Default: “symbol,name,taxid,entrezgene,ensemblgene”.
- **species** (*str*) – can be used to limit the gene hits from given species. You can use “common names” for nine common species (human, mouse, rat, fruitfly, nematode, zebrafish, thale-cress, frog and pig). All other species, you can provide their taxonomy ids. Multiple species can be passed using comma as a separator. Default: human,mouse,rat.
- **dotfield** – control the format of the returned fields when passed “fields” parameter contains dot notation, e.g. “fields=refseq.rna”. If True the returned data object contains a single “refseq.rna” field, otherwise (False), a single “refseq” field with a sub-field of “rna”. Default: True.

- **email** (*str*) – If you are regular users of this services, the mygeneinfo maintainers/authors encourage you to provide an email, so that we can better track the usage or follow up with you.

```

mgi = MyGeneInfoe()
mgi.get_genes(("301345,22637"))
# first one is rat, second is mouse. This will return a 'notfound'
# entry and the second entry as expected.
mgi.get_genes("301345,22637", species="mouse")

```

get_metadata()

get_one_gene(*geneid*, *fields*='symbol,name,taxid,entrezgene,ensemblgene', *dotfield*=True, *email*=None)

Get matching gene objects for one gene id

Parameters

- **geneid** – a valid gene ID
- **fields** (*str*) – a comma-separated fields to limit the fields returned from the matching gene hits. The supported field names can be found from any gene object (e.g. <http://mygene.info/v3/gene/1017>). Note that it supports dot notation as well, e.g., you can pass “refseq.rna”. If “fields=all”, all available fields will be returned. Default: “symbol,name,taxid,entrezgene,ensemblgene”.
- **dotfield** – control the format of the returned fields when passed “fields” parameter contains dot notation, e.g. “fields=refseq.rna”. If True the returned data object contains a single “refseq.rna” field, otherwise (False), a single “refseq” field with a sub-field of “rna”. Default: True.
- **email** (*str*) – If you are regular users of this services, the mygeneinfo maintainers/authors encourage you to provide an email, so that we can better track the usage or follow up with you.

```

mgi = MyGeneInfoe()
mgi.get_genes("301345")

```

get_one_query(*query*, *email*=None, *dotfield*=True, *fields*='symbol,name,taxid,entrezgene,ensemblgene', *species*='human,mouse,rat', *size*=10, *_from*=0, *sort*=None, *facets*=None, *entrezonly*=False, *ensemblonly*=False)

Make gene query and return matching gene list. Support JSONP and CORS as well.

Parameters

- **query** (*str*) – Query string. Examples “CDK2”, “NM_052827”, “204639_at”, “chr1:151,073,054-151,383,976”, “hg19.chr1:151073054-151383976”. The detailed query syntax can be found from our docs.
- **fields** (*str*) – a comma-separated fields to limit the fields returned from the matching gene hits. The supported field names can be found from any gene object (e.g. <http://mygene.info/v3/gene/1017>). Note that it supports dot notation as well, e.g., you can pass “refseq.rna”. If “fields=all”, all available fields will be returned. Default: “symbol,name,taxid,entrezgene,ensemblgene”.
- **species** (*str*) – can be used to limit the gene hits from given species. You can use “common names” for nine common species (human, mouse, rat, fruitfly, nematode, zebrafish, thale-cress, frog and pig). All other species, you can provide their taxonomy ids. Multiple species can be passed using comma as a separator. Default: human,mouse,rat.

- **size** (*int*) – the maximum number of matching gene hits to return (with a cap of 1000 at the moment). Default: 10.
- **_from** (*int*) – the number of matching gene hits to skip, starting from 0. Combining with “size” parameter, this can be useful for paging. Default: 0.
- **sort** – the comma-separated fields to sort on. Prefix with “-” for descending order, otherwise in ascending order. Default: sort by matching scores in descending order.
- **facets** (*str*) – a single field or comma-separated fields to return facets, for example, “facets=taxid”, “facets=taxid,type_of_gene”.
- **entrezonly** (*bool*) – when passed as True, the query returns only the hits with valid Entrez gene ids. Default: False.
- **ensembleonly** (*bool*) – when passed as True, the query returns only the hits with valid Ensembl gene ids. Default: False.
- **dotfield** – control the format of the returned fields when passed “fields” parameter contains dot notation, e.g. “fields=refseq.rna”. If True the returned data object contains a single “refseq.rna” field, otherwise (False), a single “refseq” field with a sub-field of “rna”. Default: True.
- **email** (*str*) – If you are regular users of this services, the mygeneinfo maintainers/authors encourage you to provide an email, so that we can better track the usage or follow up with you.

get_queries(*query*, *email*=None, *dotfield*=True, *scopes*='all', *species*='human,mouse,rat', *fields*='symbol,name,taxid,entrezgene,ensemblgene')

Make gene query and return matching gene list. Support JSONP and CORS as well.

Parameters

- **query** (*str*) – Query string. Examples “CDK2”, “NM_052827”, “204639_at”, “chr1:151,073,054-151,383,976”, “hg19.chr1:151073054-151383976”. The detailed query syntax can be found from our docs.
- **fields** (*str*) – a comma-separated fields to limit the fields returned from the matching gene hits. The supported field names can be found from any gene object (e.g. <http://mygene.info/v3/gene/1017>). Note that it supports dot notation as well, e.g., you can pass “refseq.rna”. If “fields=all”, all available fields will be returned. Default: “symbol,name,taxid,entrezgene,ensemblgene”.
- **species** (*str*) – can be used to limit the gene hits from given species. You can use “common names” for nine common species (human, mouse, rat, fruitfly, nematode, zebrafish, thale-cress, frog and pig). All other species, you can provide their taxonomy ids. Multiple species can be passed using comma as a separator. Default: human,mouse,rat.
- **dotfield** – control the format of the returned fields when passed “fields” parameter contains dot notation, e.g. “fields=refseq.rna”. If True the returned data object contains a single “refseq.rna” field, otherwise (False), a single “refseq” field with a sub-field of “rna”. Default: True.
- **email** (*str*) – If you are regular users of this services, the mygeneinfo maintainers/authors encourage you to provide an email, so that we can better track the usage or follow up with you.
- **scopes** (*str*) – not documented. Set to ‘all’

get_taxonomy()

NCBIblast

Interface to the NCBI BLAST web service

What is NCBI BLAST ?

URL

<http://blast.ncbi.nlm.nih.gov/>

service

http://www.ebi.ac.uk/Tools/webservices/services/sss/ncbi_blast_rest

“NCBI BLAST - Protein Database Query

The emphasis of this tool is to find regions of sequence similarity, which will yield functional and evolutionary clues about the structure and function of your novel sequence.”

—from NCBIblast web page

class NCBIblast(*verbose=False*)

Interface to the `NCBIblast` service.

```
>>> from bioservices import *
>>> s = NCBIblast(verbose=False)
>>> jobid = s.run(program="blastp", sequence=s._sequence_example,
>>>               stype="protein", database="uniprotkb", email="name@provider")
>>> s.getResult(jobid, "out")
```

Warning: It is very important to provide a real e-mail address as your job otherwise very likely will be killed and your IP, Organisation or entire domain black-listed.

When running a blast request, a program is required. You can obtain the list using:

```
>>> s.parametersDetails("program")
[u'blastp', u'blastx', u'blastn', u'tblastx', u'tblastn']
```

- blastn: Search a nucleotide database using a nucleotide query
- blastp: Search protein database using a protein query
- blastx: Search protein database using a translated nucleotide query
- tblastn Search translated nucleotide database using a protein query
- tblastx Search translated nucleotide database using a translated nucleotide query

NCBIblast constructor

Parameters

verbose (*bool*) – prints informative messages

property databases

Returns accepted databases.

get_parameter_details(*parameterId*)

Get detailed information about a parameter.

Returns

An XML document providing details about the parameter or a list of values that can take the parameters if the XML could be parsed.

For example:

```
>>> s.parameter_details("matrix")
[u'BLOSUM45',
 u'BLOSUM50',
 u'BLOSUM62',
 u'BLOSUM80',
 u'BLOSUM90',
 u'PAM30',
 u'PAM70',
 u'PAM250']
```

get_parameters()

List parameter names.

Returns

An XML document containing a list of parameter names.

```
>>> from bioservices import ncbiblast
>>> n = ncbiblast.NCBIBlast()
>>> res = n.get_parameters()
>>> [x.text for x in res.findAll("id")]
```

See also:

[parameters](#) to get a list of the parameters without need to process the XML output.

get_result(*jobid*, *result_type*)

Get the job result of the specified type.

param str jobid

a job identifier returned by [run\(\)](#).

param str result_type

type of result to retrieve. See [getResultTypes\(\)](#).

The output from the tool itself. Use the ‘format’ parameter to retrieve the output in different formats, the ‘compressed’ parameter to retrieve the xml output in compressed form. Format options:

```
0 = pairwise,
1 = query-anchored showing identities,
2 = query-anchored no identities,
```

(continues on next page)

(continued from previous page)

```

3 = flat query-anchored showing identities,
4 = flat query-anchored no identities,
5 = XML Blast output,
6 = tabular,
7 = tabular with comment lines,
8 = Text ASN.1,
9 = Binary ASN.1,
10 = Comma-separated values,
11 = BLAST archive format (ASN.1).

```

See NCBI Blast documentation for details. Use the 'compressed' parameter to return the XML output in compressed form. e.g. '?format=5&compressed=true'.

get_result_types(*jobid*)

Get available result types for a finished job.

Parameters

- **jobid** (*str*) – a job identifier returned by *run()*.
- **verbose** (*bool*) – print the identifiers together with their label, mediaTypes, description and filesuffix.

Returns

A dictionary, which keys correspond to the identifiers. Each identifier is itself a dictionary containing the label, description, file suffix and mediaType of the identifier.

get_status(*jobid*)

Get status of a submitted job

Parameters

- **jobid** (*str*) –
- **jobid** – a job identifier returned by *run()*.

Returns

A string giving the jobid status (e.g. FINISHED).

The values for the status are:

- **RUNNING**: the job is currently being processed.
- **FINISHED**: job has finished, and the results can then be retrieved.
- **ERROR**: an error occurred attempting to get the job status.
- **FAILURE**: the job failed.
- **NOT_FOUND**: the job cannot be found.

property parameters

run(*program=None, database=None, sequence=None, stype='protein', email=None, **kargs*)

Submit a job with the specified parameters.

Compulsary arguments

Parameters

- **program** (*str*) – BLAST program to use to perform the search (e.g., blastp)
- **sequence** (*str*) – query sequence. The use of fasta formatted sequence is recommended.
- **database** (*list*) – list of database names for search or possible a single string (for one database). There are some mismatch between the output of parametersDetails(“database”) and the accepted values. For instance UniProt Knowledgebase should be given as “uniprotkb”.
- **email** (*str*) – a valid email address. Will be checked by the service itself.

Optional arguments. If not provided, a default value will be used

Parameters

- **type** (*str*) – query sequence type in ‘dna’, ‘rna’ or ‘protein’ (default is protein).
- **matrix** (*str*) – scoring matrix to be used in the search (e.g., BLOSUM45).
- **gapalign** (*bool*) – perform gapped alignments.
- **alignments** (*int*) – maximum number of alignments displayed in the output.
- **exp** – E-value threshold.
- **filter** (*bool*) – low complexity sequence filter to process the query sequence before performing the search.
- **scores** (*int*) – maximum number of scores displayed in the output.
- **dropoff** (*int*) – amount score must drop before extension of hits is halted.
- **match_scores** – match/miss-match scores to generate a scoring matrix for nucleotide searches.
- **gapopen** (*int*) – penalty for the initiation of a gap.
- **gapext** (*int*) – penalty for each base/residue in a gap.
- **seqrange** – region of the query sequence to use for the search. Default: whole sequence.

Returns

A jobid that can be analysed with `getResult()`, `getStatus()`, ...

The up to data values accepted for each of these parameters can be retrieved from the [`get_parameter_details\(\)`](#).

For instance,:

```
from bioservices import NCBIblast
n = NCBIblast()
n.get_parameter_details("program")
```

Example:

```

jobid = n.run(program="blastp",
              sequence=n._sequence_example,
              stype="protein",
              database="uniprotkb",
              email="test@yahoo.fr")

```

Database can be a list of databases:

```

database=["uniprotkb", "uniprotkb_swissprot"]

```

The returned object is a jobid, which status can be checked. It must be finished before analysing/geeting the results.

See also:

`getResult()`

Warning: Cases are not important. Spaces in the database case should be replaced by underscore.

Note: database returned by the server have meaningless names since they do not map to the expected names. An example is “ENA Sequence Release” that should be provided as `em_rel`

<http://www.ebi.ac.uk/Tools/sss/ncbiblast/help/index-nucleotide.html>

wait(*jobId*)

This function checks the status of a jobid while it is running

Parameters

- **jobid** (*str*) – a job identifier returned by `run()`.
- **checkInterval** (*int*) – interval between requests in seconds.

OmniPath Commons

Interface to OmniPath web service

What is OmniPath ?

URL

<http://omnipathdb.org>

URL

<https://github.com/saezlab/pypath/blob/master/webservice.rst>

A comprehensive collection of literature curated human signaling pathways.

—From OmniPath web site, March 2016

```

class OmniPath(verbose=False, cache=False)

```

Interface to the [OmniPath](#) service

```
>>> from bioservices import OmniPath
>>> o = OmniPath()
>>> net = o.get_network()
>>> interactions = o.get_interactions('P00533')
```

Constructor OmniPath

Parameters

verbose – set to False to prevent informative messages

get_about()

Information about the version

get_info()

Currently returns HTML page

get_interactions(query="", frmt='json', fields=[])

Interactions of proteins

Parameters

- **query** (*str*) – a valid uniprot identifier (e.g. P00533). It can also be a list of uniprot identifiers, or a string with comma-separated identifiers.
- **fields** (*str*) – additional fields to be added to the output (e.g., sources, references)
- **frmt** (*str*) – format of the output (json or tabular)

Example:

```
res_one = o.get_interactions('P00533')
res_many = o.get_interactions('P00533,015117,Q96FE5')
res_many = o.get_interactions(['P00533', '015117', 'Q96FE5'])

res_one = o.get_interactions('P00533', fields='sources')
res_one = o.get_interactions('P00533', fields=['source'])
res_one = o.get_interactions('P00533', fields=['source', 'references'])
```

You may also keep query to an empty string, but the entire DB will then be downloaded. This may take time and the timeout may need to be increased manually.

If frmt is set to TSV, the output is a TSV table with a header. If set to json, a dictionary is returned.

get_network(frmt='json')

Get basic statistics about the whole network including sources

get_ptms(query="", ptm_type=None, frmt='json', fields=[])

List enzymes, substrates and PTMs

Parameters

- **query** (*str*) – a valid uniprot identifier (e.g. P00533). It can also be a list of uniprot identifiers, or a string with comma-separated identifiers.
- **ptm_type** (*str*) – restrict the output to this type of PTM (e.g., phosphorylation)
- **fields** (*str*) – additional fields to be added to the output (e.g., sources, references)

get_resources(frmt='json')

Return statistics about the databases and their contents

Panther

Interface to some part of the Panther web service

What is Panther ?

URL

<http://www.panther.org>

Citation

The PANTHER (Protein ANalysis THrough Evolutionary Relationships) Classification System was designed to classify proteins (and their genes) in order to facilitate high-throughput analysis. Proteins have been classified according to:

- Family and subfamily: families are groups of evolutionarily related proteins; subfamilies are related proteins that also have the same function
- Molecular function: the function of the protein by itself or with directly interacting proteins at a biochemical level, e.g. a protein kinase
- Biological process: the function of the protein in the context of a larger network of proteins that interact to accomplish a process at the level of the cell or organism, e.g. mitosis.
- Pathway: similar to biological process, but a pathway also explicitly specifies the relationships between the interacting molecules.

—From PantherDB (about) , Feb 2020

class Panther(verbose=True, cache=False)

Interface to Panther pages

```
>>> from bioservices import Panther
>>> p = Panther()
>>> p.get_supported_genomes()
>>> p.get_ortholog("zap70", 9606)

>>> from bioservices import Panther
>>> p = Panther()
>>> taxon = [x[0]['taxon_id'] for x in p.get_supported_genomes() if "coli" in x[
↳ 'name'].lower()]
>>> # you may also use our method called search_organism
>>> taxon = p.get_taxon_id(pattern="coli")
>>> res = p.get_mapping("abrB,ackA,acuI", taxon)
```

The `get_mapping` returns for each gene ID the GO terms corresponding to each ID. Those go terms may belong to different categories (see `meth:get_annotation_datasets`):

- MF for molecular function
- BP for biological process
- PC for Protein class
- CC Cellular location
- Pathway

Note that results from the website application <http://pantherdb.org/> do not agree with the output of the get_mapping service... Try out the dgt gene from ecoli for example

Constructor

Parameters

verbose – set to False to prevent informative messages

get_annotation_datasets()

Retrieve the list of supported annotation data sets

get_enrichment(*gene_list, organism, annotation, enrichment_test='Fisher', correction='FDR', ref_gene_list=None*)

Returns over represented genes

Compares a test gene list to a reference gene list, and determines whether a particular class (e.g. molecular function, biological process, cellular component, PANTHER protein class, the PANTHER pathway or Reactome pathway) of genes is overrepresented or underrepresented.

Parameters

- **organism** – a valid taxon ID
- **enrichment_test** – either **Fisher** or **Binomial** test
- **correction** – correction for multiple testing. Either **FDR**, **Bonferonni**, or **None**.
- **annotation** – one of the supported PANTHER annotation data types. See [get_annotation_datasets\(\)](#) to retrieve a list of supported annotation data types
- **ref_gene_list** – if not specified, the system will use all the genes for the specified organism. Otherwise, a list delimited by comma. Maximum of 100000 Identifiers can be any of the following: Ensemble gene identifier, Ensemble protein identifier, Ensemble transcript identifier, Entrez gene id, gene symbol, NCBI GI, HGNC Id, International protein index id, NCBI UniGene id, UniProt accession and UniProt id.

Returns

a dictionary with the following keys. 'reference' contains the organism, 'input_list' is the input gene list with unmapped genes. 'result' contains the list of candidates.

```
>>> from bioservices import Panther
>>> p = Panther()
>>> res = p.get_enrichment('zap70,mek1,erk', 9606, "GO:0008150")
>>> For molecular function, use :
>>> res = p.get_enrichment('zap70,mek1,erk', 9606,
    "ANNOT_TYPE_ID_PANTHER_GO_SLIM_MF")
```

get_family_msa(*family, taxon_list=None*)

Returns MSA information for the specified family.

Parameters

- **family** – family ID
- **taxon_list** – Zero or more taxon IDs separated by ','.

get_family_ortholog(*family, taxon_list=None*)

Search for matching orthologs in target organisms

Also return the corresponding position in the target organism sequence. The system searches for matching orthologs in the gene family that contains the search gene associated with the search term.

Parameters

- **family** – Family ID
- **taxon_list** – Zero or more taxon IDs separated by ‘,’.

get_homolog_position(*gene, organism, position, ortholog_type='all'*)

Parameters

- **gene** – Can be any of the following: Ensemble gene identifier, Ensemble protein identifier, Ensemble transcript identifier, Entrez gene id, gene symbol, NCBI GI, HGNC Id, International protein index id, NCBI UniGene id, UniProt accession and UniProt id
- **organism** – a valid taxon ID
- **ortholog_type** – optional parameter to specify ortholog type of target organism

get_mapping(*gene_list, taxon*)

Map identifiers

Each identifier to be delimited by comma i.e. ‘,’. Maximum of 1000 Identifiers can be any of the following: Ensemble gene identifier, Ensemble protein identifier, Ensemble transcript identifier, Entrez gene id, gene symbol, NCBI GI, HGNC Id, International protein index id, NCBI UniGene id, UniProt accession and UniProt id

Parameters

- **gene_list** – see above
- **taxon** – one taxon ID. See supported [get_supported_genomes\(\)](#)

If an identifier is not found, information can be found in the unmapped_genes key while found identifiers are in the mapped_genes key.

Warning: found and not found identifiers are dispatched into unmapped and mapped genes. If there are not found identifiers, the input gene list and the mapped genes list do not have the same length. The input names are not stored in the output. Developers should be aware of that feature.

get_ortholog(*gene_list, organism, target_organism=None, ortholog_type='all'*)

search for matching orthologs in target organisms.

Searches for matching orthologs in the gene family that contains the search gene associated with the search terms. Returns ortholog genes in target organisms given a search organism, the search terms and a list of target organisms.

Parameters

- **gene_list** –
- **organism** – a valid taxon ID
- **target_organism** – zero or more taxon IDs separated by ‘,’. See [get_supported_genomes\(\)](#)
- **ortholog_type** – optional parameter to specify ortholog type of target organism

Returns

a dictionary with “mapped” and “unmapped” keys, each of them being a list. For each unmapped gene, a dictionary with id and organism is returned. For the mapped gene, a list of ortholog is returned.

get_pathways()

Returns all pathways from pantherdb

get_supported_families(*N=1000, progress=True*)

Returns the list of supported PANTHER family IDs

This services returns only 1000 items per request. This is defined by the index. For instance index set to 1 returns the first 1000 families. Index set to 2 returns families between index 1000 and 2000 and so on. As of 20 Feb 2020, there was about 15,000 families.

This function simplifies your life by calling the service as many times as required. Therefore it returns all families in one go.

get_supported_genomes(*type=None*)

Returns list of supported organisms.

Parameters

type – can be chrLoc to restrict the search

get_taxon_id(*pattern=None*)

return all taxons supported by the service

If pattern is provided, we filter the name to keep those that contain the filter. If only one is found, we return the name itself, otherwise a list of candidates

get_tree_info(*family, taxon_list=None*)

Returns tree topology information and node attributes for the specified family.

Parameters

- **family** – Family ID
- **taxon_list** – Zero or more taxon IDs separated by ‘,’.

Pathway Commons

This module provides a class *PathwayCommons*

What is PathwayCommons ?

URL

<http://www.pathwaycommons.org/about>

REST

Pathway Commons is a convenient point of access to biological pathway information collected from public pathway databases, which you can search, visualize and download. All data is freely available, under the license terms of each contributing database.

—PathwayCommons home page, Nov 2013

Data is freely available, under the license terms of each contributing database.

class PathwayCommons(*verbose=True, cache=False*)

Interface to the *PathwayCommons* service

```
>>> from bioservices import *
>>> pc2 = PathwayCommons(verbose=False)
>>> res = pc2.get("http://identifiers.org/uniprot/Q06609")
```

Todo: traverse() method not implemented.

Constructor

Parameters

verbose (*bool*) – prints informative messages

property default_extension

set extension of the requests (default is json). Can be 'json' or 'xml'

get(uri, frmt='BIOPAX')

Retrieves full pathway information for a set of elements

elements can be for example pathway, interaction or physical entity given the RDF IDs. Get commands only retrieve the BioPAX elements that are directly mapped to the ID. Use the [traverse\(\)](#) query to traverse BioPAX graph and obtain child/owner elements.

Parameters

- **uri** (*str*) – valid/existing BioPAX element's URI (RDF ID; for utility classes that were "normalized", such as entity refereneces and controlled vocabularies, it is usually a Identifiers.org URL. Multiple IDs can be provided using list uri=[<http://identifiers.org/uniprot/Q06609>, <http://identifiers.org/uniprot/Q549Z0>] See also about MIRIAM and Identifiers.org.
- **format** (*str*) – output format (values)

Returns

a complete BioPAX representation for the record pointed to by the given URI is returned. Other output formats are produced by converting the BioPAX record on demand and can be specified by the optional format parameter. Please be advised that with some output formats it might return "no result found" error if the conversion is not applicable for the BioPAX result. For example, BINARY_SIF output usually works if there are some interactions, complexes, or pathways in the retrieved set and not only physical entities.

```
>>> from bioservices import PathwayCommons
>>> pc2 = PathwayCommons(verbose=False)
>>> res = pc2.get("col5a1")
>>> res = pc2.get("http://identifiers.org/uniprot/Q06609")
```

get_sifgraph_common_stream(source, limit=1, direction='DOWNSTREAM', pattern=None)

finds the common stream for them; extracts a sub-network from the loaded Pathway Commons SIF model.

