BioServices

Release 1.4.15

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

February 08, 2017

Contents

BioServices

Citations If you use BioServices, please cite *Cokelaer et al, Bioinformatics (2013)*.
See bioinformatics link for details.

Note: Contributions to implement new wrappers are more than welcome. See BioServices wiki to join the development, and the *Developer Guide* on how to implement new wrappers. Although contributors have their names in the header files, you can also look at the *Contributors* directly.

CHAPTER 1

Overview and Installation

1.1 Overview

BioServices is a Python package that provides access to many Bioinformatics Web Services (e.g., UniProt) and a framework to easily implement Web Service 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 Biological Web Services. By doing so, elaboration of new applications that combine several Web Services should be fostered.

One of the main philosophy of **BioServices** is to make use of the existing REST or SOAP/WSDL Web Services and therefore existing databases, not to re-invent new databases.

The first release of **BioServices** provides a wrapping to more than 18 Web Services (more if we consider **BioMart** and **PSICQUIC** portals that link to many other Web Services).

Here are only some of Web Services that are available in **BioServices**:

`bioservices.arrayexpress.ArrayExpress`	Interface to the ArrayExpress service
`bioservices.biomodels.BioModels`	Interface to the BioModels service
`bioservices.chembl.ChEMBL`	Interface to ChEMBL
`bioservices.kegg.KEGG`	Interface to the KEGG service
`bioservices.muscle.MUSCLE`	Interface to the MUSCLE service.
`bioservices.pdb.PDB`	Interface to part of the PDB service
`bioservices.uniprot.UniProt`	Interface to the UniProt service
`bioservices.ncbiblast.NCBIblast`	Interface to the NCBIblast service.
`bioservices.wikipathway.Wikipathway`	

Full list is available in the User Guide and Reference here below.

1.2 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.

CHAPTER 2

User guide

2.1 Quick Start

2.1.1 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 heteregeneous. However, output are mostly XML formatted or in tabulated separated column format (TSV). When XML is returned, it is usually parsed via the BeautilSoup 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**.

2.1.2 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
MELRPWLLWVVAATGTLVLLAADAQGQKVFTNTWAVRIPGGPAVANSVARKHGFLNLGQI
FGDYYHFWHRGVTKRSLSPHRPRHSRLQREPQVQWLEQQVAKRRTKRDVYQEPTDPKFPQ
QWYLSGVTQRDLNVKAAWAQGYTGHGIVVSILDDGIEKNHPDLAGNYDPGASFDVNDQDP
DPQPRYTQMNDNRHGTRCAGEVAAVANNGVCGVGVAYNARIGGVRMLDGEVTDAVEARSL
GLNPNHIHIYSASWGPEDDGKTVDGPARLAEEAFFRGVSQGRGGLGSIFVWASGNGGREH
DSCNCDGYTNSIYTLSISSATQFGNVPWYSEACSSTLATTYSSGNQNEKQIVTTDLRQKC
TESHTGTSASAPLAAGIIALTLEANKNLTWRDMQHLVVQTSKPAHLNANDWATNGVGRKV
SHSYGYGLLDAGAMVALAQNWTTVAPQRKCIIDILTEPKDIGKRLEVRKTVTACLGEPNH
ITRLEHAQARLTLSYNRRGDLAIHLVSPMGTRSTLLAARPHDYSADGFNDWAFMTTHSWD
EDPSGEWVLEIENTSEANNYGTLTKFTLVLYGTAPEGLPVPPESSGCKTLTSSQACVVCE
EGFSLHQKSCVQHCPPGFAPQVLDTHYSTENDVETIRASVCAPCHASCATCQGPALTDCL
SCPSHASLDPVEQTCSRQSQSSRESPPQQQPPRLPPEVEAGQRLRAGLLPSHLPEVVAGL
SCAFIVLVFVTVFLVLQLRSGFSFRGVKVYTMDRGLISYKGLPPEAWQEECPSDSEEDEG
RGERTAFIKDQSAL
```

See also:

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

2.1.3 KEGG service

The KEGG interface is similar but contains more methods. The tutorial presents the KEGG itnerface 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 elements 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 iIs 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

2.1.4 QuickGO

To acces to the GO interface, simply create an instance and look for a 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

2.1.5 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

2.1.6 Biomodels service

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