Parameters

- **source** – set of gene identifiers (HGNC symbol). Can be a list of identifiers or just one string(if only one identifier)
- **limit** (*int*) – Graph traversal depth. Limit > 1 value can result in very large data or error.

- **direction** (*str*) – Graph traversal direction. Use `UNDIRECTED` if you want to see interacts-with relationships too.
- **pattern** (*str*) – Filter by binary relationship (SIF edge) type(s). one of “BOTH-STREAM”, “UPSTREAM”, “DOWNSTREAM”, “UNDIRECTED”.

returns: the graph in SIF format. The output must be stripped and

returns one line per relation. In each line, items are separated by a tabulation. You can save the text with .sif extensions and it should be ready to use e.g. in cytoscape viewer.

```
res = pc.get_sifgraph_common_stream(['BRD4', 'MYC'])
```

get_sifgraph_neighborhood(*source, limit=1, direction='BOTHSTREAM', pattern=None*)

finds the neighborhood sub-network in the Pathway Commons Simple Interaction Format (extended SIF) graph (see <http://www.pathwaycommons.org/pc2/formats#sif>)

Parameters

- **source** – set of gene identifiers (HGNC symbol). Can be a list of identifiers or just one string(if only one identifier)
- **limit** (*int*) – Graph traversal depth. Limit > 1 value can result in very large data or error.
- **direction** (*str*) – Graph traversal direction. Use `UNDIRECTED` if you want to see interacts-with relationships too.
- **pattern** (*str*) – Filter by binary relationship (SIF edge) type(s). one of “BOTH-STREAM”, “UPSTREAM”, “DOWNSTREAM”, “UNDIRECTED”.

returns: the graph in SIF format. The output must be stripped and

returns one line per relation. In each line, items are separated by a tabulation. You can save the text with .sif extensions and it should be ready to use e.g. in cytoscape viewer.

```
res = pc.get_sifgraph_neighborhood('BRD4')
```

get_sifgraph_pathsbetween(*source, limit=1, directed=False, pattern=None*)

finds the paths between them; extracts a sub-network from the Pathway Commons SIF graph.

Parameters

- **source** – set of gene identifiers (HGNC symbol). Can be a list of identifiers or just one string(if only one identifier)
- **limit** (*int*) – Graph traversal depth. Limit > 1 value can result in very large data or error.
- **directed** (*bool*) – Directionality: ‘true’ is for DOWNSTREAM/UPSTREAM, ‘false’ - UNDIRECTED
- **pattern** (*str*) – Filter by binary relationship (SIF edge) type(s). one of “BOTH-STREAM”, “UPSTREAM”, “DOWNSTREAM”, “UNDIRECTED”.

returns: the graph in SIF format. The output must be stripped and

returns one line per relation. In each line, items are separated by a tabulation. You can save the text with .sif extensions and it should be ready to use e.g. in cytoscape viewer.

get_sifgraph_pathsfromto(*source, target, limit=1, pattern=None*)

finds the paths between them; extracts a sub-network from the Pathway Commons SIF graph.

Parameters

source – set of gene identifiers (HGNC symbol). Can be a list of identifiers or just one string(if only one identifier)

param target: A target set of gene identifiers. :param int limit: Graph traversal depth. Limit > 1 value can result

in very large data or error.

Parameters

pattern (*str*) – Filter by binary relationship (SIF edge) type(s). one of “BOTHSTREAM”, “UPSTREAM”, “DOWNSTREAM”, “UNDIRECTED”.

returns: the graph in SIF format. The output must be stripped and

returns one line per relation. In each line, items are separated by a tabulation. You can save the text with .sif extensions and it should be ready to use e.g. in cytoscape viewer.

graph(*kind, source, target=None, direction=None, limit=1, frmt=None, datasource=None, organism=None*)

Finds connections and neighborhoods of elements

Connections can be for example the shortest path between two proteins or the neighborhood for a particular protein state or all states.

Graph searches take detailed BioPAX semantics such as generics or nested complexes into account and traverse the graph accordingly. The starting points can be either physical entites or entity references.

In the case of the latter the graph search starts from ALL the physical entities that belong to that particular entity references, i.e. all of its states. Note that we integrate BioPAX data from multiple databases based on our proteins and small molecules data warehouse and consistently normalize UnificationXref, EntityReference, Provenance, BioSource, and ControlledVocabulary objects when we are absolutely sure that two objects of the same type are equivalent. We, however, do not merge physical entities and reactions from different sources as matching and aligning pathways at that level is still an open research problem. As a result, graph searches can return several similar but disconnected sub-networks that correspond to the pathway data from different providers (though some physical entities often refer to the same small molecule or protein reference or controlled vocabulary).

Parameters

- **kind** (*str*) – graph query
- **source** (*str*) – source object’s URI/ID. Multiple source URIs/IDs must be encoded as list of valid URI **source**=[‘<http://identifiers.org/uniprot/Q06609>’, ‘<http://identifiers.org/uniprot/Q549Z0>’].
- **target** (*str*) – required for PATHSFROMTO graph query. target URI/ID. Multiple target URIs must be encoded as list (see source parameter).
- **direction** (*str*) – graph search direction in [BOTHSTREAM, DOWNSTREAM, UPSTREAM] see `_valid_directions` attribute.
- **limit** (*int*) – graph query search distance limit (default = 1).
- **format** (*str*) – output format. see `_valid-format`
- **datasource** (*str*) – datasource filter (same as for ‘search’).
- **organism** (*str*) – organism filter (same as for ‘search’).

Returns

By default, graph queries return a complete BioPAX representation of the subnetwork matched by the algorithm. Other output formats are available as specified by the optional

format parameter. Please be advised that some output format choices might cause “no result found” error if the conversion is not applicable for the BioPAX result (e.g., BINARY_SIF output fails if there are no interactions, complexes, nor pathways in the retrieved set).

```
>>> from bioservices import PathwayCommons
>>> pc2 = PathwayCommons(verbose=False)
>>> res = pc2.graph(source="http://identifiers.org/uniprot/P20908",
                    kind="neighborhood", format="EXTENDED_BINARY_SIF")
```

search(*q*, *page*=0, *datasource*=None, *organism*=None, *type*=None)

Text search in PathwayCommons using Lucene query syntax

Some of the parameters are BioPAX properties, others are composite relationships.

All index fields are (case-sensitive): comment, ecnumber, keyword, name, pathway, term, xrefdb, xrefid, dataSource, and organism.

The pathway field maps to all participants of pathways that contain the keyword(s) in any of its text fields.

Finally, keyword is a transitive aggregate field that includes all searchable keywords of that element and its child elements.

All searches can also be filtered by data source and organism.

It is also possible to restrict the domain class using the ‘type’ parameter.

This query can be used standalone or to retrieve starting points for graph searches.

Parameters

- **q** (*str*) – requires a keyword, name, external identifier, or a Lucene query string.
- **page** (*int*) – (N>=0, default is 0), search result page number.
- **datasource** (*str*) – filter by data source (use names or URIs of pathway data sources or of any existing Provenance object). If multiple data source values are specified, a union of hits from specified sources is returned. `datasource=[reactome,pid]` returns hits associated with Reactome or PID.
- **organism** (*str*) – The organism can be specified either by official name, e.g. “homo sapiens” or by NCBI taxonomy id, e.g. “9606”. Similar to data sources, if multiple organisms are declared a union of all hits from specified organisms is returned. For example `organism=[9606, 10016]` returns results for both human and mice.
- **type** (*str*) – BioPAX class filter. (e.g., ‘pathway’, ‘proteinreference’)

```
>>> from bioservices import PathwayCommons
>>> pc2 = PathwayCommons(verbose=False)
>>> pc2.search("Q06609")
>>> pc2.search("brca2", type="proteinreference",
              organism="homo sapiens", datasource="pid")
>>> pc2.search("name:'col5a1'", type="proteinreference", organism=9606)
>>> pc2.search("a*", page=3)
```

Find the FGFR2 keyword:

```
pc2.search("FGFR2")
```

Find pathways by FGFR2 keyword in any index field.:


```
pc2.search("FGFR2", type="pathway")
```

Finds control interactions that contain the word binding but not transcription in their indexed fields:

```
pc2.search("binding NOT transcription", type="control")
```

Find all interactions that directly or indirectly participate in a pathway that has a keyword match for “immune” (Note the star after immune):

```
pc.search("pathway:immune*", type="conversion")
```

Find all Reactome pathways:

```
pc.search("*", type="pathway", datasource="reactome")
```

top_pathways(*query='*', datasource=None, organism=None*)

This command returns all *top* pathways

Pathways can be top or pathways that are neither ‘controlled’ nor ‘pathwayComponent’ of another process.

param query

a keyword, name, external identifier or lucene query string like in ‘search’. Default is “*”

param str datasource

filter by data source (same as search)

param str organism

organism filter. 9606 for human.

return

dictionary with information about top pathways. Check the “searchHit” key for information about “dataSource” for instance

```
>>> from bioservices import PathwayCommons
>>> pc2 = PathwayCommons(verbose=False)
>>> res = pc2.top_pathways()
```

https://www.pathwaycommons.org/pc2/top_pathways?q=TP53

traverse(*uri, path*)

Provides XPath-like access to the PC.

The format of the path query is in the form:

```
[InitialClass]/[property1]:[classRestriction(optional)]/[property2]... A "*"
```

sign after the property instructs path accessor to transitively traverse that property. For example, the following path accessor will traverse through all physical entity components within a complex:

```
"Complex/component*/entityReference/xref:UnificationXref"
```

The following will list display names of all participants of interactions, which are components (pathway-Component) of a pathway (note: pathwayOrder property, where same or other interactions can be reached, is not considered here):

```
"Pathway/pathwayComponent:Interaction/participant*/displayName"
```

The optional parameter `classRestriction` allows to restrict/filter the returned property values to a certain subclass of the range of that property. In the first example above, this is used to get only the Unification Xrefs. Path accessors can use all the official BioPAX properties as well as additional derived classes and parameters in paxtools such as inverse parameters and interfaces that represent anonymous union classes in OWL. (See Paxtools documentation for more details).

Parameters

- **uri** (*str*) – a biopax element URI - specified similar to the ‘GET’ command. multiple IDs are allowed as a list of strings.
- **path** (*str*) – a BioPAX property path in the form of `property1[:type1]/property2[:type2]`; see above, inverse properties, Paxtools, `org.biopax.paxtools.controller.PathAccessor`.

See also:

[properties](#)

Returns

XML result that follows the Search Response XML Schema (TraverseResponse type; pagination is disabled: returns all values at once)

```
from bioservices import PathwayCommons
pc2 = PathwayCommons(verbose=False)
res = pc2.traverse(uri=['http://identifiers.org/uniprot/P38398', 'http://
identifiers.org/uniprot/Q06609'], path="ProteinReference/organism")
res = pc2.traverse(uri="http://identifiers.org/uniprot/Q06609",
    path="ProteinReference/entityReferenceOf:Protein/name")
res = pc2.traverse("http://identifiers.org/uniprot/P38398",
    path="ProteinReference/entityReferenceOf:Protein")
res = pc2.traverse(uri=["http://identifiers.org/uniprot/P38398",
    "http://identifiers.org/taxonomy/9606"], path="Named/name")
```

PDB/PDBe modules

Interface to the PDB web Service (New API Jan 2021).

What is PDB ?

URL

<http://www.rcsb.org/pdb/>

REST

<http://search.rcsb.org/#search-api>

An Information Portal to Biological Macromolecular Structures

—PDB home page, Jan 2021

```
class PDB(verbose=False, cache=False)
```

Interface to [PDB](#) service (new API Jan 2021)

With the new API, one method called `search()` is provided by PDB. To perform a search you need to define a query. Here is an example

```
>>> from bioservices import PDB
>>> s = PDB()
>>> query = {"query":
...         {"type": "terminal",
...          "service": "text",
...          "parameters": {
...              "value": "thymidine kinase"
...          }
...         },
...         "return_type": "entry"}
>>> res = s.search(query, return_type=return_type)
```

Note: as of December 2020, a new API has been set up by PDB. Some previous functionalities such as return list of Ligand are not supported anymore (Jan 2021). However, many more powerful searches are available. I encourage everyone to look at the PDB page for complex examples: <http://search.rcsb.org/#examples>

As mentioned above, the PDB service provides one method called `search` available in `search()`. We will not cover all the power and capability of this search function. Users should refer to the official PDB help for that. Yet, given examples from PDB should all work with this method.

When possible, we will add convenient aliases function in this class. For now we have for example the `get_current_ids()` and `get_similarity_sequence()` that users may find useful.

The main idea behind the PDB API is to create queries that can access to different types of services. A query will need to at least two keys:

- **query**
- **return_type**

Consider this basic example that searches for the text *thymidine kinase*:

```
{
  "query": {
    "type": "terminal",
    "service": "text",
    "parameters": {
      "value": "thymidine kinase"
    }
  },
  "return_type": "entry"
}
```

Here the query is defined by a **query** and a **return_type** indeed. The return type is a simple value such as **entry**. The query itself is composed of 3 pairs of key/value. Here we have the type service and parameters as defined below.

The query can have several fields:

- **type**: the clause type can be either **terminal** or **group**
 - **terminal**: performs an atomic search operation, e.g. searches for a particular value in a particular field.

- **group**: wraps other terminal or group nodes and is used to combine multiple queries in a logical fashion.
- **service**:
 - **text**: linguistic searches against textual annotations.
 - **sequence**: uses MMSeq2 to perform sequence matching searches (blast-like). following targets that are available:
 - * `pdb_protein_sequence`,
 - * `pdb_dna_sequence`,
 - * `pdb_na_sequence`
 - **seqmotif**: performs short motif searches against nucleotide or protein sequences using 3 different inputs:
 - * `simple` (e.g., `CXCXXL`)
 - * `prosite` (e.g., `C-X-C-X(2)-[LIVMYFWC]`)
 - * `regex` (e.g., `CXCX{2}[LIVMYFWC]`)
 - **structure**: searches matching a global 3D shape of assemblies or chains of a given entry (identified by PDB ID), in either strict (`strict_shape_match`) or relaxed (`relaxed_shape_match`) modes
 - **strucmotif**: Performs structural motif searches on all available PDB structures.
 - **chemical**: queries of small-molecule constituents of PDB structures, based on chemical formula and chemical structure. Queries for matching and similar chemical structures can be performed using SMILES and InChI descriptors as search targets.
 - * `graph-strict`: atom type, formal charge, bond order, atom and bond chirality, aromatic assignment are used as matching criteria for this search type.
 - * `graph-relaxed`: atom type, formal charge and bond order are used as matching criteria for this search type.
 - * `graph-relaxed-stereo`: atom type, formal charge, bond order, atom and bond chirality are used as matching criteria for this search type.
 - * `fingerprint-similarity`: Tanimoto similarity is used as the matching criteria

Concerning the **return_type** key, it can be one of :

- **entry**: a list of PDB IDs.
- **assembly**: list of PDB IDs appended with assembly IDs in the format of a `[pdb_id]-[assembly_id]`, corresponding to biological assemblies.
- **polymer_entity**: list of PDB IDs appended with entity IDs in the format of a `[pdb_id]_[entity_id]`, corresponding to polymeric molecular entities.
- **non_polymer_entity**: list of PDB IDs appended with entity IDs in the format of a `[pdb_id]_[entity_id]`, corresponding to non-polymeric entities (or ligands).
- **polymer_instance**: list of PDB IDs appended with asym IDs in the format of a `[pdb_id].[asym_id]`, corresponding to instances of certain polymeric molecular entities, also known as chains.

Optional arguments

There are many optional arguments. Let us see a couple of them. Pagination can be set (default is 10 entries) using the **request_options** (optional) key. Consider this query example:

```
{
  "query": {
    "type": "terminal",
    "service": "text",
    "parameters": {
      "attribute": "rcsb_polymer_entity.formula_weight",
      "operator": "greater",
      "value": 500
    }
  },
  "request_options": {
    "pager": {
      "start": 0,
      "rows": 100
    }
  },
  "return_type": "polymer_entity"
}
```

Here, the query searches for the polymer_entity that have a formula weight above 500. With request_options pager set to 100, we will get the first 100 hits.

To return all hits, set this field in the request_options:

```
"return_all_hits": true
```

Coming back at the first basic example, we can reuse it to illustrate how to refine the search using attribute and operators:

```
{
  "query": {
    "type": "terminal",
    "service": "text",
    "parameters": {
      "value": "thymidine kinase",
      "attribute": "exptl.method",
      "operator": "exact_match",
    }
  },
  "return_type": "entry"
}
```

All valid combo of operators and attributes can be found here: <http://search.rcsb.org/search-attributes.html>

For instance, in the example above only in, exact_match and exists can be used with exptl.method attribute. This is not checked in bioservices.

Sorting is determined by the sort object in the request_options context. It allows you to add one or more sorting conditions to control the order of the search result hits. The sort operation is defined on a per field level, with special field name for score to sort by score (the default)<

By default sorting is done in descending order ("desc"). The sort can be reversed by setting direction property to "asc". This example demonstrates how to sort the search results by release date:

```
{
  "query": {
    "type": "terminal",
    "service": "text",
    "parameters": {
      "attribute": "struct.title",
      "operator": "contains_phrase",
      "value": "\"hiv protease\""
    }
  },
  "request_options": {
    "sort": [
      {
        "sort_by": "rcsb_accession_info.initial_release_date",
        "direction": "desc"
      }
    ]
  },
  "return_type": "entry"
}
```

Again, many more complex examples can be found on PDB page.

Constructor

Parameters

verbose (*bool*) – prints informative messages (default is off)

get_current_ids()

Get a list of all current PDB IDs.

get_similarity_sequence(*seq*)

Search of seauence similarity search with protein sequence

seq = "VLSPADKTNVKA AWGKVG AHAGEYGA EALERMFLSFPTTKTYFPHFDL-SHGSAQVKGHGKKVADALTAVAHVDDMPNAL" results = p.get_similarity_sequence(*seq*)

search(*query*, *request_options*=None, *request_info*=None, *return_type*=None)

search request represented as a JSON object.

This is the only function in PDB API. You should be able to perform any valid PDB searches here (see the [bioservices.pdb.PDB](#) documentation for details. Note, however, that we have aliases methods in BioServices that will be added on demand for common searches.

Parameters

- **query** (*str*) – the search expression. Can be omitted if, instead of IDs retrieval, facets or count operation should be performed. In this case the request must be configured via the *request_options* context.
- **request_options** (*str*) – (optional) controls various aspects of the search request including pagination, sorting, scoring and faceting.
- **request_info** (*str*) – additional information about the query, e.g. *query_id*. (optional)
- **return_type** (*str*) – type of results to return.

Returns

json results

You must define a query as defined in the PDB web page. For example the following query search for macromolecular PDB entities that share 90% sequence identity with GTPase HRas protein from Gallus gallus (Chicken):

```
query = {
  "query": {
    "type": "terminal",
    "service": "sequence",
    "parameters": {
      "value_cutoff": 1,
      "identity_cutoff": 0.9,
      "target": "pdb_protein_sequence",
      "value":
↵ "MTEYKL VVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGETCLLDILDTAGQEEYSAMRDQYMRTGEGFLCVFAINNTKSFE
↵
    }
  },
  "request_options": {
    "scoring_strategy": "sequence"
  },
  "return_type": "polymer_entity"
}
```

What is important is that the dictionary called **query** contains 2 compulsory keys namely **query** and **return_type**. The two other optional keys are **request_options** and **return_info**

You would then call the PDB search as follows:

```
from bioservices import PDB
p = PDB()
results = p.search(query)
```

Now, in BioServices, you can also decompose the query as follows:

```
query = {
  "type": "terminal",
  "service": "sequence",
  "parameters": {
    "value_cutoff": 1,
    "identity_cutoff": 0.9,
    "target": "pdb_protein_sequence",
    "value":
↵ "MTEYKL VVVGAGGVGKSALTIQLIQNHFVDEYDPTIEDSYRKQVVIDGETCLLDILDTAGQEEYSAMRDQYMRTGEGFLCVFAINNTKSFE
↵
  }}
request_options = { "scoring_strategy": "sequence" }
return_type= "polymer_entity"
```

and then use PDB search again:

```
from bioservices import PDB
p = PDB()
```

(continues on next page)

(continued from previous page)

```
results = p.search(query, request_options=request_options, return_type=return_
↪type)
```

or even simpler for the Pythonic lovers:

```
results = p.search(**query)
```

Interface to the PDBe web Service.

What is PDBe ?

URL

<https://www.ebi.ac.uk/pdbe/>

REST

<https://www.ebi.ac.uk/pdbe/api/doc/>

PDBe is a founding member of the Worldwide Protein Data Bank which collects, organises and disseminates data on biological macromolecular structures. In collaboration with the other Worldwide Protein Data Bank (wwPDB) partners, we work to collate, maintain and provide access to the global repository of macromolecular structure models, the Protein Data Bank (PDB).

—PDBe home page, June 2020

```
class PDBe(verbose=False, cache=False)
```

Interface to part of the PDBe service

```
>>> from bioservices import PDBe
>>> s = PDBe()
>>> res = s.get_file("1FBV", "pdb")
```

Constructor

Parameters

verbose (*bool*) – prints informative messages (default is off)

get_assembly(query)

Provides information for each assembly of a given PDB ID. T

This information is broken down at the entity level for each assembly. The information given includes the molecule name, type and class, the chains where the molecule occur, and the number of copies of each entity in the assembly.

Parameters

query – a 4-character PDB id code

```
p.get_assembly('1cbs')
```

get_binding_sites(query)

Provides details on binding sites in the entry

STRUCT_SITE records in PDB files (or mmCIF equivalent thereof), such as ligand, residues in the site, description of the site, etc.

Parameters**query** – a 4-character PDB id code

```
p.get_binding_sites('1cbs')
```

get_drugbank_annotation(query)

This call provides DrugBank annotation of all ligands, i.e. ‘bound’

Parameters**query** – a 4-character PDB id code

```
p.get_drugbank_annotation('5hht')
```

get_electron_density_statistics(query)

This call details the statistics for electron density.

Parameters**query** – a 4-character PDB id code

```
p.get_electron_density_statistics('1cbs')
```

get_experiment(query)

Provides details of experiment(s) carried out in determining the structure of the entry.

Each experiment is described in a separate dictionary. For X-ray diffraction, the description consists of resolution, spacegroup, cell dimensions, R and Rfree, refinement program, etc. For NMR, details of spectrometer, sample, spectra, refinement, etc. are included. For EM, details of specimen, imaging, acquisition, reconstruction, fitting etc. are included.

Parameters**query** – a 4-character PDB id code

```
p.get_experiment('1cbs')
```

get_files(query)

Provides URLs and brief descriptions (labels) for PDB entry

Also, for mmCIF files, biological assembly files, FASTA file for sequences, SIFTS cross reference XML files, validation XML files, X-ray structure factor file, NMR experimental constraints files, etc.

Parameters**query** – a 4-character PDB id code

```
p.get_files('1cbs')
```

get_functional_annotation(query)

Provides functional annotation of all ligands, i.e. ‘bound’

Parameters**query** – a 4-character PDB id code

```
p.get_functional_annotation('1cbs')
```

get_ligand_monomers(query)

Provides a list of modelled instances of ligands,

ligands i.e. ‘bound’ molecules that are not waters.

Parameters**query** – a 4-character PDB id code

```
p.get_ligand_monomers('1cbs')
```

get_modified_residues(query)

Provides a list of modelled instances of modified amino acids or nucleotides in protein, DNA or RNA chains.

Parameters**query** – a 4-character PDB id code

```
p.get_modified_residues('4v5j')
```

get_molecules(query)

Return details of molecules (or entities in mmCIF-speak) modelled in the entry

This can be entity id, description, type, polymer-type (if applicable), number of copies in the entry, sample preparation method, source organism(s) (if applicable), etc.

Parameters**query** – a 4-character PDB id code

```
p.get_molecules('1cbs')
```

get_mutated_residues(query)

Provides a list of modelled instances of mutated amino acids or nucleotides in protein, DNA or RNA chains.

Parameters**query** – a 4-character PDB id code

```
p.get_mutated_residues('1bgj')
```

get_nmr_resources(query)

This call provides URLs of available additional resources for NMR entries. E.g., mapping between structure (PDB) and chemical shift (BMRB) entries. :param query: a 4-character PDB id code

```
p.get_nmr_resources('1cbs')
```

get_observed_ranges(query)

Provides observed ranges, i.e., segments of structural coverage of
polymeric molecules that are modelled fully or partly

Parameters**query** – a 4-character PDB id code

```
p.get_observed_ranges('1cbs')
```

get_observed_ranges_in_pdb_chain(query, chain_id)

Provides observed ranges, i.e., segments of structural coverage of
polymeric molecules in a particular chain

Parameters

- **query** – a 4-character PDB id code

- **query** – a PDB chain ID

```
p.get_observed_ranges_in_pdb_chain('1cbs', "A")
```

get_observed_residues_ratio(*query*)

Provides the ratio of observed residues for each chain in each molecule

The list of chains within an entity is sorted by observed_ratio (descending order), partial_ratio (ascending order), and number_residues (descending order).

Parameters

query – a 4-character PDB id code

```
p.get_observed_residues_ratio('1cbs')
```

get_related_dataset(*query*)

Provides DOI's for related raw experimental datasets

Includes diffraction image data, small-angle scattering data and electron micrographs.

Parameters

query – a 4-character PDB id code

```
p.get_cofactor('5o8b')
```

get_related_publications(*query*)

Return publications obtained from both EuroPMC and UniProt. T

These are articles which cite the primary citation of the entry, or open-access articles which mention the entry id without explicitly citing the primary citation of an entry.

Parameters

query – a 4-character PDB id code

```
p.get_related_publications('1cbs')
```

get_release_status(*query*)

Provides status of a PDB entry (released, obsoleted, on-hold etc) along with some other information such as authors, title, experimental method, etc.

Parameters

query – a 4-character PDB id code

```
p.get_release_status('1cbs')
```

get_residue_listing(*query*)

Provides lists all residues (modelled or otherwise) in the entry.

Except waters, along with details of the fraction of expected atoms modelled for the residue and any alternate conformers.

Parameters

query – a 4-character PDB id code

```
p.get_residue_listing('1cbs')
```

get_residue_listing_in_pdb_chain(*query*, *chain_id*)

Provides all residues (modelled or otherwise) in the entry

Except waters, along with details of the fraction of expected atoms modelled for the residue and any alternate conformers.

Parameters

- **query** – a 4-character PDB id code
- **query** – a PDB chain ID

```
p.get_residue_listing_in_pdb_chain('1cbs')
```

get_secondary_structure(*query*)

Provides residue ranges of regular secondary structure

(alpha helices and beta strands) found in protein chains of the entry. For strands, sheet id can be used to identify a beta sheet.

Parameters

query – a 4-character PDB id code

```
p.get_secondary_structure('1cbs')
```

get_summary(*query*)

Returns summary of a PDB entry

This can be title of the entry, list of depositors, date of deposition, date of release, date of latest revision, experimental method, list of related entries in case split entries, etc.

Parameters

query – a 4-character PDB id code

```
p.get_summary('1cbs')
p.get_summary('1cbs', '2kv8')
p.get_summary(['1cbs', '2kv8'])
```

PRIDE module

Interface to PRIDE web service

What is PRIDE ?**URL**

<http://www.ebi.ac.uk/pride/archive/>

URL

<http://www.ebi.ac.uk/pride/ws/archive>

The PRIDE PRoteomics IDentifications database is a centralized, standards compliant, public data repository for proteomics data, including protein and peptide identifications, post-translational modifications and supporting spectral evidence.

—From PRIDE web site, Jan 2015

class PRIDE(*verbose=False, cache=False*)

Interface to the [PRIDE](#) service

Constructor

Parameters

verbose – set to False to prevent informative messages

get_assay_count(*identifier*)

Count assays for a project accession number

Parameters

identifier (*str*) – a project accession number

Returns

integer

```
>>> p = PRIDE()
>>> assays = p.get_assay_count('PRD000001')
5
```

get_assay_list(*identifier*)

Return list of assays for a project accession number

Parameters

identifier (*str*) – project accession number. See [get_project_list\(\)](#)

Returns

list of dictionaries. Each dictionary represents an assay.

```
>>> p = PRIDE()
>>> assays = p.get_assay_list('PRD000001')
>>> len(assays) # could be found with get_assay_count_project_accession
5
>>> assays[1]['assayAccession']
1643
```

get_assays(*identifier*)

Retrieve assay information by assay accession

Parameters

identifier (*int*) – assay accession number

```
>>> p = PRIDE()
>>> res = p.get_assays(1643)
>>> res['proteinCount']
276
```

get_file_count(*identifier*)

return count of files in a project

Parameters

identifier (*str*) – a project accession number

Returns

int

```
>>> p.get_file_count('PRD000001')
5
```

get_file_count_assay(*identifier*)

list files for an assay

Parameters

identifier (*int*) – assay accession number

Returns

int

```
p.get_file_assay(1643)
```

get_file_list(*identifier*)

return list of files for a project

Parameters

identifier (*str*) – a project accession number

```
>>> files = p.get_file_count('PRD000001')
>>> len(files)
5
```

get_file_list_assay(*identifier*)

list files for an assay

Parameters

identifier (*int*) – assay accession number

Returns

list of dictionary, Each dictionary represents a file data structure

```
res = p.get_file_assay(1643)
```

get_peptide_count(*identifier*, *sequence=None*)

Count peptide identifications by project accession

Parameters

identifier (*str*) – a project accession number

Returns

int

```
>>> p.get_peptide_count('PRD000001', sequence='PLIPIVVEQTGR')
4
>>> p.get_peptide_count('PRD000001')
6758
```

get_peptide_count_assay(*identifier*, *sequence=None*)

Count peptide identifications by assay accession

Parameters

identifier (*str*) – an assay accession number

Returns

int

```
>>> p.get_peptide_count_assay(1643, sequence='AAATQKKVER')
5
>>> p.get_peptide_count_assay(1643)
1696
```

get_peptide_list(*identifier*, *sequence=None*, *show=10*, *page=0*)

Retrieve peptide identifications by project accession (and sequence)

Parameters

- **identifier** (*str*) – a project accession number
- **sequence** (*str*) – the peptide sequence to limit the query on (optional). If provided, *show* and *page* are not used
- **show** (*int*) – how many results to return per page
- **page** (*int*) – which page (starting from 0) of the result to return

```
>>> peptides = p.get_peptide_list('PRD000001', sequence='PLIPIVVEQTGR')
>>> len(peptides)
4
>>> peptides = p.get_peptide_list('PRD000001')
>>> len(peptides)
10
>>> peptides = p.get_peptide_list('PRD000001', show=100)
```

Note: the function merge two functions from the PRIDE API (*get_peptide_list* and *get_peptide_list_sequence*)

get_peptide_list_assay(*identifier*, *sequence=None*, *show=10*, *page=0*)

Retrieve peptide identifications by assay accession (and sequence)

Parameters

- **identifier** (*str*) – an assay accession number
- **sequence** (*str*) – the peptide sequence to limit the query on (optional). If provided, *show* and *page* are not used
- **show** (*int*) – how many results to return per page
- **page** (*int*) – which page (starting from 0) of the result to return

```
>>> peptides = p.get_peptide_list_assay(1643, sequence='AAATQKKVER')
>>> len(peptides)
5
>>> peptides = p.get_peptide_list_assay(1643)
>>> len(peptides)
10
>>> peptides = p.get_peptide_list_assay(1643, show=100)
```

Note: the function merge two functions from the PRIDE API (*get_peptide_list* and *get_peptide_list_sequence*)

get_project(*identifier*)

Retrieve project information by accession

Parameters

identifier (*str*) – a valid PRIDE identifier e.g., PRD000001

Returns

a dictionary with the project details. See <http://www.ebi.ac.uk/pride/ws/archive/#!/project> for details

```
>>> from bioservices import PRIDE
>>> p = PRIDE()
>>> res = p.get_project("PRD000001")
>>> res['numPeptides']
6758
```

get_project_count(*query=""*, *speciesFilter=None*, *ptmsFilter=None*, *tissueFilter=None*, *diseaseFilter=None*, *titleFilter=None*, *instrumentFilter=None*, *experimentTypeFilter=None*, *quantificationfilter=None*, *projectTagFilter=None*)

Count projects for given criteria

Takes same query parameters as the /list operation; typically used to retrieve number of results before querying with /list

Parameters

- **query** (*str*) – search term to query for
- **speciesFilter** (*str*) – filter by species (NCBI taxon ID or name)
- **ptmsFilter** (*str*) – filter by PTM annotation query
- **tissueFilter** (*str*) – filter by tissue annotation
- **diseaseFilter** (*str*) – filter by disease annotation
- **titleFilter** (*str*) – filter the title for keywords
- **instrumentFilter** (*str*) – filter for instrument names or keywords
- **experimentTypeFilter** (*str*) – filter by experiment type
- **quantificationFilter** (*str*) – filter by quantification annotation
- **projectTagFilter** (*str*) – filter by project tags

Returns

number of projects (integer)

get_project_list(*query=""*, *show=10*, *page=0*, *sort=None*, *order='desc'*, *speciesFilter=None*, *ptmsFilter=None*, *tissueFilter=None*, *diseaseFilter=None*, *titleFilter=None*, *instrumentFilter=None*, *experimentTypeFilter=None*, *quantificationfilter=None*, *projectTagFilter=None*)

list projects or given criteria

Parameters

- **query** (*str*) – search term to query for
- **show** (*int*) – how many results to return per page
- **page** (*int*) – which page (starting from 0) of the result to return
- **sort** (*str*) – the field to sort on

- **order** (*str*) – the sorting order (asc or desc)
- **speciesFilter** (*str*) – filter by species (NCBI taxon ID or name)
- **ptmsFilter** (*str*) – filter by PTM annotation query
- **tissueFilter** (*str*) – filter by tissue annotation
- **diseaseFilter** (*str*) – filter by disease annotation
- **titleFilter** (*str*) – filter the title for keywords
- **instrumentFilter** (*str*) – filter for instrument names or keywords
- **experimentTypeFilter** (*str*) – filter by experiment type
- **quantificationFilter** (*str*) – filter by quantification annotation
- **projectTagFilter** (*str*) – filter by project tags

```
>>> p = PRIDE()
>>> projects = p.get_project_list(show=100)
```

get_protein_count(*identifier*)

Count protein identifications by project accession

Parameters

identifier (*str*) – a project accession number

Returns

int

get_protein_count_assay(*identifier*)

Count protein identifications by assay accession

Parameters

identifier (*str*) – a project accession number

Returns

int

get_protein_list(*identifier*, *show=10*, *page=0*)

Retrieve protein identifications by project accession

Parameters

- **identifier** (*str*) – a project accession number
- **show** (*int*) – how many results to return per page
- **page** (*int*) – which page (starting from 0) of the result to return

get_protein_list_assay(*identifier*, *show=10*, *page=0*)

Retrieve protein identifications by assay accession

Parameters

- **identifier** (*str*) – a project accession number
- **show** (*int*) – how many results to return per page
- **page** (*int*) – which page (starting from 0) of the result to return

PSICQUIC

Interface to the PSICQUIC web service

What is PSICQUIC ?

URL

<http://code.google.com/p/psicquic/>

REST

http://code.google.com/p/psicquic/wiki/PsicquicSpec_1_3_Rest

“PSICQUIC is an effort from the HUPO Proteomics Standard Initiative (HUPO-PSI) to standardise the access to molecular interaction databases programmatically. The PSICQUIC View web interface shows that PSICQUIC provides access to 25 active service “

—Dec 2012

About queries

source: PSICQUIC View web page

The idea behind PSICQUIC is to retrieve information related to protein interactions from various databases. Note that protein interactions does not necessarily mean protein-protein interactions. In order to be effective, the query format has been standardised.

To do a search you can use the Molecular Interaction Query Language which is based on Lucene's syntax. Here are some rules

- Use OR or space ' ' to search for ANY of the terms in a field
- Use AND if you want to search for those interactions where ALL of your terms are found
- Use quotes (") if you look for a specific phrase (group of terms that must be searched together) or terms containing special characters that may otherwise be interpreted by our query engine (eg. ':' in a GO term)
- Use parenthesis for complex queries (e.g. '(XXX OR YYY) AND ZZZ')
- **Wildcards (*,?) can be used between letters in a term or at the end of terms to do fuzzy queries,** but never at the beginning of a term.
- **Optionally, you can prepend a symbol in front of your term.**
 - * (plus): include this term. Equivalent to AND. e.g. +P12345
 - * (minus): do not include this term. Equivalent to NOT. e.g. -P12345
 - Nothing in front of the term. Equivalent to OR. e.g. P12345
- Implicit fields are used when no field is specified (simple search). For instance, if you put 'P12345' in the simple query box, this will mean the same as identifier:P12345 OR pubid:P12345 OR pubauth:P12345 OR species:P12345 OR type:P12345 OR detmethod:P12345 OR interaction_id:P12345

About the MITAB output

The output returned by a query contains a list of entries. Each entry is formatted following the MITAB output.

Here below are listed the name of the field returned ordered as they would appear in one entry. The first item is always idA whatever version of MITAB is used. The version 25 of MITAB contains the first 15 fields in the table below. Newer version may include more fields but always include the 15 from MITAB 25 in the same order. See the link from [irefindex about mitab](#) for more information.

| Field Name | Searches on | Implicit* | Example |
|----------------|---|-----------|-------------------------------|
| idA | Identifier A | No | idA:P74565 |
| idB | Identifier B | No | idB:P74565 |
| id | Identifiers (A or B) | No | id:P74565 |
| alias | Aliases (A or B) | No | alias:(KHDRBS1 HCK) |
| identifiers | Identifiers and Aliases undistinctively | Yes | identifier:P74565 |
| pubauth | Publication 1st author(s) | Yes | pubauth:scott |
| pubid | Publication Identifier(s) OR | Yes | pubid:(10837477 12029088) |
| taxidA | Tax ID interactor A: the tax ID or the species name | No | taxidA:mouse |
| taxidB | Tax ID interactor B: the tax ID or species name | No | taxidB:9606 |
| species | Species. Tax ID A or Tax ID B | Yes | species:human |
| type | Interaction type(s) | Yes | type:"physical interaction" |
| detmethod | Interaction Detection method(s) | Yes | detmethod:"two hybrid*" |
| interaction_id | Interaction identifier(s) | Yes | interaction_id:EBI-761050 |
| pbioroleA | Biological role A | Yes | pbioroleA:ancillary |
| pbioroleB | Biological role B | Yes | pbioroleB:"MI:0684" |
| pbiorole | Biological roles (A or B) | Yes | pbiorole:enzyme |
| ptypeA | Interactor type A | Yes | ptypeA:protein |
| ptypeB | Interactor type B | Yes | ptypeB:"gene" |
| ptype | Interactor types (A or B) | Yes | pbiorole:"small molecule" |
| pxrefA | Interactor xref A (or Identifier A) | Yes | pxrefA:"GO:0003824" |
| pxrefB | Interactor xref B (or Identifier B) | | Yes pxrefB:"GO:0003824" |
| pxref | Interactor xrefs (A or B or Identifier A or Identifier B) | Yes | pxref:"catalytic activity" |
| xref | Interaction xrefs (or Interaction identifiers) | Yes | xref:"nuclear pore" |
| annot | Interaction annotations and tags | Yes | annot:"internally curated" |
| update | Update date | Yes | update:[20100101 TO 20120101] |

continues on next page

Table 1 – continued from previous page

| Field Name | Searches on | Implicit* | Example |
|------------|--|-----------|---|
| negative | Negative interaction boolean | Yes | negative:true |
| complex | Complex expansion | Yes | complex:"spoke expanded" |
| ftypeA | Feature type of participant A | Yes | ftypeA:"sufficient to bind" |
| ftypeB | Feature type of participant B | Yes | ftypeB:mutation |
| ftype | Feature type of participant A or B | Yes | ftype:"binding site" |
| pmethodA | Participant identification method A | Yes | pmethodA:"western blot" |
| pmethodB | Participant identification method B | Yes | pmethodB:"sequence tag identification" |
| pmethod | Participant identification methods (A or B) | Yes | pmethod:immunostaining |
| stc | Stoichiometry (A or B). Only true or false, just to be able to filter interac- tion having stoichiometry available | Yes | stc:true |
| param | Interaction parameters. Only true or false, just to be able to filter interac- tion having parameters available | Yes | param:true |

class PSICQUIC(verbose=True)

Interface to the PSICQUIC service

There are 2 interfaces to the PSICQUIC service (REST and WSDL) but we used the REST only.

This service provides a common interface to more than 25 other services related to protein. So, we won't detail all the possibility of this service. Here is an example that consists of looking for interactors of the protein ZAP70 within the IntAct database:

```
>>> from bioservices import *
>>> s = PSICQUIC()
>>> res = s.query("intact", "zap70")
>>> len(res) # there are 11 interactions found
11
>>> for x in res[1]:
...     print(x)
uniprotkb:095169
uniprotkb:P43403
intact:EBI-716238
intact:EBI-1211276
psi-mi:ndub8_human(display_long)|uniprotkb:NADH-ubiquinone oxidoreductase ASHI
.
```

(continues on next page)

(continued from previous page)

.

Here we have a list of entries. There are 15 of them (depending on the *output* parameter). The meaning of the entries is described on PSICQUIC website: <https://code.google.com/p/psicquic/wiki/MITAB25Format> . In short:

1. Unique identifier for interactor A
2. Unique identifier for interactor B.
3. Alternative identifier for interactor A, for example the official gene
4. Alternative identifier for interactor B.
5. Aliases for A, separated by “|”
6. Aliases for B.
7. Interaction detection methods, taken from the corresponding PSI-MI
8. First author surname(s) of the publication(s)
9. Identifier of the publication
10. NCBI Taxonomy identifier for interactor A.
11. NCBI Taxonomy identifier for interactor B.
12. Interaction types,
13. Source databases and identifiers,
14. Interaction identifier(s) i
15. Confidence score. Denoted as scoreType:value.

Another example with reactome database:

```
res = s.query("reactome", "Q9Y266")
```

Warning: PSICQUIC gives access to 25 other services. We cannot create a dedicated parsing for all of them. So, the `::query` method returns the raw data. Addition class may provide dedicated parsing in the future.

See also:

bioservices.biogrid.BioGRID

Constructor

Parameters

verbose (*bool*) – print informative messages

```
>>> from bioservices import PSICQUIC
>>> s = PSICQUIC()
```

property activeDBs

returns the active DBs only

convert(*data*, *db=None*)

convertAll(*data*)

property formats

Returns the possible output formats

getInteractionCounter(*query*)

Returns a dictionary with database as key and results as values

Parameters

query (*str*) – a valid query

Returns

a dictionary which key as database and value as number of entries

Consider only the active database.

getName(*data*)

knownName(*data*)

Scan all entries (MITAB) and returns simplified version

Each item in the input list of mitab entry The output is made of 2 lists corresponding to interactor A and B found in the mitab entries.

elements in the input list takes the following forms:

| |
|------------------------------------|
| DB1 : ID1 DB2 : ID2 DB3 : ID3 |
|------------------------------------|

The | sign separates equivalent IDs from different databases.

We want to keep only one. The first known database is kept. If in the list of DB:ID pairs no known database is found, then we keep the first one whatsoever.

known databases are those available in the uniprot mapping tools.

chembl and chebi IDs are kept unchanged.

mappingOneDB(*data*)

postCleaning(*data*, *keep_only='HUMAN'*, *remove_db=['chebi', 'chembl']*, *keep_self_loop=False*, *verbose=True*)

Remove entries with a None and keep only those with the keep pattern

postCleaningAll(*data*, *keep_only='HUMAN'*, *flatten=True*, *verbose=True*)

even more cleaning by ignoring score, db and interaction len(set([(x[0],x[1]) for x in retnew]))

preCleaning(*data*)

remove entries where IdA or IdB is set to “-”

print_status()

Prints the services that are available

Returns

Nothing

The output is tabulated. The columns are:

- names
- active

- count
- version
- rest URL
- soap URL
- rest example
- restricted

See also:

If you want the data into lists, see all attributes starting with `registry` such as `registry_names()`

query(*service, query, output='tab25', version='current', firstResult=None, maxResults=None*)

Send a query to a specific database

Parameters

- **service** (*str*) – a registered service. See `registry_names`.
- **query** (*str*) – a valid query. Can be * or a protein name.
- **output** (*str*) – a valid format. See `s._formats`

```
s.query("intact", "brca2", "tab27")
s.query("intact", "zap70", "xml25")
s.query("matrixdb", "*", "xml25")
```

This is the programmatic approach to this website:

<http://www.ebi.ac.uk/Tools/webservices/psicquic/view/main.xhtml>

Another example consist in accessing the *string* database for fetching protein-protein interaction data of a particular model organism. Here we restrict the query to 100 results:

```
s.query("string", "species:10090", firstResult=0, maxResults=100, output="tab25")
```

spaces are automatically converted

```
s.query("biogrid", "ZAP70 AND species:9606")
```

Warning: AND must be in big caps. Some database are ore permissive than other (e.g., intact accepts "and"). species must be a valid ID number. Again, some DB are more permissive and may accept the name (e.g., human)

To obtain the number of interactions in intact for the human specy:

```
>>> len(p.query("intact", "species:9606"))
```

queryAll(*query, databases=None, output='tab25', version='current', firstResult=None, maxResults=None*)

Same as `query` but runs on all active database

Parameters

databases (*list*) – database to query. Queries all active DB if not provided

Returns

dictionary where keys correspond to databases and values to the output of the query.

```
res = s.queryAll("ZAP70 AND species:9606")
```

read_registry()

Reads and returns the active registry

property registry

returns the registry of psiquic

property registry_actives

returns active state of each service

property registry_counts

returns number of entries in each service

property registry_names

returns all services available (names)

property registry_restexamples

returns REST example for each service

property registry_restricted

returns restricted status of services

property registry_resturls

returns URL of REST services

property registry_soapurls

returns URL of WSDL service

property registry_versions

returns version of each service

Rhea

Interface to the Rhea web services

What is Rhea ?**URL**

<http://www.ebi.ac.uk/rhea/>

Citations

See <http://www.ebi.ac.uk/rhea/about.xhtml>

Rhea is a reaction database, where all reaction participants (reactants and products) are linked to the ChEBI database (Chemical Entities of Biological Interest) which provides detailed information about structure, formula and charge. Rhea provides built-in validations that ensure both elemental and charge balance of the reactions... While the main focus of Rhea is enzyme-catalysed reactions, other biochemical reactions are also included.

The database is extensively cross-referenced. Reactions are currently linked to the EC list, KEGG and MetaCyc, and the reactions will be used in the IntEnz database and in all relevant UniProtKB entries. Furthermore, the reactions will also be used in the UniPathway database to generate pathways and metabolic networks.

—from Rhea Home page, Dec 2012 (<http://www.ebi.ac.uk/rhea/about.xhtml>)


```
class Rhea(verbose=True, cache=False)
```

Interface to the [Rhea](#) service

You can search by compound name, ChEBI ID, reaction ID, cross reference (e.g., EC number) or citation (author name, title, abstract text, publication ID). You can use double quotes - to match an exact phrase - and the following wildcards:

- ? (question mark = one character),
- * (asterisk = several characters).