```
>>> from bioservices import biomodels
>>> b = biomodels.BioModels()
>>> model = b.getModelSBMLById('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.modelsId
```

Of course it does not tell you anything about a model; there are more useful functions such as *getModelsIdByUniprotId()* and others from the getModelsIdBy family.

See also:

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

See also:

Biomodels tutorial for more details

2.1.7 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 *entry()* methods. Let us first find information about the chemical product **caffein** using the `search()` method:

```
xml_response = r.search("caffein*")
```

The output is in XML format. Python provides lots of tools to deal with xml so you can surely find good tools.

Within bioservices, we wrap all returned XML documents into a BeautifulSoup object that ease the manipulation of XML documents.

As an example, we can extract all fields "id" as follows:

```
>>> [x.getText() for x in xml_response.findAll("id")]
[u'27902', u'10280', u'20944', u'30447', u'30319', u'30315', u'30311', u'30307']
```

The second method provided is the `entry()` method. Given an Id, you can query the Rhea database using Id found earlier (e.g., 10280):

```
>>> xml_response = r.entry(10280, "biopax2")
```

Warning: the r.entry output is also in XML format but we do not provide a specific XML parser for it unlike for the "search" method.

output format can be found in

```
>>> r.format_entry
['cmlreact', 'biopax2', 'rxn']
```

See also:

Reference guide of *bioservices.rhea.Rhea* for more details

2.1.8 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 accesssed to quite easily as demonstrated in the *Developer Guide* section.

2.2 Tutorials

This section present 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 Contributino to this section are welcomed.

Contents

2.2.1 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 oter 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 intereted 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
```

It is quite powerful and more examples will be showm. 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',
'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 neccesseraly 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:K04
'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 metho where you can specify the output format. In particular you can erquest 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 extact 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.

2.2.2 Biomodels tutorial

Start a biomodels interface:

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

look at the list of models Id:

```
print(s.modelsId)
```

Get a specific model given its Id. Let us play with the first model:

```
>>> s.modelsId[0]
'BIOMD0000000299'
```

and look at some meta information:

```
>>> print(s.getSimpleModelsByIds(s.modelsId[0]))
<?xml version="1.0" encoding="UTF-8"?>
<simpleModels>
<simpleModel>
    <referenceId>10643740</referenceId>
    <modelId>BIOMD0000000299</modelId>
    <modelSubmissionId>MODEL1101140000</modelSubmissionId>
    <modelName>Leloup1999_CircadianRhythms_Neurospora</modelName>
    <publicationId>10643740</publicationId>
    <authors>
        <author>Leloup JC</author>
        <author>Gonze D</author>
        <author>Goldbeter A</author>
    </authors>
    <encoders>
        <encoder>Catherine Lloyd</encoder>
        <encoder>Vijayalakshmi Chelliah</encoder>
    </encoders>
    <lastModificationDate>2011-01-18T12:23:47+00:00</lastModificationDate>
</simpleModel>
</simpleModels>
```

Some of these information can be retrieved specifically:

```
>>> ID = s.modelsId[0]
>>> print(s.getPublicationByModelId(ID))
10643740
```

2.2.3 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
```

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
MPDPAAHLPFFYGSISRAEAEEHLKLAGMADGLFLLRQCLRSLGGYVLSLVHDVRFHHFP
IERQLNGTYAIAGGKAHCGPAELCEFYSRDPDGLPCNLRKPCNRPSGLEPQPGVFDCLRD
AMVRDYVRQTWKLEGEALEQAIISQAPQVEKLIATTAHERMPWYHSSLTREEAERKLYSG
AQTDGKFLLRPRKEQGTYALSLIYGKTVYHYLISQDKAGKYCIPEGTKFDTLWQLVEYLK
LKADGLIYCLKEACPNSSASNASGAAAPTLPAHPSTLTHPQRRIDTLNSDGYTPEPARIT
SPDKPRPMPMDTSVYESPYSDPEELKDKKLFLKRDNLLIADIELGCGNFGSVRQGVYRMR
KKQIDVAIKVLKQGTEKADTEEMMREAQIMHQLDNPYIVRLIGVCQAEALMLVMEMAGGG
PLHKFLVGKREEIPVSNVAELLHQVSMGMKYLEEKNFVHRDLAARNVLLVNRHYAKISDF
GLSKALGADDSYYTARSAGKWPLKWYAPECINFRKFSSRSDVWSYGVTMWEALSYGQKPY
KKMKGPEVMAFIEQGKRMECPPECPPELYALMSDCWIYKWEDRPDFLTVEQRMRACYYSL
ASKVEGPPGSTQKAEAACA
```