Searching for `caffe*` will find reactions with participants such as caffeine, trans-caffeic acid or caffeoyl-CoA:

```
from bioservices import Rhea
r = Rhea()
response = r.search("caffe*")
```

Searching for `a?e?o*` will find reactions with participants such as acetoin, acetone or adenosine.:

```
from bioservices import Rhea
r = Rhea()
response = r.search("a?e?o*")
```

The `search()` entry() methods require a list of valid columns. By default all columns are used but you can restrict to only a few. Here is the description of the columns:

```
rhea-id : reaction identifier (with prefix RHEA)
equation : textual description of the reaction equation
chebi : comma-separated list of ChEBI names used as reaction participants
chebi-id : comma-separated list of ChEBI identifiers used as reaction participants
ec : comma-separated list of EC numbers (with prefix EC)
uniprot : number of proteins (UniProtKB entries) annotated with the Rhea reaction
pubmed : comma-separated list of PubMed identifiers (without prefix)
```

and 5 cross-references:

```
reaction-xref(EcoCyc)
reaction-xref(MetaCyc)
reaction-xref(KEGG)
reaction-xref(Reactome)
reaction-xref(M-CSA)
```

Rhea constructor

Parameters

verbose (*bool*) – True by default

```
>>> from bioservices import Rhea
>>> r = Rhea()
```

`get_metabolites(rxn_id)`

Given a Rhea (<http://www.rhea-db.org/>) reaction id, returns its participant metabolites as a dict: {metabolite: stoichiometry},

e.g. '2 H + 1 O2 = 1 H2O' would be represented as { 'H': -2, 'O2': -1, 'H2O': 1 }.

Parameters

rxn_id – Rhea reaction id

Returns

dict of participant metabolites.

query(*query*, *columns=None*, *frmt='tsv'*, *limit=None*)

Retrieve a concrete reaction for the given id in a given format

Parameters

- **query** (*str*) – the entry to retrieve
- **frmt** (*str*) – the result format (tsv); only tsv accepted for now (Nov 2020).
- **limit** (*int*) – maximum number of results to retrieve

Returns

dataframe

Retrieve Rhea reaction identifiers and equation text:

```
r.query("", columns="rhea-id,equation", limit=10)
```

Retrieve Rhea reactions with enzymes curated in UniProtKB (only first 10 entries):

```
r.query("uniprot:*", columns="rhea-id,equation", limit=10)
```

To retrieve a specific entry:

```
df = r.get_entry("rhea:10661")
```

Changed in version 1.8.0: (*entry()* method renamed in *query()* and no more format required. Must be given in the entry name e.g. *query*("10281.rxn") instead of *entry*(10281, *format*="rxn") the option *frmt* is now related to the result format

search(*query*, *columns=None*, *limit=None*, *frmt='tsv'*)

Search for Rhea (mimics <https://www.rhea-db.org/>)

Parameters

- **query** (*str*) – the search term using format parameter
- **format** (*str*) – the biopax2 or cmlreact format (default)

Returns

A pandas DataFrame.

```
>>> r = Rhea()
>>> df = r.search("caffeine")
>>> df = r.search("caffeine", columns='rhea-id,equation')
```

Reactome

Interface to the Reactome webs services

What is Reactome?

URL

<http://www.reactome.org/ReactomeGWT/entrypoint.html>

Citation

<http://www.reactome.org/citation.html>

REST

<http://reactomews.oicr.on.ca:8080/ReactomeRESTfulAPI/RESTfulWS>

“REACTOME is an open-source, open access, manually curated and peer-reviewed pathway database. Pathway annotations are authored by expert biologists, in collaboration with Reactome editorial staff and cross-referenced to many bioinformatics databases. These include NCBI Entrez Gene, Ensembl and UniProt databases, the UCSC and HapMap Genome Browsers, the KEGG Compound and ChEBI small molecule databases, PubMed, and Gene Ontology. ... “

—from Reactome web site

class Reactome(*verbose=True, cache=False*)

Todo: interactors, orthology, particiapnts, person, query, refernces, schema

get_complex_subunits(*identifier, excludeStructuresSpecifies=False*)

A list with the entities contained in a given complex

Retrieves the list of subunits that constitute any given complex. In case the complex comprises other complexes, this method recursively traverses the content returning each contained PhysicalEntity. Contained complexes and entity sets can be excluded setting the ‘excludeStructures’ optional parameter to ‘true’

Parameters

- **identifier** – The complex for which subunits are requested
- **excludeStructures** – Specifies whether contained complexes and entity sets are excluded in the response

```
r.get_complex_subunits("R-HSA-5674003")
```

get_complexes(*resources, identifier*)

A list of complexes containing the pair (identifier, resource)

Retrieves the list of complexes that contain a given (identifier, resource). The method deconstructs the complexes into all its participants to do so.

Parameters

- **resource** – The resource of the identifier for complexes are requested (e.g. UniProt)
- **identifier** – The identifier for which complexes are requested

```
r.get_complexes(resources, identifier)
r.get_complexes("UniProt", "P43403")
```

get_discover(*identifier*)

The schema.org for an Event in Reactome knowledgebase

For each event (reaction or pathway) this method generates a json file representing the dataset object as defined by schema.org (http). This is mainly used by search engines in order to index the data

```
r.data_discover("R-HSA-446203")
```

get_diseases()

list of diseases objects

get_diseases_doid()

retrieves the list of disease DOIDs annotated in Reactome

return: dictionary with DOID contained in the values()

get_entity_componentOf(*identifier*)

A list of larger structures containing the entity

Retrieves the list of structures (Complexes and Sets) that include the given entity as their component. It should be mentioned that the list includes only simplified entries (type, names, ids) and not full information about each item.

```
r.get_entity_componentOf("R-HSA-199420")
```

get_entity_otherForms(*identifier*)

All other forms of PhysicalEntity

Retrieves a list containing all other forms of the given PhysicalEntity. These other forms are PhysicalEntities that share the same ReferenceEntity identifier, e.g. PTEN H93R[R-HSA-2318524] and PTEN C124R[R-HSA-2317439] are two forms of PTEN.

```
r.get_entity_otherForms("R-HSA-199420")
```

get_event_ancestors(*identifier*)

The ancestors of a given event

The Reactome definition of events includes pathways and reactions. Although events are organised in a hierarchical structure, a single event can be in more than one location, i.e. a reaction can take part in different pathways while, in the same way, a sub-pathway can take part in many pathways. Therefore, this method retrieves a list of all possible paths from the requested event to the top level pathway(s).

Parameters

identifier – The event for which the ancestors are requested

```
r.get_event_ancestors("R-HSA-5673001")
```

get_eventsHierarchy(*species*)

The full event hierarchy for a given species

Events (pathways and reactions) in Reactome are organised in a hierarchical structure for every species. By following all 'hasEvent' relationships, this method retrieves the full event hierarchy for any given species. The result is a list of tree structures, one for each TopLevelPathway. Every event in these trees is represented by a PathwayBrowserNode. The latter contains the stable identifier, the name, the species, the url, the type, and the diagram of the particular event.

Parameters

species – Allowed species filter: SpeciesName (eg: Homo sapiens) SpeciesTaxId (eg: 9606)

```
r.get_eventsHierarchy(9606)
```

get_exporter_diagram(*identifier*, *ext*='png', *quality*=5, *diagramProfile*='Modern',
analysisProfile='Standard', *filename*=None)

Export a given pathway diagram to raster file

This method accepts identifiers for Event class instances. When a diagrammed pathway is provided, the diagram is exported to the specified format. When a subpathway is provided, the diagram for the parent is exported and the events that are part of the subpathways are selected. When a reaction is provided, the diagram containing the reaction is exported and the reaction is selected.

Parameters

- **identifier** – Event identifier (it can be a pathway with diagram, a subpathway or a reaction)
- **ext** – File extension (defines the image format) in png, jpeg, jpg, svg, gif
- **quality** – Result image quality between [1 - 10]. It defines the quality of the final image (Default 5)
- **flg** – not implemented
- **sel** – not implemented
- **diagramProfile** – Diagram Color Profile
- **token** – not implemented
- **analysisProfile** – Analysis Color Profile
- **expColumn** – not implemented
- **filename** – if given, save the results in the provided filename

return: raw data if filename parameter is not set. Otherwise, the data is saved in the filename and the function returns None

get_exporter_fireworks()

get_exporter_reaction()

get_exporter_sbml(*identifier*)

Export given Pathway to SBML

Parameters

- identifier** – DbId or StId of the requested database object

```
r.exporter_sbml("R-HSA-68616")
```

get_interactors_psicquic_molecule_details()

Retrieve clustered interaction, sorted by score, of a given accession by resource.

get_interactors_psicquic_molecule_summary()

Retrieve a summary of a given accession by resource

get_interactors_psicquic_resources()

Retrieve a list of all Psicquic Registries services

get_interactors_static_molecule_details()

Retrieve a detailed interaction information of a given accession

get_interactors_static_molecule_pathways()

Retrieve a list of lower level pathways where the interacting molecules can be found

get_interactors_static_molecule_summary()

Retrieve a summary of a given accession

get_mapping_identifier_pathways(resource, identifier)

get_mapping_identifier_reactions(resource, identifier)

get_pathway_containedEvents(identifier)

All the events contained in the given event

Events are the building blocks used in Reactome to represent all biological processes, and they include pathways and reactions. Typically, an event can contain other events. For example, a pathway can contain smaller pathways and reactions. This method recursively retrieves all the events contained in any given event.

```
res = r.get_pathway_containedEvents("R-HSA-5673001")
```

get_pathway_containedEvents_by_attribute(identifier, attribute)

A single property for each event contained in the given event

Events are the building blocks used in Reactome to represent all biological processes, and they include pathways and reactions. Typically, an event can contain other events. For example, a pathway can contain smaller pathways (subpathways) and reactions. This method recursively retrieves a single attribute for each of the events contained in the given event.

Parameters

- **identifier** – The event for which the contained events are requested
- **attribute** – Attribute to be filtered

```
r.get_pathway_containedEvents_by_attribute("R-HSA-5673001", "stId")
```

get_pathways_low_diagram_entity(identifier)

A list of lower level pathways with diagram containing a given entity or event

This method traverses the event hierarchy and retrieves the list of all lower level pathways that have a diagram and contain the given PhysicalEntity or Event.

Parameters

- **identifier** – The entity that has to be present in the pathways
- **species** – The species for which the pathways are requested. Taxonomy identifier (eg: 9606) or species name (eg: 'Homo sapiens')

```
r.get_pathways_low_diagram_entity("R-HSA-199420")
```

get_pathways_low_diagram_entity_allForms(identifier)

```
r.get_pathways_low_diagram_entity_allForms("R-HSA-199420")
```

get_pathways_low_entity(*identifier*)

A list of lower level pathways containing a given entity or event

This method traverses the event hierarchy and retrieves the list of all lower level pathways that contain the given PhysicalEntity or Event.

```
r.get_pathways_low_entity("R-HSA-199420")
```

get_pathways_low_entity_allForms(*identifier*)

A list of lower level pathways containing any form of a given entity

This method traverses the event hierarchy and retrieves the list of all lower level pathways that contain the given PhysicalEntity in any of its variant forms. These variant forms include for example different post-translationally modified versions of a single protein, or the same chemical in different compartments.

```
r.get_pathways_low_entity_allForms("R-HSA-199420")
```

get_pathways_top(*species*)**get_references**(*identifier*)

All referenceEntities for a given identifier

Retrieves a list containing all the reference entities for a given identifier.

```
r.get_references(15377)
```

get_species_all()

the list of all species in Reactome

get_species_main()

the list of main species in Reactome

```
r.get_species_main()
```

property name**search_facet**()

A list of facets corresponding to the whole Reactome search data

This method retrieves faceting information on the whole Reactome search data.

search_facet_query(*query*)

A list of facets corresponding to a specific query

This method retrieves faceting information on a specific query

search_query(*query*)

Queries Solr against the Reactome knowledgebase

This method performs a Solr query on the Reactome knowledgebase. Results can be provided in a paginated format.

search_spellcheck(*query*)

Spell-check suggestions for a given query

This method retrieves a list of spell-check suggestions for a given search term.

search_suggest(query)

Autosuggestions for a given query

This method retrieves a list of suggestions for a given search term.

```
>>> r.http_get("search/suggest?query=apopt")
['apoptosis', 'apoptosome', 'apoptosome-mediated', 'apoptotic']
```

property version

class ReactomeAnalysis(verbose=True, cache=False)

Constructor

Parameters

- **name** (*str*) – a name for this service
- **url** (*str*) – its URL
- **verbose** (*bool*) – prints informative messages if True (default is True)
- **requests_per_sec** – maximum number of requests per seconds are restricted to 3. You can change that value. If you reach the limit, an error is raise. The reason for this limitation is that some services (e.g., NCBI) may black list you IP. If you need or can do more (e.g., ChEMBL does not seem to have restrictions), change the value. You can also have several instance but again, if you send too many requests at the same, your future requests may be retracted. Currently implemented for REST only

All instances have an attribute called **logging** that is an instance of the **logging** module. It can be used to print information, warning, error messages:

```
self.logging.info("informative message")
self.logging.warning("warning message")
self.logging.error("error message")
```

The attribute **debugLevel** can be used to set the behaviour of the logging messages. If the argument **verbose** is True, the **debugLevel** is set to INFO. If **verbose** is False, the **debugLevel** is set to WARNING. However, you can use the **debugLevel** attribute to change it to one of DEBUG, INFO, WARNING, ERROR, CRITICAL. **debugLevel=WARNING** means that only WARNING, ERROR and CRITICAL messages are shown.

identifiers(genes)

s.identifiers("TP53") .. warning:: works for oe gene only for now

class ReactomeOld(verbose=True, cache=False)

Reactome interface

some data can be download on the main [website](#)

Constructor

Parameters

- **name** (*str*) – a name for this service
- **url** (*str*) – its URL
- **verbose** (*bool*) – prints informative messages if True (default is True)
- **requests_per_sec** – maximum number of requests per seconds are restricted to 3. You can change that value. If you reach the limit, an error is raise. The reason for this limitation is that some services (e.g., NCBI) may black list you IP. If you need or can do more (e.g., ChEMBL does not seem to have restrictions), change the value. You can also have several instance but again, if you send too many requests at the same, your future requests may be retracted. Currently implemented for REST only

All instances have an attribute called `logging` that is an instance of the `logging` module. It can be used to print information, warning, error messages:

```
self.logging.info("informative message")
self.logging.warning("warning message")
self.logging.error("error message")
```

The attribute `debugLevel` can be used to set the behaviour of the logging messages. If the argument `verbose` is True, the `debugLevel` is set to INFO. If `verbose` is False, the `debugLevel` is set to WARNING. However, you can use the `debugLevel` attribute to change it to one of DEBUG, INFO, WARNING, ERROR, CRITICAL. `debugLevel=WARNING` means that only WARNING, ERROR and CRITICAL messages are shown.

SBML_exporter(*identifier*)

Get the SBML XML text of a pathway identifier

Parameters

identifier (*int*) – Pathway database identifier

Returns

SBML object in XML format as a string

```
>>> from bioservices import Reactome
>>> s = Reactome()
>>> xml = s.SBML_exporter(109581)
```

biopax_exporter(*identifier, level=2*)

Get BioPAX file

The passed identifier has to be a valid event identifier. If there is no matching ID in the database, it will return an empty string.

Parameters

- **level** (*int*) – BioPAX level: one of two values: 2 or 3
- **identifier** (*int*) – event database identifier

Returns

BioPAX RDF document

```
>>> # for Apoptosis:
>>> s = Reactome()
>>> res = s.biopax_exporter(109581)
```

bioservices_get_reactants_from_reaction_identifier(*reaction*)

Fetch information from the reaction HTML page

Note: draft version

front_page_items(*species*)

Get list of front page items listed in the Reactome Pathway Browser

Parameters

species (*str*) – Full species name that should be encoded for URL (e.g. homo+sapiens for human, or mus+musculus for mouse) + can be replaced by spaces.

Returns

list of fully encoded Pathway objects in JSON

```
>>> s = Reactome()
>>> res = s.front_page_items("homo sapiens")
>>> print(res[0]['name'])
['Apoptosis']
```

See also:

[Pathway Browser](#)

get_all_reactions()

Return list of reactions from the Pathway

get_list_pathways()

Return list of pathways from reactome website

Returns

list of lists. Each sub-lis contains 3 items: reactome pathway identifier, description and species

get_species()

Return list of species from all pathways

highlight_pathway_diagram(*identifier, genes, frmt='PNG'*)

Highlight a diagram for a specified pathway based on its identifier

Parameters

- **identifier** (*int*) – a valid pathway identifier
- **genes** (*list*) – a list of string to indicate the genes to highlight
- **frmt** (*int*) – PNG or PDF

Returns

This method should be used after method queryHitPathways.

```
res = s.http_post("highlightPathwayDiagram/68875/PNG", frmt="txt",
data="CDC2")
with open("test.png", 'wb') as f:
import binascii
f.write(binascii.a2b_base64(res))
```

list_by_query(*classname*, ***kargs*)

Get list of objects from Reactome database

Parameters

- **name** (*str class*) –
- **kargs** – further attribute values encoded in key-value pair

Returns

list of dictionaries. Each dictionary contains information about a given pathway

To query a list of pathways with names as “Apoptosis”:

```
>>> s = Reactome()
>>> res = list_by_query("Pathway", name="apoptosis")
>>> identifiers = [x['dbId'] for x in res]
```

pathway_complexes(*identifier*)

Get complexes belonging to a pathway

Parameters

identifier (*int*) – Pathway database identifier

Returns

list of all PhysicalEntity objects that participate in the Pathway.(in JSON)

```
>>> s = Reactome()
>>> s.pathway_complexes(109581)
```

pathway_diagram(*identifier*, *frmt*='PNG')

Retrieve pathway diagram

Parameters

- **identifier** (*int*) – Pathway database identifier
- **frmt** (*str*) – PNG, PDF, or XML.

Returns

Base64 encoded pathway diagram for PNG or PDF. XML text for the XML file type.

```
>>> s = Reactome()
>>> s.pathway_diagram('109581', 'PNG', view=True)
>>> s.pathway_diagram('109581', 'PNG', save=True)
```

Todo: if PNG or PDF, the output is base64 but there is no facility to easily save the results in a file for now

pathway_hierarchy(*species*)

Get the pathway hierarchy for a species as displayed in Reactome pathway browser.

Parameters

species (*str*) – species name that should be with + or spaces (e.g. ‘homo+sapiens’ for human, or ‘mus musculus’ for mouse)

Returns

XML text containing pathways and reactions

```
s.pathway_hierarchy("homo sapiens")
```

pathway_participants(*identifier*)

Get list of pathway participants for a pathway using

Parameters

identifier (*int*) – Pathway database identifier

Returns

list of fully encoded PhysicalEntity objects in the pathway (in JSON)

```
>>> s = Reactome()
>>> s.pathway_participants(109581)
```

query_by_id(*classname*, *identifier*)

Get Reactome Database for a specific object.

Parameters

- **classname** (*str*) – e.g. Pathway
- **identifier** (*int*) – database identifier or stable identifier if available

It returns a full object, including full class information about all the attributes of the returned object. For example, if the object has one PublicationSource attribute, it will return a full PublicationSource object within the returned object.

```
>>> s.query_by_id("Pathway", "109581")
```

query_by_ids(*classname*, *identifiers*)

Parameters

- **classname** (*str*) – e.g. Pathway
- **identifiers** (*list*) – list of strings or int

```
>>> s.quey_by_ids("Pathway", "CDC2")
```

Warning: not sure the wrapping is correct

query_hit_pathways(*query*)

Get pathways that contain one or more genes passed in the query list.

In the Reactome data model, pathways are organized in a hierarchical structure. The returned pathways in this method are pathways having detailed manually drawn pathway diagrams. Currently only human pathways will be returned from this method.

```
s.query_hit_pathways('CDC2')
s.query_hit_pathways(['CDC2'])
```

query_pathway_for_entities(*identifiers*)

Get pathway objects by specifying an array of PhysicalEntity database identifiers.

The returned Pathways should contain the passed EventEntity objects. All passed EventEntity database identifiers should be in the database.

species_list()

Get the list of species used Reactome

Readseq

This module provides a class *Seqret* to access to Seqret WS.

What is Seqret ?**URL**

<http://www.ebi.ac.uk/Tools/services/rest/seqret/>

Service**Citations**

<http://www.ncbi.nlm.nih.gov/pubmed/18428689>

EMBOSS seqret reads and converts biosequences between a selection of common biological sequence formats, including EMBL, GenBank and fasta sequence formats.

Seqret homepage – Sep 2017

class Seqret(verbose=True)

Interface to the *Seqret* service

```
>>> from bioservices import *
>>> s = Seqret()
```

The ReadSeq service was replaced by #the Seqret services (2015).

Changed in version 0.15.

Constructor**Parameters**

verbose (*bool*) –

get_parameter_details(parameterId)

Get details of a specific parameter.

Parameters

parameter (*str*) – identifier/name of the parameter to fetch details of.

Returns

a data structure describing the parameter and its values.

```
rs = ReadSeq()
print(rs.get_parameter_details("stype"))
```

get_parameters()

Get a list of the parameter names.

Returns

a list of strings giving the names of the parameters.

get_result(*jobid*, *result_type*='out')

Get the result of a job of the specified type.

Parameters

- **jobid** (*str*) – job identifier.
- **parameters** – optional list of `wsRawOutputParameter` used to provide additional parameters for derived result types.

get_result_types(*jobid*)

Get the available result types for a finished job.

Parameters

jobid (*str*) – job identifier.

Returns

a list of `wsResultType` data structures describing the available result types.

get_status(*jobid*=None)

Get the status of a submitted job.

Parameters

jobid (*str*) – job identifier.

Returns

string containing the status.

The values for the status are:

- **RUNNING**: the job is currently being processed.
- **FINISHED**: job has finished, and the results can then be retrieved.
- **ERROR**: an error occurred attempting to get the job status.
- **FAILURE**: the job failed.
- **NOT_FOUND**: the job cannot be found.

property parameters

Get list of parameter names

run(*email*, *title*, ***kargs*)

Submit a job to the service.

Parameters

- **email** (*str*) – user e-mail address.
- **title** (*str*) – job title.
- **params** – parameters for the tool as returned by `get_parameter_details()`.

Returns

string containing the job identifier (`jobId`).

Deprecated (olf readseq service):

| Format Name | Value |
|----------------|-------|
| Auto-detected | 0 |
| EMBL | 4 |
| GenBank | 2 |
| Fasta(Pearson) | 8 |

(continues on next page)

(continued from previous page)

| | |
|-----------------|----|
| Clustal/ALN | 22 |
| ACEDB | 25 |
| BLAST | 20 |
| DNAStrider | 6 |
| FlatFeat/FFF | 23 |
| GCG | 5 |
| GFF | 24 |
| IG/Stanford | 1 |
| MSF | 15 |
| NBRF | 3 |
| PAUP/NEXUS | 17 |
| Phylip(Phylip4) | 12 |
| Phylip3.2 | 11 |
| PIR/CODATA | 14 |
| Plain/Raw | 13 |
| SCF | 21 |
| XML | 19 |

As output, you also have

Pretty 18

```
s = readseq.Seqret()
jobid = s.run("cokelaer@test.co.uk", "test", sequence=fasta, inputformat=8,
             outputformat=2)
genbank = s.get_result(s._jobid)
```

UniChem

This module provides a class *UniChem*

What is UniChem

URL

<https://www.ebi.ac.uk/uniche/inf/webservices>

REST

<https://www.ebi.ac.uk/uniche/rest>

“UniChem is a ‘Unified Chemical Identifier’ system, designed to assist in the rapid cross-referencing of chemical structures, and their identifiers, between databases (read more). “

—From UniChem web page June 2013

```
class UniChem(verbose=False, cache=False)
```

Interface to the *UniChem* service

```
>>> from bioservices import UniChem
>>> u = UniChem()
```

There are lots of sources such as ChEMBL, Chebi, etc. You will probably need the identifiers of those sources. You can get all information about a source using these methods:

```
# Get information about a source
u.get_source_info_by_name('chembl')
u.get_source_info_by_id(10)
u.get_id_from_name('chembl')
u.get_all_src_ids()
```

but for developers, everything is contained in the `source_ids` dictionary.

The first important method provided by Unichem API is the `get_compounds()`. For example, you can request all compounds related to the ChEMBL12 identifier from ChEMBL using:

```
res = u.get_compounds('CHEMBL12', 'chembl')
compounds = res['compounds'][0]
```

Note that the second argument is 'chembl' and lower/upper cases is important. All names are stored in `source_ids` together with their identifiers.

You can use also `get_id_from_name()` and `get_name_from_id` if needed.

Legacy methods are available:

```
get_compound_ids_from_src_id -> use get_compounds()
get_src_compound_ids_from_inchikey -> replaced by get_compounds()
get_all_src_ids() -> uses new API
get_src_compound_ids_all_from_inchikey -> get_source_by_inchikey()
get_verbose_src_compound_ids_from_inchikey -> get_sources_by_inchikey_verbose()
get_structure -> uses new API get_compounds() and bioservices code get_structure_all
-> dropped
get_src_compound_id_url -> dropped. One can use the get_compounds()
get_src_compound_ids_all_from_obsolete -> removed

get_src_compound_ids_from_src_compound_id -> removed; was obsolet
get_src_compound_ids_all_from_src_compound_id -> removed; was already obsolet
get_all_compound_ids_from_all_src_id -> removed. no more API
get_mapping -> removed. no more API
get_auxiliary_mappings -> removed. no more API
```

Most old functions can be replaced by a syntax such as:

```
res = u.get_compound('CHEMBL12', 'chembl')
res['compounds'][0]
```

Constructor UniChem

Parameters

verbose – set to False to prevent informative messages

get_all_src_ids()

Obtain all `src_ids` of sources available in UniChem

Returns

list of 'src_id's.

```
uni.get_all_src_ids()
```

get_compounds(compound, source_type)

Get matched compounds information

Parameters

- **compound** (*str*) – InChI, InChIKey, Name, UCI or Compound Source ID
- **source_type** – uci, inchi, inchikey, sourceID (e.g. chembl)
- **sourceID** (*str*) – ID for the source assigned in UniChem when the type is “sourceID”

Returns

a list of matched compounds and their assigned sources

A legacy function allows you to retrieve a compound from its inchikey:

```
u.get_sources_by_inchikey('GZUITABIAKMVPG-UHFFFAOYSA-N')
```

However, this new function is faster presumably and allows you to do the same:

```
res = u.get_compounds('GZUITABIAKMVPG-UHFFFAOYSA-N', 'inchikey')
res['compounds']
```

You can get the first element, from which inchi, sources, standardInchikey, uci can be extracted. The **sources** key contains all compound identifiers for each source:

```
res['compounds'][0]['uci']
res['compounds'][0]['sources']
```

Looks like there is always a single element in res[‘compounds’] but since it is a list, you must access to first element (unique) using [0] syntax.

get_connectivity(*compound*, *source_type*)

Fetch multiple source data sets for a given compound with common connectivity to a given id on the database source, InChI, InChIkey or UCI

Parameters

- **compound** (*str*) – InChI, InChIKey, Name, UCI or Compound Source ID (e.g. chembl)
- **source_type** – uci, inchi, inchikey, sourceID

The returned dictionary contains 5 keys:

- response: service response (‘Success’ if everything is right)
- searchedCompound: the summary in terms of inchi, standardInchikey and uci
- **sources: a dictionary with e.g. compoundID and name of the source.**
A ‘comparison’ dictionary is also provided.
- totalCompounds: number of searchedCompound entries
- totalSources: number of sources entries

get_id_from_name(*name*)

Return the ID a source given its name.

Parameters

name (*str*) – a valid database name (e.g., chembl)

```
u.get_id_from_name("chembl")
```

get_images(*uci*, *filename=None*)

Return / create compound image

Parameters

- **uci** – the UCI of the compound
- **filename** – optional file name to save the SVG+XML output

Returns

the SVG+XML string

get_inchi_from_inchikey(*inchikey*)

Get a list of inchis given a valid inchikey.

Parameters

inchikey – InChI Key to search. Unlike the rest API, you can also provide a list.

Returns

a list of inchis matching the InChI Key provided. If input is a list, a dictionary is returned where keys are the inchikey input lists.

```
from bioservices import UniChem
u = UniChem()
res = u.get_inchi("AAOVKJBEBIDNHE-UHFFFAOYSA-N")
```

Note: this is a legacy function. introduced in v1.9 after unichem API update

get_source_info_by_id(*ID*)

get_source_info_by_name(*src_name*)

Description: Obtain all information on a source by querying with a source id

Parameters

src_name (*int*) – valid identifiers can be found in *source_ids* e.g. chebi, chembl)

Returns

dictionary (or list of dictionaries) with following keys:

- UCICount: number of entries
- baseUrl: URL of the source
- created: date of creation
- description: a description of the content of the source
- lastUpdated: last date of the update
- name: the unique name for the source in UniChem, always lower case
- nameLabel: A name for the source suitable for use as a 'label' for the source
- nameLong: the full name of the source, as defined by the source
- private: is it private or not ?
- sourceID: the *src_id* for this source
- srcDetails: details about the source
- srcReleaseDate: release date of the source database
- srcReleaseNumber: release number of the source
- srcUrl: *src_url* (the main home page of the source)
- updateComments: possible updates from this source

```
>>> res = get_source_by_name("chebi")
```

get_sources()

Returns all information about all sources used in Unichem

```
from bioservices import UniChem
u = UniChem()
res = u.get_sources_information()
res['sources']
```

get_sources_by_inchikey(*inchikey*)

Get sources by inchikey

Parameters

inchikey – InChI Key to search. Unlike the rest API, you can also provide a list.

Returns

A list of sources for the provided InChIKey if input is a single string. a dictionary with keys as inchikey if input is a list.

Note: this is a legacy function. introduced in v1.9 after unichem API update

get_sources_by_inchikey_verbose(*inchikey*)

Get sources by inchikey

Parameters

inchikey – InChI Key to search. Unlike the rest API, you can also provide a list.

Returns

A list of sources for the provided InChIKey if input is a single string. a dictionary with keys as inchikey if input is a list.

Note: this is a legacy function. introduced in v1.9 after unichem API update

get_structure(*compound_id*, *src_id*)

Obtain structure(s) CURRENTLY assigned to a query *src_compound_id*.

Parameters

- **compound_id** (*str*) – a valid compound identifier
- **src_id** (*int*) – corresponding database identifier (name or id).

Returns

dictionary with ‘standardinchi’ and ‘standardinchikey’ keys

```
>>> uni.get_structure("ChEMBL12", "chembl")
```

UniProt

Interface to some part of the UniProt web service

What is UniProt ?

URL

<http://www.uniprot.org>

Citation

“The Universal Protein Resource (UniProt) is a comprehensive resource for protein sequence and annotation data. The UniProt databases are the UniProt Knowledgebase (UniProtKB), the UniProt Reference Clusters (UniRef), and the UniProt Archive (UniParc). The UniProt Metagenomic and Environmental Sequences (UniMES) database is a repository specifically developed for metagenomic and environmental data.”

—From Uniprot web site (help/about) , Dec 2012

```
class UniProt(verbose=False, cache=False)
```

Interface to the [UniProt](#) service

```
>>> from bioservices import UniProt
>>> u = UniProt(verbose=False)
>>> u.mapping("UniProtKB_AC-ID", "KEGG", query='P43403')
defaultdict(<type 'list'>, {'P43403': ['hsa:7535']})
>>> res = u.search("P43403")

# Returns sequence on the ZAP70_HUMAN accession Id
>>> sequence = u.search("ZAP70_HUMAN", columns="sequence")
```

Changed in version 1.10: Uniprot update its service in June 2022. Changes were made in the bioservices API with small changes. User API is more or less the same. Main issues that may be faced are related to change of output column names. Please see the `_legacy_names` for corresponding changes.

Some notes about searches. The *and* and *or* are now upper cases. The *organism* and *taxonomy* fields are now *organism_id* and *taxonomy_id*

Constructor

Parameters

- **verbose** – set to False to prevent informative messages
- **cache** – set to True to cache request

```
get_df(entries, nChunk=100, organism=None, limit=10)
```

Given a list of uniprot entries, this method returns a dataframe with all possible columns

Parameters

- **entries** – list of valid entry name. if list is too large (about >200), you need to split the list
- **chunk** –
- **limit** – limit number of entries per identifier to 10. You can set it to None to keep all entries but this will be very slow

Returns

dataframe with indices being the uniprot id (e.g. DIG1_YEAST)

get_fasta(uniprot_id)

Returns FASTA string given a valid identifier

Parameters

uniprot_id (*str*) – a valid identifier (e.g. P12345)

This is just an alias to [retrieve\(\)](#) when setting the format to 'fasta'. Method kept for legacy.

mapping(*fr='UniProtKB_AC-ID', to='KEGG', query='P13368', polling_interval_seconds=3, max_waiting_time=100*)

This is an interface to the UniProt mapping service

Parameters

- **fr** – the source database identifier. See [valid_mapping](#).
- **to** – the targetted database identifier. See [valid_mapping](#).
- **query** – a string containing one or more IDs separated by a space It can also be a list of strings.
- **polling_interval_seconds** – the number of seconds between each status check of the current job
- **max_waiting_time** – the maximum number of seconds to wait for the final answer.

Returns

a dictionary with two possible keys. The first one is 'results' with the from / to answers and the second one 'failedIds' with Ids that were not found

```
>>> u.mapping("UniProtKB_AC-ID", "KEGG", 'P43403')
{'results': [{'from': 'P43403', 'to': 'hsa:7535'}]}
```

The output is a dictionary. Identifiers that were not found are stored in the keys 'failedIds'. Successful queries are stored in the 'results' key that is a list of dictionaries with two keys set to 'from' and 'to'. The 'from' key should be in your input list. The 'to' key is the result. Here we have the KEGG identifier recognised by its prefix 'hsa:', which is for human. Sometimes the output ('to') it is more complicated. Consider the following example:

```
u.mapping("UniParc", "UniProtKB", 'UPI00000000001,UPI00000000002')
```

You will see that the UniParc results is more complex than just an identifier.

See [valid_mapping](#) attribut for list of valid mapping identifiers.

Note that according to Uniprot (June 2022), there are various limits on ID Mapping Job Submission:

| Limit | Details |
|---------|--|
| 100,000 | Total number of ids allowed in comma separated param ids in /idmapping/run api |
| 500,000 | Total number of "mapped to" ids allowed |
| 100,000 | Total number of "mapped to" ids allowed to be enriched by UniProt data |
| 10,000 | Total number of "mapped to" ids allowed with filtering |

Changed in version 1.1.1: to return a dictionary instead of a list

Changed in version 1.1.2: the values for each key is now made of a list instead of strings so as to store more than one values.

Changed in version 1.2.0: input query can also be a list of strings instead of just a string

Changed in version 1.3.1::: use `http_post` instead of `http_get`. This is 3 times faster and allows queries with more than 600 entries in one go.

quick_search(*query*, *include_isoforms=False*, *sort='score'*, *limit=None*)

a specialised version of `search()`

This is equivalent to:

```
u = uniprot.UniProt()
u.search(query, frmt='tsv', sort="score", limit=None)
```

Returns

a dictionary.

retrieve(*uniprot_id*, *frmt='json'*, *database='uniprot'*, *include=False*)

Search for a uniprot ID in UniProtKB database

Parameters

- **uniprot** (*str*) – a valid UniProtKB ID, or uniref, uniparc or taxonomy.
- **frmt** (*str*) – expected output format amongst xml, txt, fasta, gff, rdf
- **database** (*str*) – database name in (uniprot, uniparc, uniref, taxonomy)
- **include** (*bool*) – include data with RDF format.

Returns

if the parameter `uniprot_id` is string, the output will be a a list of identifiers is provided, the output is also a list otherwise, a string. The content of the string of items in the list depends on the value of **frmt**.

```
>>> u = UniProt()
>>> res = u.retrieve("P09958", frmt="txt")
>>> fasta = u.retrieve(['P29317', 'Q5BKX8', 'Q8TCD6'], frmt='fasta')
>>> print(fasta[0])
```

Changed in version 1.10: the xml format is now returned as raw XML. It is not interpreted anymore. The RDF has now an additional option to include data from referenced data sets directly in the returned data (set `include=True` parameter). Default output format is now set to json.

search(*query*, *frmt='tsv'*, *columns=None*, *include_isoforms=False*, *sort='score'*, *compress=False*, *limit=None*, *offset=None*, *maxTrials=10*, *database='uniprotkb'*)

Provide some interface to the uniprot search interface.

Parameters

- **query** (*str*) – query must be a valid uniprot query. See <https://www.uniprot.org/help/query-fields> and examples below
- **frmt** (*str*) – a valid format amongst html, tab, xls, fasta, gff, txt, xml, rdf, list, rss. If tab or xls, you can also provide the columns argument. (default is tab)
- **columns** (*str*) – comma-separated list of values. Works only if format is tsv or xls. For UniProtKB, some possible columns are: id, entry name, length, organism. See also [valid_mapping](#) for the full list of column keywords.
- **include_isoform** (*bool*) – include isoform sequences when the `frmt` parameter is fasta. Include description when `frmt` is rdf.

- **sort** (*str*) – by score by default. Set to None to bypass this behaviour
- **compress** (*bool*) – gzip the results
- **limit** (*int*) – Maximum number of results to retrieve.
- **offset** (*int*) – Offset of the first result, typically used together with the limit parameter.
- **maxTrials** (*int*) – this request is unstable, so we may want to try several time.

To obtain the list of uniprot ID returned by the search of zap70 can be retrieved as follows:

```
>>> u.search('zap70+AND+organism_id:9606')
>>> u.search("zap70+AND+taxonomy_id:9606", frmt="tsv", limit=3,
...         columns="entry_name,length,id, gene_names")
Entry name  Length  Entry   Gene names
CBLB_HUMAN  982  Q13191  CBLB RNF56 Nbla00127
CBL_HUMAN   906  P22681  CBL  CBL2 RNF55
CD3Z_HUMAN  164  P20963  CD247 CD3Z T3Z TCRZ
```

other examples:

```
>>> u.search("ZAP70+AND+organism_id:9606", limit=3, columns="id,xref_pdb")
```

You can also do a search on several keywords. This is especially useful if you have a list of known entry names.:

```
>>> u.search("ZAP70_HUMAN+OR+CBL_HUMAN", frmt="tsv", limit=3,
...         columns="entry name,length,id, genes")
Entry name  Length  Entry   Gene names
```

Finally, note that when you search for a query, you may have several hits:

```
>>> u.search("P12345")
```

including the ID P12345 but also related entries. If you need only the entry that perfectly match the query, use:

```
>>> u.search("accession:P12345")
```

This was provided from a user issue that was solved here: <https://github.com/cokelaer/bioservices/issues/122>

Warning: some columns although valid may not return anything, not even in the header: 'score', 'taxonomy', 'tools'. this is a uniprot feature, not bioservices.

Changed in version 1.10: Due to uniprot API changes in June 2022:

- parameter 'include' is not named 'include_isoform'
- default parameter 'tab' is now 'tsv' but does not change the results

uniref(*query*)

Calls UniRef service

This is an alias to [retrieve\(\)](#)

```
>>> u = UniProt()
>>> u.uniref("Q03063")
```

Another example from <https://github.com/cokelaer/bioservices/issues/121> is the combination of uniprot and uniref filters:

```
u.uniref("uniprot:(ec:1.1.1.282 taxonomy_name:bacteria reviewed:true)")
```

Changed in version 1.10: due to uniprot API changes in June 2022, we now return a json instead of a pandas dataframe.

property valid_mapping

DBFetch

Interface to DBFetch web service

What is DBFetch

URL

<http://www.ebi.ac.uk/Tools/webservices/services/dbfetch>

Service

http://www.ebi.ac.uk/Tools/webservices/services/dbfetch_rest

“DBFetch allows you to retrieve entries from various up-to-date biological databases using entry identifiers or accession numbers. This is equivalent to the CGI based dbfetch service. Like the CGI service a request can return a maximum of 200 entries.”

—From <http://www.ebi.ac.uk/Tools/webservices/services/dbfetch> , Dec 2012

```
class DBFetch(verbose=False)
```

Interface to DBFetch service

```
>>> from bioservices import DBFetch
>>> w = DBFetch()
>>> data = w.fetchBatch("uniprot", "zap70_human", "xml", "raw")
```

For more information about the API, check this page: <http://www.ebi.ac.uk/Tools/dbfetch/syntax.jsp>

Constructor

Parameters

verbose (*bool*) – print informative messages

```
fetch(query, db='ena_sequence', format='default', style='raw', pageHtml=False)
```

Fetch an entry in a defined format and style.

Parameters

- **query** (*str*) – the entry identifier in db:id format (e.g. ‘UniProtKB:WAP_RAT’).
- **format** (*str*) – the name of the format required (default to fasta).
- **style** (*str*) – the name of the style required (raw, default, html)

Returns

The format of the response depends on the format/style parameter.

```
from bioservices import DBFetch
u = DBFetch()
db.fetch(db="ena_sequence", format="fasta", query="L12344,L12345")
db.fetch(db="uniprot", format="fasta", query="P53503")
```

If db is omitted, the default is ena_sequence. If format is omitted, the default is EMBL format. The default style is raw data.

get_all_database_info()

Get details of all available databases, includes formats and result styles.

Returns

A list of data structures describing the databases. See `getDatabaseInfo()` for a description of the data structure.

get_database_format_styles(db, format)

Get a list of style names available for a given database and format.

Parameters

- **db** (*str*) – database name to get available styles for (e.g. uniprotkb).
- **format** (*str*) – the data format to get available styles for (e.g. fasta).

Returns

An array of strings containing the style names.

```
>>> u.get_database_format_styles("uniprotkb", "fasta")
['default', 'raw', 'html']
```

get_database_formats(db)

Get list of format names for a given database.

Parameters

db (*str*) – valid database name

```
>>> db.get_database_formats("uniprotkb")
['default',
 'annot',
 'entrysize',
 'fasta',
 'gff3',
 'seqxml',
 'uniprot',
 'uniprottrdxml',
 'uniprotxml',
 'dasgff',
 'gff2']
```

get_database_info(db=None)

Get details describing specific database (data formats, styles)

Parameters

db (*str*) – a valid database.

Returns

The output can be introspected and contains several attributes

```
>>> res = u.get_database_info('uniprotkb')
>>> print(res['description'])
'The UniProt Knowledgebase (UniProtKB) is the central access point for
↳ extensive curated protein information, including function, classification,
↳ and cross-references. Search UniProtKB to retrieve everything that is known
↳ about a particular sequence.'
```

property supported_databases

Alias to getSupportedDBs.

Wikipathway

Interface to the WikiPathway service

What is WikiPathway ?

URL

<http://www.wikipathways.org/index.php/WikiPathways>

REST

<http://webservice.wikipathways.org/>

Citation

doi:10.1371/journal.pone.0006447

” WikiPathways is an open, public platform dedicated to the curation of biological pathways by and for the scientific community.”

—From WikiPathway web site. Dec 2012

class WikiPathways(verbose=True, cache=False)

Interface to Pathway service

```
>>> from bioservices import WikiPathways
>>> s = Wikipathway()
>>> s.organism # default organism
'Homo sapiens'
```

Examples:

```
s.findPathwaysByText('MTOR')
s.getPathway('WP1471')
s.getPathwaysByOntologyTerm('DOID:344')
s.findPathwaysByXref('P45985')
```

The methods that require a login are not implemented (*login()*, *updatePathway()*, *removeCurationTag()*, *saveCurationTag()*, *createPathway()*)

Methods not implemented at all:

- u'getCurationTagHistory': No API found in Wikipathway web page
- u'getRelations': No API found in Wikipathway web page

Constructor

Parameters

verbose (*bool*) –

createPathway(*gpmlCode*, *authInfo*)

Create a new pathway on the WikiPathways website with a given GPML code.

Warning: Interface not exposed in bioservices.

Note: To create/modify pathways via the web service, you need to have an account with web service write permissions. Please contact us to request write access for the web service.

Parameters

- **gpml** (*str*) – The GPML code.
- **auth** (*object WSAuth*) – The authentication info.

Returns

WSPathwayInfo The pathway info for the created pathway (containing identifier, revision, etc.).

findInteractions(*query*)

Find interactions defined in WikiPathways pathways.

Parameters

query (*str*) – The name of an entity to find interactions for (e.g. 'P53')

Returns

list of dictionaries

```
res = w.findInteractions("P53")
```

findPathwaysByLiterature(*query*)

Find pathways by their literature references.

Parameters

query (*str*) – The query, can be a pubmed id, author name or title keyword.

Returns

dictionary with Pathway as keys

```
res = s.findPathwaysByLiterature(18651794)
```

findPathwaysByText(*query*, *species=None*)

Find pathways using a textual search on the description and text labels of the pathway objects.

The query syntax offers several options:

- Combine terms with AND and OR. Combining terms with a space is equal to using OR ('p53 OR apoptosis' gives the same result as 'p53 apoptosis').
- Group terms with parentheses, e.g. '(apoptosis OR mapk) AND p53'
- You can use wildcards * and ?. * searches for one or more characters, ? searches for only one character.

- Use quotes to escape special characters. E.g. “‘apoptosis*’” will include the * in the search and not use it as wildcard.

This function supports REST-style invocation. Example: <http://www.wikipathways.org/wpi/webService/webService.php/findPathwaysByText?query=apoptosis>

Parameters

- **query** (*str*) – The search query (e.g. ‘apoptosis’ or ‘p53’).
- **species** (*str*) – The species to limit the search to (leave blank to search on all species).

Returns

Array of WSSearchResult An array of search results.

```
s.findPathwaysByText(query="p53 OR mapk",species='Homo sapiens')
```

Warning: AND or OR must be in big caps

findPathwaysByXref(ids, codes=None)

Find pathways by searching on the external references of DataNodes.

Parameters

- **ids** (*str string*) – One or more DataNode identifier(s) (e.g. ‘P45985’). Datanodes can be (gene/protein/metabolite identifiers). For one node, you can use a string (or number) or list of one identifier. you can also provide a list of identifiers.
- **codes** (*str*) – You can restrict the search to a specific database. See <http://developers.pathvisio.org/wiki/DatabasesMapps#Supporteddatabasesystems> for details. Examples are “L” for entrez gene, “En” for ensembl. See also the note here below for multiple identifiers/codes.

Returns

a dictionary

```
>>> s.findPathwaysByXref(ids="P45985")
>>> s.findPathwaysByXref(ids="P45985", codes="L")
>>> s.findPathwaysByXref(ids=["P45985"], codes=["L"])
>>> s.findPathwaysByXref(ids=["P45985", "ENSG00000130164"], codes=["L", "En"])
```

Note that in the last example, we specify multiple ids and codes parameters to query for multiple xrefs at once. In that case, the number of ids and codes parameters should match. Moreover, they will be paired to form xrefs, so P45985 is searched for in the “L” database while “ENSG00000130164” is searched for in the En” database only.

getColoredPathway(pathwayId, filetype='svg', revision=0, color=None, graphId=None)

Get a colored image version of the pathway.

Parameters

- **pwId** (*str*) – The pathway identifier.
- **revision** (*int*) – The revision number of the pathway (use ‘0’ for most recent version).
- **fileType** (*str*) – The image type (One of ‘svg’, ‘pdf’ or ‘png’). Not yet implemented. svg is returned for now.

Returns

Binary form of the image.

Todo: graphId, color parameters

getCurationTags(*pathwayId*)

Get all curation tags for the given pathway.

Parameters

pathwayId (*str*) – the pathway identifier.

Returns

Array of WSCurationTag. The curation tags.

```
s.getCurationTags("WP4")
```

getCurationTagsByName(*name*)

Get all curation tags for the given tag name.

Use this method if you want to find all pathways that are tagged with a specific curation tag.

Parameters

tagName (*str*) – The tag name.

Returns

Array of WSCurationTag. The curation tags (one instance for each pathway that has been tagged).

```
s.getCurationTagsByName("Curation:FeaturedPathway")
```

getOntologyTermsByPathway(*pathwayId*)

Get a list of ontology terms for a given pathway.

Parameters

pathwayId (*str*) – the pathway identifier.

Returns

Array of WSOntologyTerm. The ontology terms.

```
s.getOntologyTermsByPathway("WP4")
```

getPathway(*pathwayId*, *revision=0*)

Download a pathway from WikiPathways.

Parameters

- **pathwayId** (*str*) – the pathway identifier.
- **revision** (*int*) – the revision number of the pathway (use '0' for most recent version).

Returns

The pathway as a dictionary. The pathway is stored in gpml format.

```
s.getPathway("WP2320")
```

getPathwayAs(*pathwayId*, *filetype='png'*, *revision=0*)

Download a pathway in the specified file format.