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 = NCBIblast(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

Searching..................................................done



                                                                 Score    E
Sequences producing significant alignments:                      (bits) 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 finised before printing the results, which may be quite long. We could look at the beginnin 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 (
```

There are other ways to perform this conversion using the *bioservices.uniprot.UniProt.mapping()* or `bioeservices.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 dabase 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 intercations 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)
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")
```

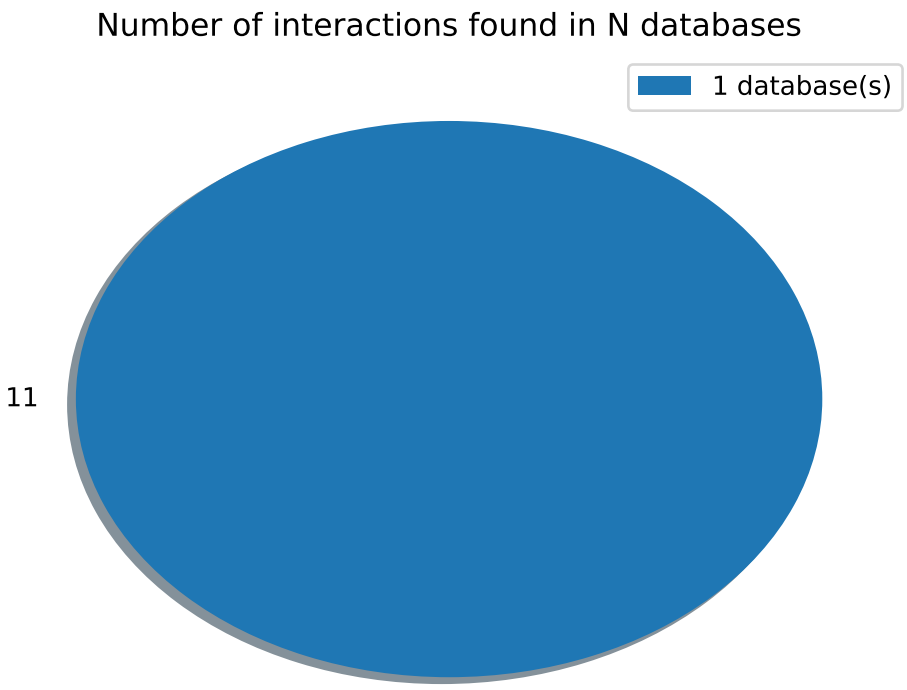
Warning: some databases may be offline. If so, try we another database. Type "s.activeDBs".

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 usefule. 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.

2.2.4 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 identifer 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"))
```

Form 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'}
```

2.2.5 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 `bioesrvices.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")
```

2.2.6 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) but let us use BioServices to access to the Cosmic mart programmatically.

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']
```