Parameters

- **pathwayId** (*str*) – the pathway identifier.

- **filetype** (*str*) – the file format (default is .owl).
- **revision** (*int*) – the revision number of the pathway (use '0' for most recent version - this is default).

Returns

The file contents

Changed in version 1.3.0: return raw output of the service without any parsing

Note: use `savePathwayAs()` to save into a file.

getPathwayHistory(*pathwayId*, *date*)

Get the revision history of a pathway.

Parameters

- **pathwayId** (*str*) – the pathway identifier.
- **date** (*str*) – limit the results by date, only history items after the given date (timestamp format) will be included. Can be a string or number of the form YYYYMMDDHHMMSS.

Returns

The revision history.

Warning: seems unstable does not return the results systematically.

```
s.getPathwayHistory("WP4", 20110101000000)
```

getPathwayInfo(*pathwayId*)

Get some general info about the pathway.

Parameters

pathwayId (*str*) – the pathway identifier.

Returns

The pathway info.

```
>>> from bioservices import *
>>> s = Wikipathway()
>>> s.getPathwayInfo("WP2320")
```

getPathwaysByOntologyTerm(*terms*)

Get a list of pathways tagged with a given ontology term.

Parameters

terms (*str*) – the ontology term identifier.

Returns

dataframe with pathways information.

```
>>> from bioservices import WikiPathways
>>> s = Wikipathway()
>>> s.getPathwaysByOntologyTerm('PW:0000724')
```

getPathwaysByParentOntologyTerm(*term*)

Get a list of pathways tagged with any ontology term that is the child of the given Ontology term.

Parameters

term (*str*) – the ontology term identifier.

Returns

List of WSPathwayInfo The pathway information.

getRecentChanges(*timestamp*)

Get the recently changed pathways.

Parameters

timestamp (*str*) – Only get changes from after this time. Timestamp format: yyyyymmddMMHHSS (string or number)

Returns

The changed pathways in XML format

```
s.getRecentChanges(20110101000000)
```

Todo: interpret XML

listOrganisms()

listPathways(*organism=None*)

Get a list of all available pathways.

Parameters

organism (*str*) – If provided, the data is filtered to keep only the organism provided, which must be a valid name (check out [organism](#) attribute)

Returns

dataframe. Index are the pathways identifiers (e.g. WP1)

login(*username, password*)

Start a logged in session using an existing WikiPathways account.

Warning: Interface not exposed in bioservices.

This function will return an authentication code that can be used to execute methods that need authentication (e.g. updatePathway).

Parameters

- **name** (*str*) – The username of the WikiPathways account.
- **password** (*str*) – The password of the WikiPathways account.

Returns

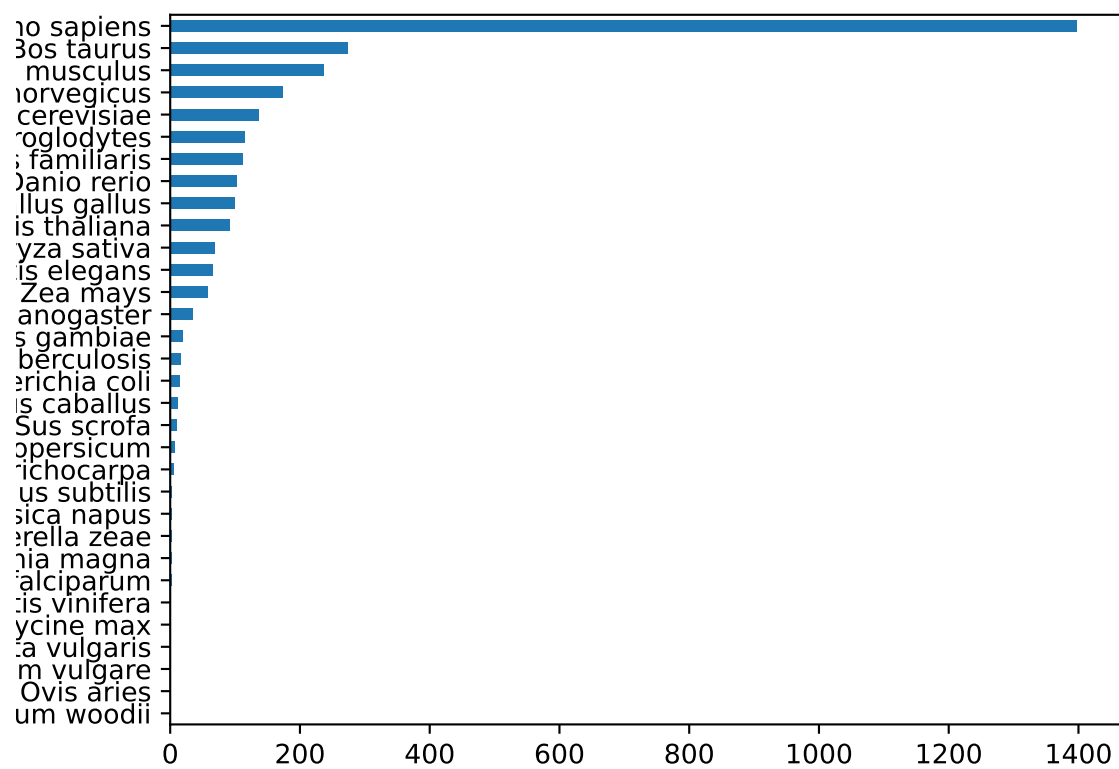
The authentication code for this session.

property organism

Read/write attribute for the organism

organisms

Get a list of all available organisms.



removeCurationTag(*pathwayId, name*)

Remove a curation tag from a pathway.

Warning: Interface not exposed in bioservices.

saveCurationTag(*pathwayId, name, revision*)

Apply a curation tag to a pathway. This operation will overwrite any existing tag with the same name.

Warning: Interface not exposed in bioservices.

Parameters

pathwayId (*str*) – the pathway identifier.

savePathwayAs(*pathwayId, filename, revision=0, display=True*)

Save a pathway.

Parameters

- **pathwayId** (*str*) – the pathway identifier.
- **filename** (*str*) – the name of the file. If a filename extension is not provided the pathway will be saved as a pdf (default).
- **revisionNumb** (*int*) – the revision number of the pathway (use '0 for most recent version).
- **display** (*bool*) – if True the pathway will be displayed in your browser.

Note: Method from bioservices. Not a WikiPathways function

Changed in version 1.7: return PNG by default instead of PDF. PDF not working as of 20 Feb 2020 even on wikipathway website.

showPathwayInBrowser(*pathwayId*)

Show a given Pathway into your favorite browser.

Parameters

pathwayId (*str*) – the pathway identifier.

updatePathway(*pathwayId, describeChanges, gpmlCode, revision=0*)

Update a pathway on WikiPathways website with a given GPML code.

Warning: Interface not exposed in bioservices.

Note: To create/modify pathways via the web service, you need to have an account with web service write permissions. Please contact us to request write access for the web service.

Parameters

- **pwId** (*str*) – The pathway identifier.
- **description** (*str*) – A description of the modifications.

- **gpml** (*str*) – The updated GPML code.
- **revision** (*int*) – The revision number of the version this GPML code was based on. This is used to prevent edit conflicts in case another client edited the pathway after this client downloaded it.
- **WSAuth_auth** (*object*) – The authentication info.

Returns

Boolean. True if the pathway was updated successfully.

6.1.9 Applications and extra tools

Web services have lots of overlap amongst themselves. For instance, fetching a FASTA sequence can be done using many different services. Yet, once a FASTA is retrieved, one may want to perform additional tasks or save the FASTA into a file or whatever repetitive functionalities not included in Web Services anymore.

The goal of this sub-package is to provide convenient tools, which are not web services per se but that makes use of one or several Web Services already available within BioServices.

Warning: this is experimental and was added in version 1.2.0 so it may change quite a lot.

Peptides

class **Peptides**(*verbose=False*)

```
>>> p = Peptides()
>>> p.get_peptide_position("Q8IYB3", "VPKPEPIPEPKESPE")
189
```

Sometimes, peptides are provided with a pattern indicating the phospho site. e.g.,

```
>>>
```

get_fasta_sequence(*uniprot_name*)

get_phosphosite_position(*uniprot_name, peptide*)

FASTA

class **FASTA**

Dedicated class to manipulates FASTA sequence(s)

Here is a FASTA file example:

```
>sp|P43408|KADA_METIG Adenylate kinase OS=Methanotorris igneus GN=adkA PE=1 SV=2
MKNKVVVVTGVPVGVTTLTQKTIEKLKEEGIEYKMNFGTVMFEVAKEEGLVEDRDQMR
KLDPDTQKRIQKLAGRKIAEMAKESNVIVDTHSTVKTPKGYLAGLPIWVLEELNPDIIVI
VETSSDEILMRRLGDATRNRDIELTSDIDEHQFMNRCAAMAYGVLTGATVKIKNRDGLL
DKAVEELISVLK
```

The format is made of a header and a sequence. Any FASTA can be read and the pair of header/sequence retrieved from the `sequence` and `header` attributes. However, headers differ from one database to another one and interpretation is not implemented except for SWISS-PROT. Identifiers can be retrieved whatsoever.

You can read a FASTA sequence from a local file or download one from UniProt

```
>>> from bioservices.apps.fasta import FASTA
>>> f = FASTA()
>>> f.load("P43403")
>>> acc = f.accession    # the accession (P43403)
>>> fasta = f.fasta      # raw FASTA string
>>> seq = f.sequence     # the sequence itself
>>> header = f.header    # the header itself
>>> identifier = f.identifier
```

You can also get a dataframe also using Pandas library.:

```
>>> f.df
```

The columns stored in the dataframe encompass:

- **Accession** that is taken from the header (e.g., P43403 from uniprot)
- **Sequence**, a copy of the FASTA sequence
- **Size**, the length of the sequence.
- **Database**, the database type found in the header (e.g., sp for SWISS-PROT; see below for a list of database and their header format).
- Some column such as **Organism** are filled only for some database
- **Identifiers** is the beginning of the header.

See also:

[*MultiFASTA*](#) for multi FASTA manipulation.

List of identifiers corresponding to different databases.

| | |
|----------------------------------|----------------------------------|
| | |
| GenBank | gi gi-number gb accession locus |
| EMBL Data Library | gi gi-number emb accession locus |
| DDBJ, DNA Database of Japan | gi gi-number dbj accession locus |
| NBRF PIR | pir entry |
| Protein Research Foundation | prf name |
| SWISS-PROT | sp accession name |
| Brookhaven Protein Data Bank (1) | pdb entry chain |
| Brookhaven Protein Data Bank (2) | entry:chain PDBID CHAIN SEQUENCE |
| Patents | pat country number |
| GenInfo Backbone Id | bbs number |
| General database identifier | gnl database identifier |
| NCBI Reference Sequence | ref accession locus |
| Local Sequence identifier | lcl identifier |

The `:meth::load_fasta` relies on UniProt service.

property PE

returns PE keyword found in the header if any

property SV

returns SV keyword found in the header if any

property accession

property dbtype

property df

property entry

returns entry only

property fasta

returns FASTA content

property gene_name

returns gene name from GN keyword found in the header if any

get_fasta(id_)

Fetches FASTA from uniprot and loads into attribute *fasta*

Parameters

id (*str*) – a given uniprot identifier

Returns

the FASTA contents

property header

returns header only

property identifier

known_dbtypes = ['sp', 'gi']

load(id_)

load_fasta(id_)

Fetches FASTA from uniprot and loads into attribute *fasta*

Parameters

id (*str*) – a given uniprot identifier

Returns

nothing

Note: same as *get_fasta()* but returns nothing

property name

property organism

returns organism from OS keyword found in the header if any

read_fasta(filename)

Reads a FASTA file and loads it

Type:

```
>>> f = FASTA()
>>> f.read_fasta(filename)
>>> f.fasta
```

Returns

nothing

Warning: If more than one FASTA is contained in the file, an error is raised

save_fasta(filename)

Save FASTA file into a filename

Parameters

- **data** (*str*) – the FASTA contents
- **filename** (*str*) – where to save it

property sequence

returns the sequence only

class MultiFASTA

Class to manipulate several several FASTA items

Here, we load some FASTA using UniProt web service:

```
>>> from bioservices import MultiFASTA
>>> mf = MultiFASTA()
>>> mf.load_fasta("P43408")
>>> mf.load_fasta("P21318")
```

You can then get back to your accession entries as follows

```
>>> mf.ids
['P43408', 'P21318']
```

And the sequences in the same order can be accessed:

```
>>> len(mf)
2
```

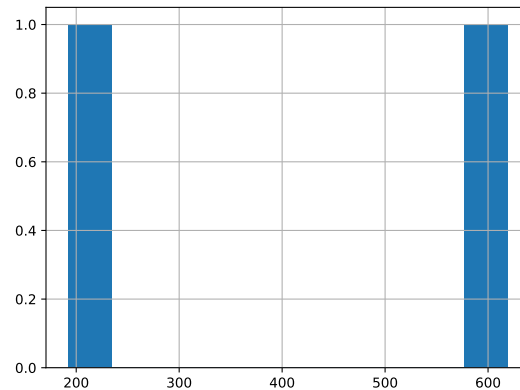
Each FASTA is stored in *fasta*, which is a dictionary where each values is an instance of *FASTA*:

```
>>> print(mf._fasta["P43408"].fasta)
>sp|P43408|KADA_METIG Adenylate kinase OS=Methanotorris igneus GN=adkA PE=1 SV=2
MKNKVVVVTGVPVGVGGTTLTQKTIEKLKEEGIEYKMVNFVTVMFEVAKEEGLVEDRDQMR
KLDPDPTQKRIQKLAGRKIAEMAKESNVIVDTHSTVKT PKGYLAGLPIWVLEELNPDIIVI
VETSSDEILMRRLGDATRNRDIELTSDIDEHQFMNRCAAMAYGVLTGATVKI IKNRDGLL
DKAVEELISVLK
```

The most convenient way to access to all data is to use the dataframe attribute:

```
>>> mf.df.Sequence
```

```
>>> from bioservices.apps import MultiFASTA
>>> f = MultiFASTA()
>>> f.load_fasta(["P43403", "P43410"])
>>> f.df.Size.hist()
```



property df

property fasta

Returns all FASTA instances

hist_size(kws)**

property ids

returns list of keys/accession identifiers

load_fasta(ids)

Loads a single FASTA file into the dictionary

read_fasta(filename)

Load several FASTA from a filename

save_fasta(filename)

Save all FASTA into a file

6.1.10 References to BioServices on the Web

- Galaxy: See the Log Archive at [Galaxy log archive](#)
- EBI: See [EBI programming web services](#)
- WikiPathways: [references](#)
- GeneProf: [example python](#)
- <http://www.scoop.it/t/bioinformatique>
- <http://bioinfo-fr.net/bioservices-module-python>

- <http://devbio.eu/?p=resources&presel=bioinfo>
- biomartian from Endre Bakken Stovner <https://github.com/endrebak/biomartian>

6.1.11 FAQs

General Errors

The most common errors come from

1. The web service that you are trying to access is down (temporarily or not)
2. The web service API has changed
3. A request inside bioservices is incorrect
4. A timeout occurred.

For the first problem, we cannot do anything except wait for the service to be up again.

For the second, you are trying to update bioservices to reflect those changes.

For the two other issues, which are really bioservices problems, we recommend to rerun your code setting the logging level to debug and send the code and errors you see.

To set the debug level on on a web service:

```
u = UniProt(verbose=True)
u.logging.level = 'DEBUG'
```

Installation issues

ValueError: unknown locale: UTF-8 under Mac OS X 10.7 - Lion

The installation with PIP is succesful but I get a “*ValueError: unknown locale: UTF-8*” under Mac OS X 10.7 - Lion when typing **from bioservices import ***.

On solution is to fix your environment by typing the following code in a shell:

```
export LANG="it_IT.UTF-8"
export LC_COLLATE="it_IT.UTF-8"
export LC_CTYPE="it_IT.UTF-8"
export LC_MESSAGES="it_IT.UTF-8"
export LC_MONETARY="it_IT.UTF-8"
export LC_NUMERIC="it_IT.UTF-8"
export LC_TIME="it_IT.UTF-8"
export LC_ALL=
```

You can check if it works by typing

```
python -c 'import locale; print(locale.getdefaultlocale());'
```

If this works without error, then it is fixed and you should be able to import bioservices. If so, make this solution persistent by adding the code into your environment. For that, just copy and paste the code in a file called `.bashrc_profile` (or `.bashrc`)

reference
[blog entry](#)

General questions

How can I figure out the taxonomy identifier of the mouse ?

You can use the Taxon class that uses Ensembl/UniProt/Eutils depending on the tasks. Here, we do not know the scientific name of taxonomy identifier of the mouse. We can use the search_by_name fuction:

Warning: Taxon class is not part of BioServices but some utilities have been added to BioKit ([github.com/biokit](https://github.com/biokit/biokit))

Changed in version 1.3.

In earlier version of BioServices, you could use:

```
>>> from bioservices import Taxon
>>> t = Taxon()
>>> t.search_by_name("mouse")
u'10090'
```

But this is now in BioKit:

```
>>> from biokit import Taxonomy
>>> t = Taxonomy()
>>> results = t.fetch_by_name('mouse')
>>> results[0]['id']
u'10090'
```

How to convert ID from one database to another ?

Many web services provides convertors. In BioServices, you can access to Kegg and UniProt that both provides convertor. For instance with Kegg, you can convert all human (hsa) Kegg Id to uniprot Id with:

```
from bioservices import *
s = KEGG()
kegg_ids, uniprot_ids = s.conv("hsa", "uniprot")
```

Or you can use the uniprot mapping function:

```
from bioservices import *
u = UniProt()
u.mapping(to="KEGG_ID", fr="ACC", query="ZAP70_HUMAN")
```

Specific Usage

Why my uniprot request takes forever ?

This may happen. Consider:

```
from bioservices import *
u = UniProt()
u.search("P53")
```


This request performed on UniProt web sites is actually pretty fast but there are 386 pages of results. In BioServices, the search commands reads the 386 pages of results and then stores the result in a variable. So it may take a while.

More generally if a request returns a very long result, it may take a while. You can use the socket module:

```
import socket
socket.setdefaulttimeout(5.)
```

After 5 seconds, the read() call will stop returning whatever has been read so far.

KEGG service

Is it possible to get the pathway information for multiple proteins ?

Currently there is no such function. You can only retrieve pathways given a single protein Id. However, you can easily write such a function. Here is the code for 2 proteins:

```
>>> p1 = k.get_pathway_by_gene("7535", "hsa")    # correspond to ZAP70
>>> p2 = k.get_pathway_by_gene("6885", "hsa")    # 6885 correspond to MAP3K7
>>> [k1 for k1 in p1.keys() if k1 in p2.keys()]
['hsa04660', 'hsa04064']
```

There are 2 pathways containing the proteins 7535 and 6885.

Interest of the BioServices classes REST and WSDL ?

There are a few technical aspects covered by BioServices to ease our life when adding new modules such as timeout, long request, headers, and so on.

What is the difference between GET and POST

When the user enters information in a form and clicks Submit , there are two ways the information can be sent from the browser to the server: in the URL, or within the body of the HTTP request.

The alternative to the GET method is the POST method. This method packages the name/value pairs inside the body of the HTTP request, which makes for a cleaner URL and imposes no size limitations on the forms output. It is also more secure.

6.1.12 Whats' new, what has changed

Revision 1.8.1

- fix change in kegg (ORG_CODE replaces DEFINITION)

Revision 1.8.0 roadmap

- removed chemspider, picr and clinvtae services due to deprecated services

Revision 1.7.12 (Jan 2021-July 2021)

- continuous integration revisited with github actions
- NEWS: * COG services added * New module: mygeneinfo pdbe * added panther module (pantherdb.org)
- CHANGES * update dbionet to fulfill future new API * Migrate to PDB new API (Jan 2021) * Update Quickgo service * Integration new Biocompare API following Combine/Harmony meeting * Update ChEMBL after an ChEMBL API change
- BUGS and FIXES * General fixes and update from @thobalose (<https://github.com/cokelaer/bioservices/pull/149>) to * PSICQUIC fix (<https://github.com/cokelaer/bioservices/issues/189>) to * Fix ENA new API * Fix NCBIblast and Muscle services (new API)
- DEPRECATED: * deprecated PICR and TCGA modules (the latter was not really available anyway)

Revision 1.7.11 (Dec 2020)

- Fix <https://github.com/cokelaer/bioservices/issues/183> (warning in uniprot)
- Fix annoying warning <https://github.com/cokelaer/bioservices/issues/184>
- Implemented new PDB services (Update API changed in Nov/Dec 2020)

Revision 1.7.10 (Nov 2020)

- Fix KEGG <https://github.com/cokelaer/bioservices/issues/182> adding NETWORK field
- Fix Pathway common with new API <https://github.com/cokelaer/bioservices/issues/135>
- Update the PathwayCommon service with new API
- Update the Rhea service with new API

Revision 1.7.9

- add missing field in KEgg Parser (REL_PATHWAY)
- fix panther.get_enrichment output (try/except)

Revision 1.7.8

- Fix ENA new API
- fix missing plugin in requirements-dev.txt

Revision 1.7.7

- small fix on pantherdb (autocorrect typo in pantherdb api)

Revision 1.7.6

- Fixing ncbiblast and muscle services
- Fixing quickgo services by adding new methods
- Fixing chembl and all failing tests and modules (omicdi, seqret, etc)

Revision 1.7.5

- NEW MODULE: mygeneinfo, pdbe
- Limits the request to 10 per seconds (3 for eutils). This fixes <https://github.com/cokelaer/bioservices/issues/7>
- update quickgo
- Update PDB module (will not be maintained in the future, to usePDB instead.)
- Fix issue in Eutils/ECitMatch reported here <https://github.com/cokelaer/bioservices/issues/169> and here <https://tinyurl.com/y6u2cyjq> on stackoverflow

Revision 1.7.4 (March 2020 Combine/Harmony2020 hackathon)

- Integration of BioModels new API (REST instead of WSDL and different functionalities) in coordination with Mihai Glont and Tung Nguyen at EMBL-EBI during the Combine-Harmony hackathon.
- Integration of BiGG models service thanks to <https://github.com/achillesrasquinha> contribution.
- Fixing notebooks (biomodels and uniprot)
- Move the miriam services to the attic

Revision 1.7.3 (March 2020)

- fixing chembl after API changed. Fix the get_status_resources method by removing document_term and target_prediction. Changed acd_log into alogp: for some reasons this was changed. There was a warning on 4th March 2020 telling that changes may occur.

Revision 1.7.2 (March 2020)

- Fixing ReadTheDocs online documentation and Changelog

Revision 1.7.1 (Feb 2020)

- Just updating the README and setup metadata

Revision 1.7.0 (Feb 2020)

- **Pull request**
 - from @thobalose (<https://github.com/cokelaer/bioservices/pull/149>) to update tests, travis recipes, pinned matplotlib to 3.0.3
- **NEWS:**
 - panther module (pantherdb.org)
 - add a test for the pubchem, pfam and eva modules, which are still in draft version though
- **CHANGES**
 - PICR module is fully commented. The service is most probably deprecated. Not on EBI website anymore
 - remove TCGA, which waws only a draft version with one method.
- **BUGS and FIXES**
 - wikipathway: fixed getPathway, savePathway, getPathwayByLiterature and coloredPathway methods. Some are failing due to some wikipathway temporary failures.
 - Fixed <https://github.com/cokelaer/bioservices/issues/148> to have a more informative error message (array express)
 - Fixed KeggParser for the GENE entries to have the correct ID name. <https://github.com/cokelaer/bioservices/issues/151>

Revision 1.6.0

- **CHANGES:**
 - rewrote entirely the ChEMBL wrapper due to new ChEMBL API.
 - removed the quickgo_old module and its tests
 - Fix typo for a “valid colum,” in uniprot module
 - Changed biomodels WSDL endpoint (thanks to <https://github.com/thobalose>.)
 - uses colorlog to have more robust and consistent logging.
- **BUGS:**
 - Fix wikipathway XML issues by outputing dictionaries now. This fixes <https://github.com/cokelaer/bioservices/issues/131>
 - Fix <https://github.com/cokelaer/bioservices/issues/137> to handle KEGG GENE field properly in KEG- GParse
 - Fix <https://github.com/cokelaer/bioservices/issues/125> thanks to <https://github.com/thobalose>.