The 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 exercice 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")
>>> 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)
```

2.2.7 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/

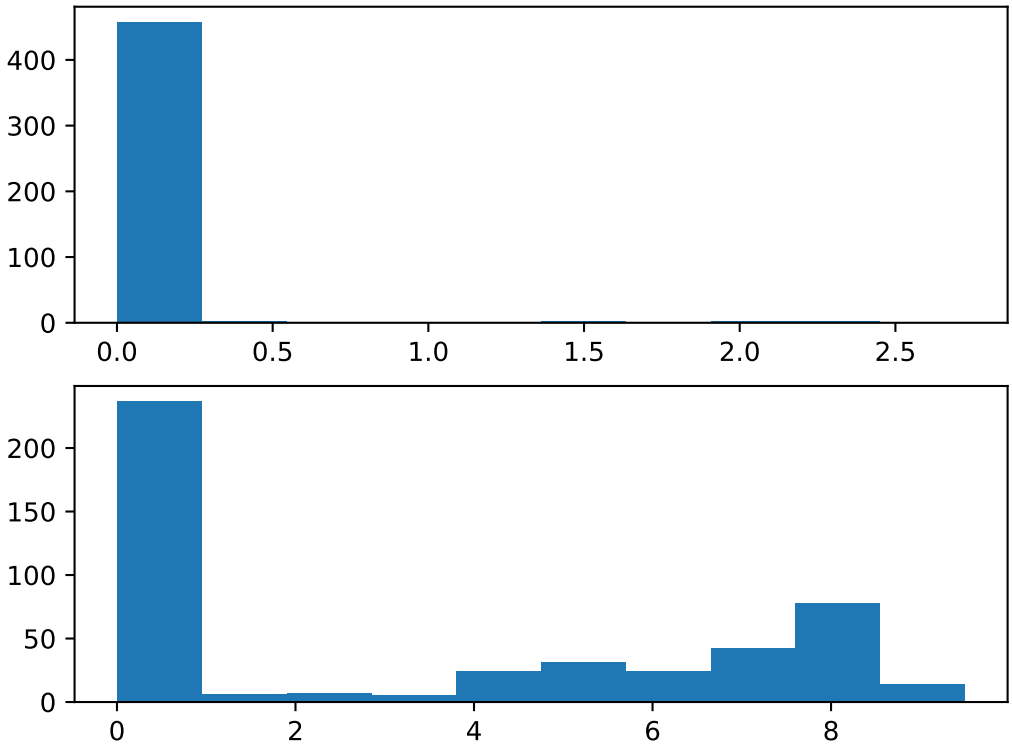
In the example below, we use geneprof to

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.

```
>>> 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"]=="embry
...
...         # aliases
...         sampleId = entry['sample_id']
...         gene = entry["Gene"]

...         # 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 simplied 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 t
>>> # 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.centrality_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")
```

Integrating expression data in pathways

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

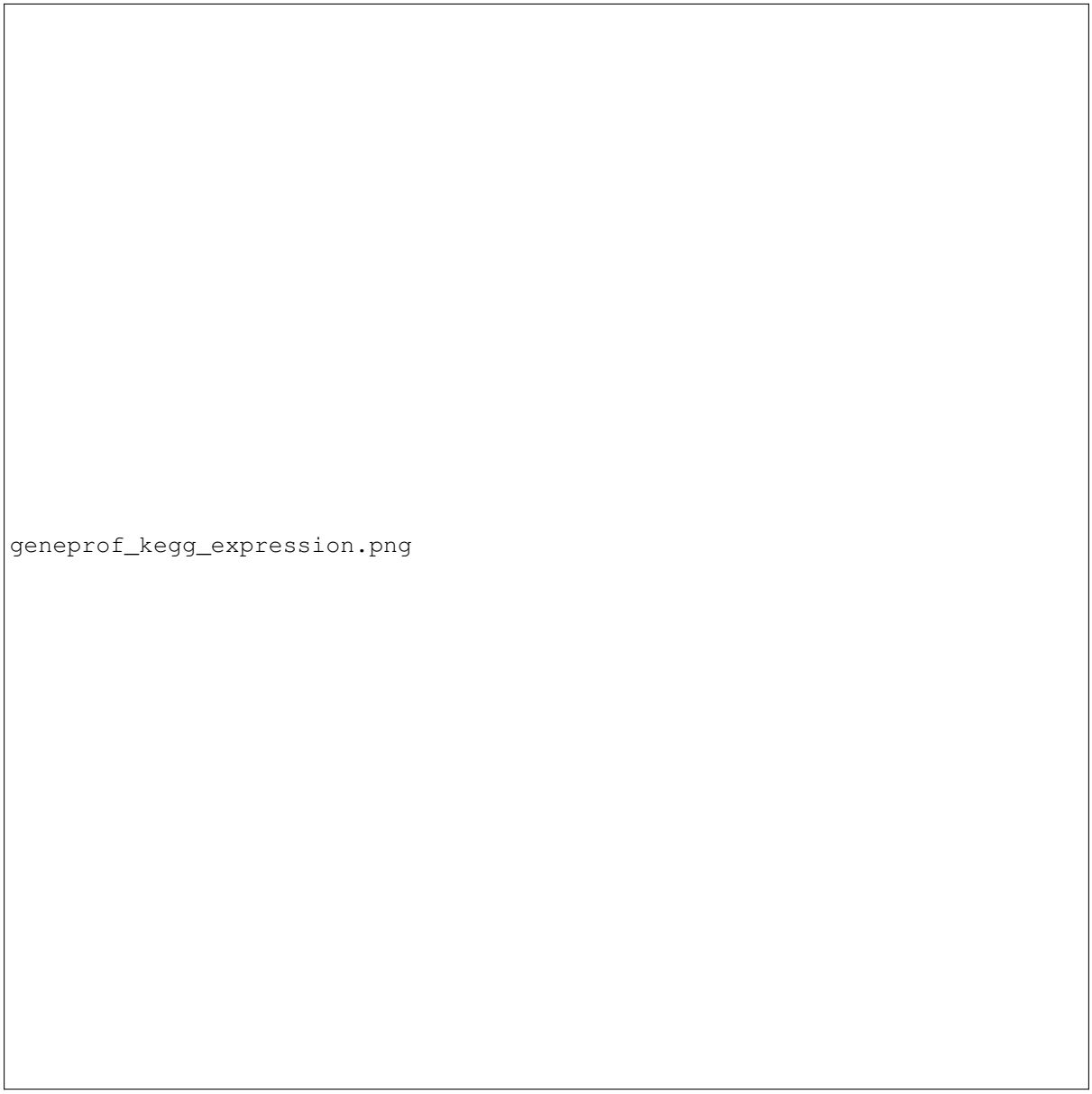
This is another example from the reference above but based on tools available in bioservices so as to overlaid highthroughput gene expression onto pathways and models from KEGG database.

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'])
>>> 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:



2.3 Combining BioServices with external tools

Contents

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

2.3.1 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 identifer. 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

2.3.2 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

>>> 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.

2.3.3 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 worflows 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 UniP
  </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., P4
  </help>
</tool>
```