Revision 1.5.2

- Fix retmode in EUtils.Efetch fonction. Was not taken into account but set to text by default but this seemed to have changed recently so this bug emerged while it was silent before.
- Issue in EUtils URL (trailing /) fixed in this PR <https://github.com/cokelaer/bioservices/pull/116>
- Major update of Reactome class. The old one is named ReactomeOld and the new one uses the new Reactome API

Revision 1.5.1

Support for Python 2.6 dropped.

- **CHANGES:**
 - using proper logging

Revision 1.5

Support for Python 3.6 on Travis.

- **BUG:**
 - kegg: fix #75 and #77 (missing keywords in KEGG)
 - kegg: fix #79 (mis-interpreted cases reported by kirienko with examples.
 - kegg: fix #85 (some entries are not interpreted)
- **CHANGES:**
 - biobdnet: conversion from WSDL to REST. Note methods' arguments changes: inputValues to input_values, dbPath to db_path. Uses pandas
 - wikipathways: conversion from WSDL to REST. All wikipathways service uses Pandas and returns dataframes.
 - Better implementation of secure host option and more xml customization #98
 - move quickgo.py to quickgo_old.py
 - move readseq to seqret (<https://github.com/cokelaer/bioservices/issues/89>)
 - move wsdbfetch to dbfetch and move from WSDL to REST service
- **NEWS:**
 - quickgo uses the new API from EBI (see changes)
 - seqret uses the new API (instead of readseq)
 - dbfetch uses new API (instead of WSDL)
- Fixes the licensing (GPLv3 everywhere)

For developers: use pytest instead of nosetests.

Revision 1.4

- **1.4.17: rhea URL changed and add get_metabolites function. Fix pride test and add missing license file**
- 1.4.16: simplify setup
- **1.4.15:**
 - **BUG:**
 - * ensembl.org in biomart was not reachable anymore. This is fixed by using requests to check URL existence.
 - * in ensembl module tolist -> to_list
 - * Fix ensembl tests
- **1.4.14:**
 - **CHANGES:**
 - * update http to https in EUtils
 - * missing TARGET field in KEGGParser reported in issue #66
- **1.4.13:**
 - **NEWS:**
 - * Add a download_fasta dedicated function to download a fasta file either from ENA or NCBI given its accession. See bioservices.apps.download_fasta. Used within Sequana project
- **1.4.12:**
 - **BUG:**
 - * Fix a regression bug in ncbiblast introduced in earlier commits <https://github.com/cokelaer/bioservices/issues/61>
 - **CHANGES:**
 - * add PRODUCT/ALL_REAC/HISTORY/SYSNAME in KEGG parser thanks to issue reported in <https://github.com/cokelaer/bioservices/issues/60>
- **1.4.11:**
 - **NEWS:**
 - * EUtils can now return a dictionary rather than a xml
 - * New method get_taxon in ENA class
 - * EnsemblFTP added to ensembl module
- 1.4.10: fixing a bug/typo in pypi
- **1.4.9:**
 - **BUG**
 - * KeggParser missing parser for the SEQUENCE keyword is now available <https://github.com/cokelaer/bioservices/issues/46> , <https://github.com/cokelaer/bioservices/issues/51>
 - **CHANGES:**
 - * Improves way biomart handles errors (see <https://github.com/cokelaer/bioservices/issues/50>)
- **1.4.8:**

- NEW: add new module for the omnipath web service in *bioservices.omnipath*.
- **1.4.7:**
 - NEWS: add method `get_run` in `RNASEQ_EBI` class.
- **1.4.6:**
 - **NEWS:**
 - * RNASEQ analysis REST API included (<http://www.ebi.ac.uk/~rpetry/geteam/rnaseq/apispec.pdf>)
- **1.4.5:**
 - **BUG:**
 - * Fixes a python3 wrong import
- **1.4.4:**
 - **CHANGES:**
 - * Uniprot: update valid columns
 - * <https://github.com/cokelaer/bioservices/pull/35> with biocarta module updates
 - **BUGS:**
 - * Fix a test in `test_utils`
 - * Fix KEGG parser <https://github.com/cokelaer/bioservices/pull/35>
 - * Fix Service input py2/3 compat and unset argument <https://github.com/cokelaer/bioservices/pull/35>
 - * Update biocarta: the website has changed and the code needed to be updated
 - NEWS: ENA module and class added
- **1.4.3**
 - BUG: fix typo in a draft tcga module
- **1.4.2**
 - CHANGES: update setup dependencies.
 - BUG: Typo fixed in uniprot list of valid columns #47
- **1.4.1**
 - **CHANGES:**
 - * Renamed `kegg.KEGG.info` into `dbinfo` , which was overloaded with Logging
 - * Updated all documentation to check examples
 - * Fixed tests and notebooks
 - * clean and tested doctests in the documentation
 - **NEWS:**
 - * Replace deprecated HGNC with the official web service from genenames.org
- **1.4.0**
 - **CHANGES:**

- * Fully update EUtils since WSDL is now down; implementation uses REST now. This fixes <https://github.com/cokelaer/bioservices/issues/41>
- * Remove the apps/taxonomy module now part of biokit.

– **NEWS:**

- * add small XML tools to parse XML dynamically in xmltools module
- * add http_delete in services.py

Revision 1.3

- 1.3.8 (progress)
 - CHANGES:
 - * cache files are now stored in the ./config/bioservices directory, this fixes <https://github.com/cokelaer/bioservices/issues/40>
- 1.3.7
 - CHANGES
 - * ArrayExpress: add new 2 methods to ease the usage
 - BUG FIXES
 - * KEGG: fix <https://github.com/cokelaer/bioservices/issues/39>
- 1.3.6
 - BUG FIXES
 - * **KEGG: Fixed during the major changes described here below**
<https://github.com/cokelaer/bioservices/issues/29>
 - CHANGES
 - * IntactL rename Intact class into IntactComplex
 - * KEGG: revisited the parsing following requests from user <https://github.com/cokelaer/bioservices/issues/30>
 - * KEGG: remove useless function (check_dbentries)
 - * **KEGG: The KEGGParser does not inherit from KEGG anymore and there is**
now a parse() method inside KEGG so user do not need to play with the 2 classes. Only KEGG is required. KEGGParser can still be used but will not have the KEGG methods anymore
- 1.3.5
 - BUG FIXES:
 - * quickgo: fix bug <https://github.com/cokelaer/bioservices/issues/22>
 - * uniprot: add missing columns (<https://github.com/cokelaer/bioservices/issues/23>)
 - * kegg: fix parser related to reaction in the Compound data structure (<https://github.com/cokelaer/bioservices/issues/27>)
 - NEWS
 - * add Intact complex web services
- 1.3.4

- BUG FIXES
- CHANGES * clinivtae: tests and doc added * services modules: DevTools class moved to easydev
- NEWS
 - * add PRIDE service + test + doc
- 1.3.3
 - BUG FIX
 - * uniprot fixing a python 3 typo
 - CHANGES
 - * pdb: add a method
 - * hgnc: add new class related to HGNC
 - NEWS
 - * services.py: add a method to ease conversion of dict to json. add attribute to limit number of requests per seconds but not yet used.
 - * taxonomy module: add new method in Taxon to look for a taxon identifier given a name
 - * NEW module ensembl completed
 - * NEW module clinivtae added (contribution from Patrick Short)
- 1.3.2
 - CHANGES:
 - * services: http_get and http_post now accepts all optional arguments from requests.
 - * services: get_headers default content is now same as urllib2
 - * pdb module: more functions added
 - * ensembl module added with some functionalities
- 1.3.1
 - CHANGES:
 - * uniprot: multi_mapping is deprecated. mapping can now handle long queries by itself.
 - * services/settings:
 - removed get_bioservices_env function, which is not used anymore
 - move urlencode in Service class into WSDLService, which will be deprecated
 - add TIMEOUT in WSDLService and REST as alias to settings.TIMEOUT so timeout can now be used in both REST and WSDL.
 - NEWS:
 - * readseq module added.
 - BUG fixes:
 - * CACHING attribute had a typo
- 1.3.0
 - NEWS

- * added REST class that uses the requests module. This class replaces of instance of RESTservice that uses urllib2, which will be deprecated later on. This speeds up the code significantly not only because requests is faster but also because we now do not need trial/time hack that was implemented inside RESTService. We also use the requests_cache module that could be used to speed go but requires to store cache files locally. Asynchronous requests is available but used only in a few places for now.
- * EUtils has been fully implemented excepting EPost. API may still change to make its usage easier but functionalities are there.
- CHANGES
 - * update code to be python-3 compatible. There are still issues with suds/requests/gevent but the code itself is python3 executable.
 - * WSDLService now uses suds instead of SOAP package by default
 - * all paramters called format have been renamed frmt (format is a python keyword)
 - * chemblpdb module and class renamed to chembl and *bioservices.chembl.ChEMBL*
 - * All classes that depends on RESTService have been updated to use the new REST class.
 - * chemblpdb:
 - get_assay_by_chemblId renamed in get_assays_by_chemblId
 - renamed get_target_by_refSeqId into get_target_by_refseq
 - kegg module: all Kegg strings replaced by KEGG so the kegg.Kegg class is now kegg.KEGG
 - * ChEBI: getUpdatedPolymer: remove useless parameters (was failing with python3)
 - * Wikipathway class renamed as WikiPathways to agree with official name
 - * biomart now uses python3 and we had to remove the threaded_request module, which does not seem to be available. So, we used the new implementation using requests but gevent is not available for python3 either so, we use requests but without the asynchronous call. This is working for now. Transparent for the user.
 - * geneprof: parameter called type and format are renamed output and frmt to not clash with python keywords. Use REST class instead of RESTService but should be transparent for the users.
 - * services do not have the checkParam method. use devtools.check_param_in_list instead.
- BUG FIXES:
 - * Fixing bug #24/25 posted on assembla related to parse_kgml_pathway second argument can now be used.
 - * wikipathway: findInteractions had a typo in i

Revision 1.2

- 1.2.6:
 - fixing bug report 22 related to KEGG.pathway2sif function that was failing.
 - add option in biomart to use different host. This is to fix an issue where biomart hangs forever. This was reported by Daniel D bug report 23 on assembla.
- 1.2.5:
 - add try/except for pandas library.
- 1.2.4:

- fixing typo in the init that fails bioservices to be used if pkg_resources is not available.
- 1.2.3
 - **updating some apps (fasta,peptides, taxon) in bioservices.apps directory**
 - * Improves UniProt module by adding a dataframe export when performing a search
 - * added the BioDBnet service.
 - * added Pathway Commons
 - * fixed UniChem: add new database identifiers and fix interpretation of the output
- 1.2.2:
 - NEW Service: *bioservices.biodbnet.BioDBNet*
 - uniprot: add multi_mapping method to use mapping method on large queries and added timeout/trials inside uniprot functions
- 1.2.1:
 - same as 1.2.0 but fixed missing mapping and apps directory in the distribution available on pypi
- 1.2.0
 - Kegg class has now an alias called KEGG
 - NEW Services: *bioservices.muscle.MUSCLE*
 - fix bug in get_fasta from uniprot class
 - add aliases to quickGO to retrieve annotation
 - NEW Service: *bioservices.pathwaycommons.PathwayCommons*
 - NEW Service: *bioservices.geneprof.GeneProf* service
 - uniprot add function to get uniprot fasta sequence
 - add apps.peptides module

Revision 1.1

- 1.1.3
 - **fix bug in chembl.get_all_targets() that was failing to return the json/dictionary as expected.**
- 1.1.2
 - **add biocarta, pfam modules (and htmltools. maybe not required.)**
 - * fix bug in uniprot.mapping to return list of values instead of a string (assembla ticket 19).
- 1.1.1:
 - **services.py: move print statements into login.warning**
 - * add documentation and examples related to Galaxy/BioPython.
 - uniprot mapping function now returns a dictionary instead of a list
 - NEW Service : class:*bioservices.hgnc.HGNC* + doc + test

Revision 1.1

- **1.1.0:**
 - **in psicquic when performing the conversion, we now use a try/except since some fields (in rare case) may be missing**
 - * add faqs in the doc + update of the README and metadata.
 - * fix typo in the list of uniprot mapping
 - * Use BeautifulSoup4 instead of 3
 - * add the ChEBI Web Service.
 - * add the UniChem Web Service.
 - * logging ERROR in Service when data cannot be converted to XML is now a simple warning
 - * kegg.conv method now returns a dictionary instead of list of tuples.

Revision 1.0

- **1.0.4**
 - add a draft version of PDB just to be able to fetch PDB data and use it with external tool such as PyMOL as shown in the new pymol.rst documentation.
 - add a missing docstring in uniprot + check to/fr parameters in UniProt.mapping method.
 - Fix a typo in PSICQUIC module.
- **1.0.3**
 - **uniprot.UniPort.search method: default value of the parameter format is now “tab”**
 - * fix 1 quickgo test
 - * a few documentation updates in biomart/uniprot/chemblpdb and tutorial.
- **1.0.2:**
 - **add SOAPpy in the setup requirements**
 - * finished ArrayExpress +doc + tests
 - * fixed a bug in KEGGParser.parseGene
 - * add methods in psicquic to parse all databases and convert to uniprot if possible. These methods are used to build an application provided in the tutorial
 - add biomart + doc + test
 - add onWeb method in Service class
 - **add chemspider draft**
 - * complete eutils
- **1.0.1**
 - Add miriam module
 - Add arrayexpress
- **1.0.0:**

- First release of bioservices

Revision 0.9

- **0.9.7:**
 - **add new feature in KEgg module to introspect kgml data sets**
 - * add biogrid test and documentation.
 - * chemblpdb improvements
 - * uniprot bug fixes (search if working as expected now)
- **0.9.6:**
 - Finalising the Kegg module
- **0.9.5:**
 - **add parser for all KEGG entries (enzyme, genome, pathway, ...)**
 - * add a show_pathway to highlight element in a pathway
- **0.9.4:**
 - cleaning up the modules
- **0.9.3:**
 - documentation cleanup
 - fix tests
 - fix a few small bugs in biomodels
 - adding getattr method for all databases in kegg model
 - Service class has new method call pubmed to load pubmed in browser
- **0.9.2:**
 - uniprot search method improved
- 0.9.1: fix typo in biomodel. add uniprot search method. add keggParser class
- **0.9.0: Stable version of bioservices including the following services:**
 BioModels, Kegg, Reactome, Chembl, PICR, QuickGO, Rhea, UniProt, WSDbfetch, NCBIblast, PSIC-QUIC, Wikipath

Up to Revision 0.5

- 0.4.9: finalise wikipathway
- 0.4.8: finalise doc of half of the services.
- 0.4.7: add psiquic service and carry on reactome
- 0.4.6: finalise kegg module and test
- 0.4.5: finalise biomodels. keff WSDL is not maintained anymore: started REST version.
- 0.4.4: finalise quickgo, rhea, picr, uniprot. Update service to use logging module.
- 0.4.3: add quickgo

- 0.4.2: add wsdbfetch/uniprot
- 0.4.1: add wikipathways module +test .
- 0.4.0: add reea service + test. Updating the documentation.
- 0.3.0: add reactome + uniprot.
- 0.2.0: finalise biomodels and add picr service + test for biomodel service..
- 0.1.0: add database and kegg modules + its documentation and tests

6.1.13 Contributors

Contributors are the authors who started the development of BioServices (and authors of this reference on [BioInformatics](#)).

In addition to the main authors of the papers the following developers have implemented modules now available in BioServices:

- Achilles Rasquinha implemented the BiGG models service *bioservices.bigg* module
- Sven-Maurice Althoff, Christian Knauth implemented the *bioservices.muscle* module.
- Patrick Short implemented the *bioservices.clinitae* module

And thank you also to the contributions from users who have sent communication via emails or via the [ticket system](#).

Special thanks to Thoba Lose (<https://github.com/thobalose>) and <https://github.com/jsmusach> for various pull requests.

Note that originally code (and earlier tickets) were hosted [elsewhere](#).

PYTHON MODULE INDEX

a

`bioservices.apps.fasta`, 198
`bioservices.apps.peptides`, 198

b

`bioservices.bigg`, 60
`bioservices.biocontainers`, 59
`bioservices.biodbnet`, 61
`bioservices.biogrid`, 64
`bioservices.biomart`, 65
`bioservices.biomodels`, 69

c

`bioservices.chebi`, 74
`bioservices.chembl`, 77
`bioservices.cog`, 88

d

`bioservices.dbfetch`, 188

e

`bioservices.ena`, 89
`bioservices.eutils`, 91

h

`bioservices.hgnc`, 114

i

`bioservices.intact`, 118

k

`bioservices.kegg`, 101

m

`bioservices.muscle`, 120
`bioservices.mygeneinfo`, 124

n

`bioservices.ncbiblast`, 127

o

`bioservices.omnipath`, 131

p

`bioservices.panther`, 133
`bioservices.pathwaycommons`, 136
`bioservices.pdb`, 142
`bioservices.pdbe`, 148
`bioservices.pride`, 152
`bioservices.psicquic`, 158

q

`bioservices.quickgo`, 98

r

`bioservices.reactome`, 167
`bioservices.rhea`, 164

s

`bioservices.seqret`, 177
`bioservices.services`, 54

u

`bioservices.unichem`, 179
`bioservices.uniprot`, 184

w

`bioservices.wikipathway`, 190

x

`bioservices.xmltools`, 58

A

accession (*FASTA property*), 200
 activeDBs (*PSICQUIC property*), 161
 add_attribute_to_xml() (*BioMart method*), 67
 add_dataset_to_xml() (*BioMart method*), 67
 add_filter_to_xml() (*BioMart method*), 67
 Annotation() (*QuickGO method*), 98
 Annotation_from_goid() (*QuickGO method*), 100
 attributes() (*BioMart method*), 67

B

BiGG (*class in bioservices.bigg*), 60
 Biocontainers (*class in bioservices.biocontainers*), 60
 BioDBNet (*class in bioservices.biodbnet*), 61
 BioGRID (*class in bioservices.biogrid*), 64
 BioMart (*class in bioservices.biomart*), 65
 BioModels (*class in bioservices.biomodels*), 70
 biopax_exporter() (*ReactomeOld method*), 173
 bioservices.apps.fasta
 module, 198
 bioservices.apps.peptides
 module, 198
 bioservices.bigg
 module, 60
 bioservices.biocontainers
 module, 59
 bioservices.biodbnet
 module, 61
 bioservices.biogrid
 module, 64
 bioservices.biomart
 module, 65
 bioservices.biomodels
 module, 69
 bioservices.chebi
 module, 74
 bioservices.chembl
 module, 77
 bioservices.cog
 module, 88
 bioservices.dbfetch
 module, 188
 bioservices.ena
 module, 89
 bioservices.eutils
 module, 91
 bioservices.hgnc
 module, 114
 bioservices.intact
 module, 118
 bioservices.kegg
 module, 101
 bioservices.muscle
 module, 120
 bioservices.mygeneinfo
 module, 124
 bioservices.ncbiblast
 module, 127
 bioservices.omnipath
 module, 131
 bioservices.panther
 module, 133
 bioservices.pathwaycommons
 module, 136
 bioservices.pdb
 module, 142
 bioservices.pdbe
 module, 148
 bioservices.pride
 module, 152
 bioservices.psiquic
 module, 158
 bioservices.quickgo
 module, 98
 bioservices.reactome
 module, 167
 bioservices.rhea
 module, 164
 bioservices.seqret
 module, 177
 bioservices.services
 module, 54
 bioservices.unichem
 module, 179

bioservices.uniprot
 module, 184
bioservices.wikipathway
 module, 190
bioservices.xmltools
 module, 58
bioservices.get_reactants_from_reaction_identification()
 (ReactomeOld method), 173
BioServicesError, 54
briteIds (KEGG property), 104

C

CACHING (Service property), 57
ChEBI (class in bioservices.chebi), 74
ChEMBL (class in bioservices.chembl), 77
clear_cache() (REST method), 55
code2Tnumber() (KEGG method), 104
COG (class in bioservices.cog), 88
compoundIds (KEGG property), 104
compounds2accession() (ChEMBL method), 82
configuration() (BioMart method), 67
content_types (REST attribute), 55
conv() (ChEBI method), 74
conv() (KEGG method), 104
convert() (PSICQUIC method), 161
convertAll() (PSICQUIC method), 162
create_attribute() (BioMart method), 67
create_filter() (BioMart method), 68
createPathway() (WikiPathways method), 191
custom_query() (BioMart method), 68

D

data_warehouse() (ENA method), 90
databases (BioMart property), 68
databases (EUtils property), 97
databases (KEGG property), 105
databases (NCBIblast property), 128
datasets() (BioMart method), 68
db2db() (BioDBNet method), 62
DBFetch (class in bioservices.dbfetch), 188
dbFind() (BioDBNet method), 62
dbinfo() (KEGG method), 105
dbOrtho() (BioDBNet method), 62
dbReport() (BioDBNet method), 63
dbtype (FASTA property), 200
dbWalk() (BioDBNet method), 63
debug_message() (REST method), 55
default_extension (PathwayCommons property), 137
delete_cache() (REST method), 55
delete_one() (REST method), 55
details() (IntactComplex method), 119
df (FASTA property), 200
df (MultiFASTA property), 202
displayNames (BioMart property), 68

download() (BiGG method), 61
drugIds (KEGG property), 106

E

easyXML (class in bioservices.xmltools), 58
easyXML() (Service method), 57
easyXMLConversion (Service property), 57
ECitMatch() (EUtils method), 92
EFetch() (EUtils method), 92
EGQuery() (EUtils method), 94
EInfo() (EUtils method), 94
ELink() (EUtils method), 94
email (EUtils attribute), 97
ENA (class in bioservices.ena), 90
entry (FASTA property), 200
entry() (KEGG method), 106
enzymeIds (KEGG property), 106
EPost() (EUtils method), 95
ESearch() (EUtils method), 95
ESpell() (EUtils method), 96
ESummary() (EUtils method), 96
EUtils (class in bioservices.eutils), 91
EUtilsParser (class in bioservices.eutils), 97

F

FASTA (class in bioservices.apps.fasta), 198
fasta (FASTA property), 200
fasta (MultiFASTA property), 202
fetch() (DBFetch method), 188
fetch() (HGNC method), 114
filters() (BioMart method), 68
find() (KEGG method), 106
findInteractions() (WikiPathways method), 191
findPathwaysByLiterature() (WikiPathways method), 191
findPathwaysByText() (WikiPathways method), 191
findPathwaysByXref() (WikiPathways method), 192
formats (PSICQUIC property), 162
front_page_items() (ReactomeOld method), 174

G

gene_name (FASTA property), 200
gene_product_search() (QuickGO method), 100
genes() (BiGG method), 61
get() (KEGG method), 107
get() (PathwayCommons method), 137
get_about() (OmniPath method), 132
get_activity() (ChEMBL method), 83
get_aliases() (HGNCDeprecated method), 116
get_all_cogs_definition() (COG method), 88
get_all_database_info() (DBFetch method), 189
get_all_models() (BioModels method), 70
get_all_names() (HGNCDeprecated method), 116
get_all_reactions() (ReactomeOld method), 174

get_all_src_ids() (*UniChem method*), 180
 get_annotation_datasets() (*Panther method*), 134
 get_approved_drugs() (*ChEMBL method*), 83
 get_assay() (*ChEMBL method*), 83
 get_assay_count() (*PRIDE method*), 153
 get_assay_list() (*PRIDE method*), 153
 get_assays() (*PRIDE method*), 153
 get_assembly() (*PDBe method*), 148
 get_async() (*REST method*), 56
 get_ATC() (*ChEMBL method*), 83
 get_binding_site() (*ChEMBL method*), 83
 get_binding_sites() (*PDBe method*), 148
 get_biotherapeutic() (*ChEMBL method*), 83
 get_cell_line() (*ChEMBL method*), 83
 get_chembl_id_lookup() (*ChEMBL method*), 83
 get_chromosome() (*HGNCDeprecated method*), 116
 get_cog_definition_by_cog_id() (*COG method*), 88
 get_cog_definition_by_name() (*COG method*), 88
 get_cogs() (*COG method*), 89
 get_cogs_by_assembly_id() (*COG method*), 89
 get_cogs_by_category() (*COG method*), 89
 get_cogs_by_category_() (*COG method*), 89
 get_cogs_by_category_id() (*COG method*), 89
 get_cogs_by_gene() (*COG method*), 89
 get_cogs_by_id() (*COG method*), 89
 get_cogs_by_id_and_category() (*COG method*), 89
 get_cogs_by_id_and_organism() (*COG method*), 89
 get_cogs_by_organism() (*COG method*), 89
 get_cogs_by_taxon_id() (*COG method*), 89
 get_complex_subunits() (*Reactome method*), 167
 get_complexes() (*Reactome method*), 167
 get_compound_record() (*ChEMBL method*), 83
 get_compound_structural_alert() (*ChEMBL method*), 83
 get_compounds() (*UniChem method*), 180
 get_connectivity() (*UniChem method*), 181
 get_current_ids() (*PDB method*), 146
 get_data() (*ENA method*), 90
 get_database_format_styles() (*DBFetch method*), 189
 get_database_formats() (*DBFetch method*), 189
 get_database_info() (*DBFetch method*), 189
 get_datasets() (*BioMart method*), 68
 get_df() (*UniProt method*), 184
 get_discover() (*Reactome method*), 167
 get_diseases() (*Reactome method*), 168
 get_diseases_doid() (*Reactome method*), 168
 get_document() (*ChEMBL method*), 83
 get_document_similarity() (*ChEMBL method*), 83
 get_document_term() (*ChEMBL method*), 84
 get_drug() (*ChEMBL method*), 84
 get_drug_indication() (*ChEMBL method*), 84
 get_drugbank_annotation() (*PDBe method*), 149
 get_electron_density_statistics() (*PDBe method*), 149
 get_enrichment() (*Panther method*), 134
 get_entity_componentOf() (*Reactome method*), 168
 get_entity_otherForms() (*Reactome method*), 168
 get_event_ancestors() (*Reactome method*), 168
 get_eventsHierarchy() (*Reactome method*), 168
 get_experiment() (*PDBe method*), 149
 get_exporter_diagram() (*Reactome method*), 169
 get_exporter_fireworks() (*Reactome method*), 169
 get_exporter_reaction() (*Reactome method*), 169
 get_exporter_sbml() (*Reactome method*), 169
 get_family_msa() (*Panther method*), 134
 get_family_ortholog() (*Panther method*), 134
 get_fasta() (*FASTA method*), 200
 get_fasta() (*UniProt method*), 185
 get_fasta_sequence() (*Peptides method*), 198
 get_file_count() (*PRIDE method*), 153
 get_file_count_assay() (*PRIDE method*), 154
 get_file_list() (*PRIDE method*), 154
 get_file_list_assay() (*PRIDE method*), 154
 get_files() (*PDBe method*), 149
 get_functional_annotation() (*PDBe method*), 149
 get_genes() (*MyGeneInfo method*), 124
 get_go_ancestors() (*QuickGO method*), 100
 get_go_chart() (*QuickGO method*), 100
 get_go_children() (*QuickGO method*), 100
 get_go_paths() (*QuickGO method*), 100
 get_go_slim() (*ChEMBL method*), 84
 get_go_terms() (*QuickGO method*), 100
 get_headers() (*REST method*), 56
 get_homolog_position() (*Panther method*), 135
 get_id_from_name() (*UniChem method*), 181
 get_image() (*ChEMBL method*), 84
 get_images() (*UniChem method*), 181
 get_inchi_from_inchikey() (*UniChem method*), 182
 get_info() (*HGNC method*), 114
 get_info() (*OmniPath method*), 132
 get_interactions() (*OmniPath method*), 132
 get_interactors_psicquic_molecule_details() (*Reactome method*), 169
 get_interactors_psicquic_molecule_summary() (*Reactome method*), 169
 get_interactors_psicquic_resources() (*Reactome method*), 169
 get_interactors_static_molecule_details() (*Reactome method*), 169
 get_interactors_static_molecule_pathways() (*Reactome method*), 170
 get_interactors_static_molecule_summary() (*Reactome method*), 170
 get_ligand_monomers() (*PDBe method*), 149
 get_list_pathways() (*ReactomeOld method*), 174
 get_mapping() (*Panther method*), 135

get_mapping_identifiers_pathways() (*Reactome method*), 170
 get_mapping_identifiers_reactions() (*Reactome method*), 170
 get_mechanism() (*ChEMBL method*), 84
 get_metabolism() (*ChEMBL method*), 84
 get_metabolites() (*Rhea method*), 165
 get_metadata() (*MyGeneInfo method*), 125
 get_model() (*BioModels method*), 70
 get_model_download() (*BioModels method*), 70
 get_model_files() (*BioModels method*), 71
 get_modified_residues() (*PDBe method*), 150
 get_molecule() (*ChEMBL method*), 84
 get_molecule_form() (*ChEMBL method*), 85
 get_molecules() (*PDBe method*), 150
 get_mutated_residues() (*PDBe method*), 150
 get_name() (*HGNCDeprecated method*), 117
 get_network() (*OmniPath method*), 132
 get_nmr_resources() (*PDBe method*), 150
 get_observed_ranges() (*PDBe method*), 150
 get_observed_ranges_in_pdb_chain() (*PDBe method*), 150
 get_observed_residues_ratio() (*PDBe method*), 151
 get_one() (*REST method*), 56
 get_one_gene() (*MyGeneInfo method*), 125
 get_one_query() (*MyGeneInfo method*), 125
 get_organism() (*ChEMBL method*), 85
 get_ortholog() (*Panther method*), 135
 get_p2m_missing() (*BioModels method*), 72
 get_p2m_representative() (*BioModels method*), 72
 get_p2m_representatives() (*BioModels method*), 72
 get_parameter_details() (*MUSCLE method*), 121
 get_parameter_details() (*NCBIblast method*), 128
 get_parameter_details() (*Seqret method*), 177
 get_parameters() (*MUSCLE method*), 121
 get_parameters() (*NCBIblast method*), 128
 get_parameters() (*Seqret method*), 177
 get_pathway_by_gene() (*KEGG method*), 107
 get_pathway_containedEvents() (*Reactome method*), 170
 get_pathway_containedEvents_by_attribute() (*Reactome method*), 170
 get_pathways() (*Panther method*), 135
 get_pathways_low_diagram_entity() (*Reactome method*), 170
 get_pathways_low_diagram_entity_allForms() (*Reactome method*), 170
 get_pathways_low_entity() (*Reactome method*), 170
 get_pathways_low_entity_allForms() (*Reactome method*), 171
 get_pathways_top() (*Reactome method*), 171
 get_pdgsmm_missing() (*BioModels method*), 72
 get_pdgsmm_representative() (*BioModels method*), 72
 get_pdgsmm_representatives() (*BioModels method*), 72
 get_peptide_count() (*PRIDE method*), 154
 get_peptide_count_assay() (*PRIDE method*), 154
 get_peptide_list() (*PRIDE method*), 155
 get_peptide_list_assay() (*PRIDE method*), 155
 get_phosphosite_position() (*Peptides method*), 198
 get_previous_names() (*HGNCDeprecated method*), 117
 get_previous_symbols() (*HGNCDeprecated method*), 117
 get_project() (*PRIDE method*), 155
 get_project_count() (*PRIDE method*), 156
 get_project_list() (*PRIDE method*), 156
 get_protein_class() (*ChEMBL method*), 85
 get_protein_count() (*PRIDE method*), 157
 get_protein_count_assay() (*PRIDE method*), 157
 get_protein_list() (*PRIDE method*), 157
 get_protein_list_assay() (*PRIDE method*), 157
 get_ptms() (*OmniPath method*), 132
 get_queries() (*MyGeneInfo method*), 126
 get_references() (*Reactome method*), 171
 get_related_dataset() (*PDBe method*), 151
 get_related_publications() (*PDBe method*), 151
 get_release_status() (*PDBe method*), 151
 get_residue_listing() (*PDBe method*), 151
 get_residue_listing_in_pdb_chain() (*PDBe method*), 151
 get_resources() (*OmniPath method*), 132
 get_result() (*MUSCLE method*), 122
 get_result() (*NCBIblast method*), 128
 get_result() (*Seqret method*), 177
 get_result_types() (*MUSCLE method*), 122
 get_result_types() (*NCBIblast method*), 129
 get_result_types() (*Seqret method*), 178
 get_secondary_structure() (*PDBe method*), 152
 get_sifgraph_common_stream() (*PathwayCommons method*), 137
 get_sifgraph_neighborhood() (*PathwayCommons method*), 138
 get_sifgraph_pathsbetween() (*PathwayCommons method*), 138
 get_sifgraph_pathsfromto() (*PathwayCommons method*), 138
 get_similarity() (*ChEMBL method*), 85
 get_similarity_sequence() (*PDB method*), 146
 get_source() (*ChEMBL method*), 86
 get_source_info_by_id() (*UniChem method*), 182
 get_source_info_by_name() (*UniChem method*), 182
 get_sources() (*UniChem method*), 183
 get_sources_by_inchikey() (*UniChem method*), 183

- get_sources_by_inchikey_verbose() (*UniChem method*), 183
- get_species() (*ReactomeOld method*), 174
- get_species_all() (*Reactome method*), 171
- get_species_main() (*Reactome method*), 171
- get_stats() (*Biocontainers method*), 60
- get_status() (*ChEMBL method*), 86
- get_status() (*MUSCLE method*), 122
- get_status() (*NCBIblast method*), 129
- get_status() (*Seqret method*), 178
- get_status_resources() (*ChEMBL method*), 86
- get_structure() (*UniChem method*), 183
- get_substructure() (*ChEMBL method*), 86
- get_summary() (*PDB method*), 152
- get_supported_families() (*Panther method*), 136
- get_supported_genomes() (*Panther method*), 136
- get_sync() (*REST method*), 56
- get_target() (*ChEMBL method*), 87
- get_target_component() (*ChEMBL method*), 87
- get_target_prediction() (*ChEMBL method*), 87
- get_target_relation() (*ChEMBL method*), 87
- get_taxon() (*ENA method*), 91
- get_taxon_id() (*Panther method*), 136
- get_taxonomic_categories() (*COG method*), 89
- get_taxonomic_category_by_name() (*COG method*), 89
- get_taxonomy() (*MyGeneInfo method*), 126
- get_tissue() (*ChEMBL method*), 87
- get_tools() (*Biocontainers method*), 60
- get_tree_info() (*Panther method*), 136
- get_versions_one_tool() (*Biocontainers method*), 60
- get_withdrawn_symbols() (*HGNCDeprecated method*), 117
- get_xml() (*BioMart method*), 68
- get_xml() (*HGNCDeprecated method*), 117
- get_xref_source() (*ChEMBL method*), 87
- get_xrefs() (*HGNCDeprecated method*), 117
- getAllOntologyChildrenInPath() (*ChEBI method*), 74
- getchildren() (*easyXML method*), 59
- getColoredPathway() (*WikiPathways method*), 192
- getCompleteEntity() (*ChEBI method*), 75
- getCompleteEntityByList() (*ChEBI method*), 75
- getCurationTags() (*WikiPathways method*), 193
- getCurationTagsByName() (*WikiPathways method*), 193
- getDirectOutputsForInput() (*BioDBNet method*), 64
- getInputs() (*BioDBNet method*), 64
- getInteractionCounter() (*PSICQUIC method*), 162
- getLiteEntity() (*ChEBI method*), 75
- getName() (*PSICQUIC method*), 162
- getOntologyChildren() (*ChEBI method*), 76
- getOntologyParents() (*ChEBI method*), 76
- getOntologyTermsByPathway() (*WikiPathways method*), 193
- getOutputsForInput() (*BioDBNet method*), 64
- getPathway() (*WikiPathways method*), 193
- getPathwayAs() (*WikiPathways method*), 193
- getPathwayHistory() (*WikiPathways method*), 194
- getPathwayInfo() (*WikiPathways method*), 194
- getPathwaysByOntologyTerm() (*WikiPathways method*), 194
- getPathwaysByParentOntologyTerm() (*WikiPathways method*), 194
- getRecentChanges() (*WikiPathways method*), 195
- getStructureSearch() (*ChEBI method*), 76
- getUpdatedPolymer() (*ChEBI method*), 76
- getUserAgent() (*REST method*), 56
- glycanIds (*KEGG property*), 108
- go_search() (*QuickGO method*), 100
- graph() (*PathwayCommons method*), 139
- ## H
- header (*FASTA property*), 200
- help() (*EUtils method*), 97
- HGNC (*class in bioservices.hgnc*), 114
- HGNCDeprecated (*class in bioservices.hgnc*), 115
- highlight_pathway_diagram() (*ReactomeOld method*), 174
- hist_size() (*MultiFASTA method*), 202
- host (*BioMart property*), 68
- hosts (*BioMart property*), 68
- http_delete() (*REST method*), 56
- http_get() (*REST method*), 56
- http_post() (*REST method*), 56
- ## I
- identifier (*FASTA property*), 200
- identifiers() (*ReactomeAnalysis method*), 172
- ids (*MultiFASTA property*), 202
- IntactComplex (*class in bioservices.intact*), 118
- isOrganism() (*KEGG method*), 108
- ## K
- KEGG (*class in bioservices.kegg*), 103
- KEGGParser (*class in bioservices.kegg*), 112
- known_dbtypes (*FASTA attribute*), 200
- knownName() (*PSICQUIC method*), 162
- koIds (*KEGG property*), 108
- ## L
- link() (*KEGG method*), 108
- list() (*KEGG method*), 108
- list_by_query() (*ReactomeOld method*), 174
- listOrganisms() (*WikiPathways method*), 195

listPathways() (*WikiPathways method*), 195
load() (*FASTA method*), 200
load_fasta() (*FASTA method*), 200
load_fasta() (*MultiFASTA method*), 202
login() (*WikiPathways method*), 195
lookfor() (*BioMart method*), 68
lookfor() (*HGNCDeprecated method*), 117
lookfor_organism() (*KEGG method*), 109
lookfor_pathway() (*KEGG method*), 109

M

mapping() (*HGNCDeprecated method*), 118
mapping() (*UniProt method*), 185
mapping_all() (*HGNCDeprecated method*), 118
mappingOneDB() (*PSICQUIC method*), 162
marts (*BioMart property*), 68
metabolites() (*BiGG method*), 61
models (*BiGG property*), 61
module

- bioservices.apps.fasta, 198
- bioservices.apps.peptides, 198
- bioservices.bigg, 60
- bioservices.biocontainers, 59
- bioservices.biobdbnet, 61
- bioservices.biogrid, 64
- bioservices.biomart, 65
- bioservices.biomodels, 69
- bioservices.chebi, 74
- bioservices.chembl, 77
- bioservices.cog, 88
- bioservices.dbfetch, 188
- bioservices.ena, 89
- bioservices.eutils, 91
- bioservices.hgnc, 114
- bioservices.intact, 118
- bioservices.kegg, 101
- bioservices.muscle, 120
- bioservices.mygeneinfo, 124
- bioservices.ncbiblast, 127
- bioservices.omnipath, 131
- bioservices.panther, 133
- bioservices.pathwaycommons, 136
- bioservices.pdb, 142
- bioservices.pdbe, 148
- bioservices.pride, 152
- bioservices.psicquic, 158
- bioservices.quickgo, 98
- bioservices.reactome, 167
- bioservices.rhea, 164
- bioservices.seqret, 177
- bioservices.services, 54
- bioservices.unichem, 179
- bioservices.uniprot, 184
- bioservices.wikipathway, 190

- bioservices.xmltools, 58
- moduleIds (*KEGG property*), 110
- MultiFASTA (*class in bioservices.apps.fasta*), 201
- MUSCLE (*class in bioservices.muscle*), 121
- MyGeneInfo (*class in bioservices.mygeneinfo*), 124

N

name (*FASTA property*), 200
name (*Reactome property*), 171
names (*BioMart property*), 68
NCBIblast (*class in bioservices.ncbiblast*), 127
new_query() (*BioMart method*), 68

O

OmniPath (*class in bioservices.omnipath*), 131
on_web() (*Service method*), 57
order_by() (*ChEMBL method*), 87
organism (*FASTA property*), 200
organism (*KEGG property*), 110
organism (*WikiPathways property*), 195
organismIds (*KEGG property*), 110
organisms (*WikiPathways attribute*), 195
organismTnumbers (*KEGG property*), 110

P

Panther (*class in bioservices.panther*), 133
parameters (*MUSCLE property*), 122
parameters (*NCBIblast property*), 129
parameters (*Seqret property*), 178
parse() (*KEGG method*), 110
parse() (*KEGGParser method*), 113
parse_kgml_pathway() (*KEGG method*), 110
parse_xml() (*EUutils method*), 97
pathway2sif() (*KEGG method*), 111
pathway_complexes() (*ReactomeOld method*), 175
pathway_diagram() (*ReactomeOld method*), 175
pathway_hierarchy() (*ReactomeOld method*), 175
pathway_participants() (*ReactomeOld method*), 176
PathwayCommons (*class in bioser-*
vices.pathwaycommons), 136
pathwayIds (*KEGG property*), 111
PDB (*class in bioservices.pdb*), 142
PDBe (*class in bioservices.pdbe*), 148
PE (*FASTA property*), 199
Peptides (*class in bioservices.apps.peptides*), 198
post_one() (*REST method*), 56
postCleaning() (*PSICQUIC method*), 162
postCleaningAll() (*PSICQUIC method*), 162
preCleaning() (*PSICQUIC method*), 162
PRIDE (*class in bioservices.pride*), 152
print_status() (*PSICQUIC method*), 162
PSICQUIC (*class in bioservices.psicquic*), 160
pubmed() (*Service method*), 57

Q

[query\(\)](#) (*BioMart method*), 68
[query\(\)](#) (*PSICQUIC method*), 163
[query\(\)](#) (*Rhea method*), 166
[query_by_id\(\)](#) (*ReactomeOld method*), 176
[query_by_ids\(\)](#) (*ReactomeOld method*), 176
[query_hit_pathways\(\)](#) (*ReactomeOld method*), 176
[query_pathway_for_entities\(\)](#) (*ReactomeOld method*), 176
[queryAll\(\)](#) (*PSICQUIC method*), 163
[quick_search\(\)](#) (*UniProt method*), 186
[QuickGO](#) (*class in bioservices.quickgo*), 98

R

[reactionIds](#) (*KEGG property*), 111
[reactions\(\)](#) (*BiGG method*), 61
[Reactome](#) (*class in bioservices.reactome*), 167
[ReactomeAnalysis](#) (*class in bioservices.reactome*), 172
[ReactomeOld](#) (*class in bioservices.reactome*), 172
[read_fasta\(\)](#) (*FASTA method*), 200
[read_fasta\(\)](#) (*MultiFASTA method*), 202
[read_registry\(\)](#) (*PSICQUIC method*), 164
[readXML](#) (*class in bioservices.xmltools*), 59
[registry](#) (*PSICQUIC property*), 164
[registry\(\)](#) (*BioMart method*), 69
[registry_actives](#) (*PSICQUIC property*), 164
[registry_counts](#) (*PSICQUIC property*), 164
[registry_names](#) (*PSICQUIC property*), 164
[registry_restexamples](#) (*PSICQUIC property*), 164
[registry_restricted](#) (*PSICQUIC property*), 164
[registry_resturls](#) (*PSICQUIC property*), 164
[registry_soapurls](#) (*PSICQUIC property*), 164
[registry_versions](#) (*PSICQUIC property*), 164
[removeCurationTag\(\)](#) (*WikiPathways method*), 195
[response_codes](#) (*Service attribute*), 57
[REST](#) (*class in bioservices.services*), 54
[retrieve\(\)](#) (*UniProt method*), 186
[Rhea](#) (*class in bioservices.rhea*), 164
[run\(\)](#) (*MUSCLE method*), 122
[run\(\)](#) (*NCBIblast method*), 129
[run\(\)](#) (*Seqret method*), 178

S

[save_fasta\(\)](#) (*FASTA method*), 201
[save_fasta\(\)](#) (*MultiFASTA method*), 202
[save_pathway\(\)](#) (*KEGG method*), 111
[save_str_to_image\(\)](#) (*Service method*), 57
[saveCurationTag\(\)](#) (*WikiPathways method*), 197
[savePathwayAs\(\)](#) (*WikiPathways method*), 197
[SBML_exporter\(\)](#) (*ReactomeOld method*), 173
[search\(\)](#) (*BiGG method*), 61
[search\(\)](#) (*BioModels method*), 73
[search\(\)](#) (*HGNC method*), 114

[search\(\)](#) (*IntactComplex method*), 119
[search\(\)](#) (*PathwayCommons method*), 140
[search\(\)](#) (*PDB method*), 146
[search\(\)](#) (*Rhea method*), 166
[search\(\)](#) (*UniProt method*), 186
[search_activity\(\)](#) (*ChEMBL method*), 88
[search_assay\(\)](#) (*ChEMBL method*), 88
[search_chembl_id_lookup\(\)](#) (*ChEMBL method*), 88
[search_document\(\)](#) (*ChEMBL method*), 88
[search_download\(\)](#) (*BioModels method*), 73
[search_facet\(\)](#) (*Reactome method*), 171
[search_facet_query\(\)](#) (*Reactome method*), 171
[search_molecule\(\)](#) (*ChEMBL method*), 88
[search_parameter\(\)](#) (*BioModels method*), 73
[search_protein_class\(\)](#) (*ChEMBL method*), 88
[search_query\(\)](#) (*Reactome method*), 171
[search_spellcheck\(\)](#) (*Reactome method*), 171
[search_suggest\(\)](#) (*Reactome method*), 171
[search_target\(\)](#) (*ChEMBL method*), 88
[Seqret](#) (*class in bioservices.seqret*), 177
[sequence](#) (*FASTA property*), 201
[Service](#) (*class in bioservices.services*), 56
[session](#) (*REST property*), 56
[show_entry\(\)](#) (*KEGG method*), 111
[show_module\(\)](#) (*KEGG method*), 112
[show_pathway\(\)](#) (*KEGG method*), 112
[showPathwayInBrowser\(\)](#) (*WikiPathways method*), 197
[snp_summary\(\)](#) (*EUtills method*), 97
[soup](#) (*easyXML property*), 59
[species_list\(\)](#) (*ReactomeOld method*), 176
[supported_databases](#) (*DBFetch property*), 190
[SV](#) (*FASTA property*), 200

T

[taxonomy_summary\(\)](#) (*EUtills method*), 97
[TIMEOUT](#) (*REST property*), 55
[TIMEOUT](#) (*WSDLService property*), 58
[Tnumber2code\(\)](#) (*KEGG method*), 104
[top_pathways\(\)](#) (*PathwayCommons method*), 141
[traverse\(\)](#) (*PathwayCommons method*), 141

U

[UniChem](#) (*class in bioservices.unichem*), 179
[UniProt](#) (*class in bioservices.uniprot*), 184
[uniref\(\)](#) (*UniProt method*), 187
[updatePathway\(\)](#) (*WikiPathways method*), 197
[url](#) (*ENA attribute*), 91
[url](#) (*Service property*), 57

V

[valid_attributes](#) (*BioMart property*), 69
[valid_mapping](#) (*UniProt property*), 188

`version` (*BiGG property*), [61](#)
`version` (*Reactome property*), [172](#)
`version()` (*BioMart method*), [69](#)

W

`wait()` (*MUSCLE method*), [123](#)
`wait()` (*NCBIblast method*), [131](#)
`WikiPathways` (*class in bioservices.wikipathway*), [190](#)
`wsdl_create_factory()` (*WSDLService method*), [58](#)
`wsdl_methods` (*WSDLService property*), [58](#)
`wsdl_methods_info()` (*WSDLService method*), [58](#)
`WSDLService` (*class in bioservices.services*), [57](#)