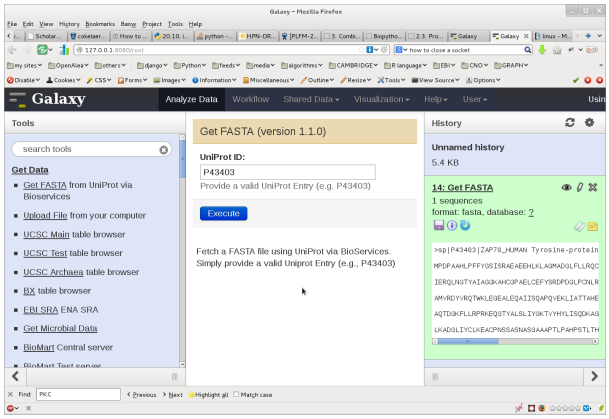
The python code will take as an input the UniProt ID and create a file that contains the FASTA data:

```
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 bioservoces ? Check that it is installed. Try 'pip install biose

    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="getext">
     <tool file="bioservices/uniprot.xml"/>
     <tool file="data_source/upload.xml"/>
...
```

Once done. start you galaxy server. The following image show 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.

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 identifer. 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

>>> 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 worflows 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 UniP
  </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., P4
  </help>
</tool>
```

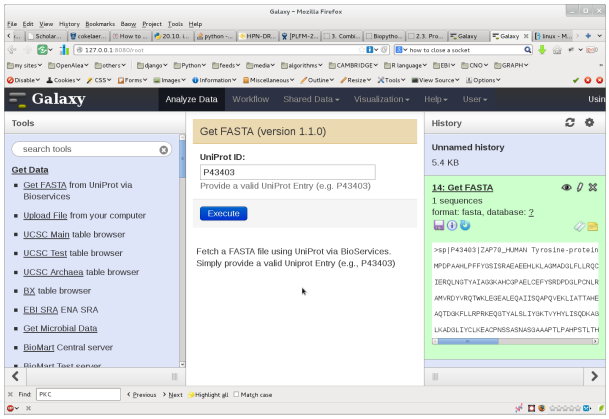
The python code will take as an input the UniProt ID and create a file that contains the FASTA data:

```
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 bioservoces ? Check that it is installed. Try 'pip install biose

    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="getext">
     <tool file="bioservices/uniprot.xml"/>
     <tool file="data_source/upload.xml"/>
...
```

Once done. start you galaxy server. The following image show 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.

2.4 Developer Guide

2.4.1 Creating a service class (REST case)

> **Warning:** since version 1.3.0, RESTService is deprecated and REST should be used. The changes should be easy: get method is now called http_get and requestPost is now http_post.

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 compulsary 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>
```

2.4.2 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.

2.5 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
```

2.6 suds and client auth

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

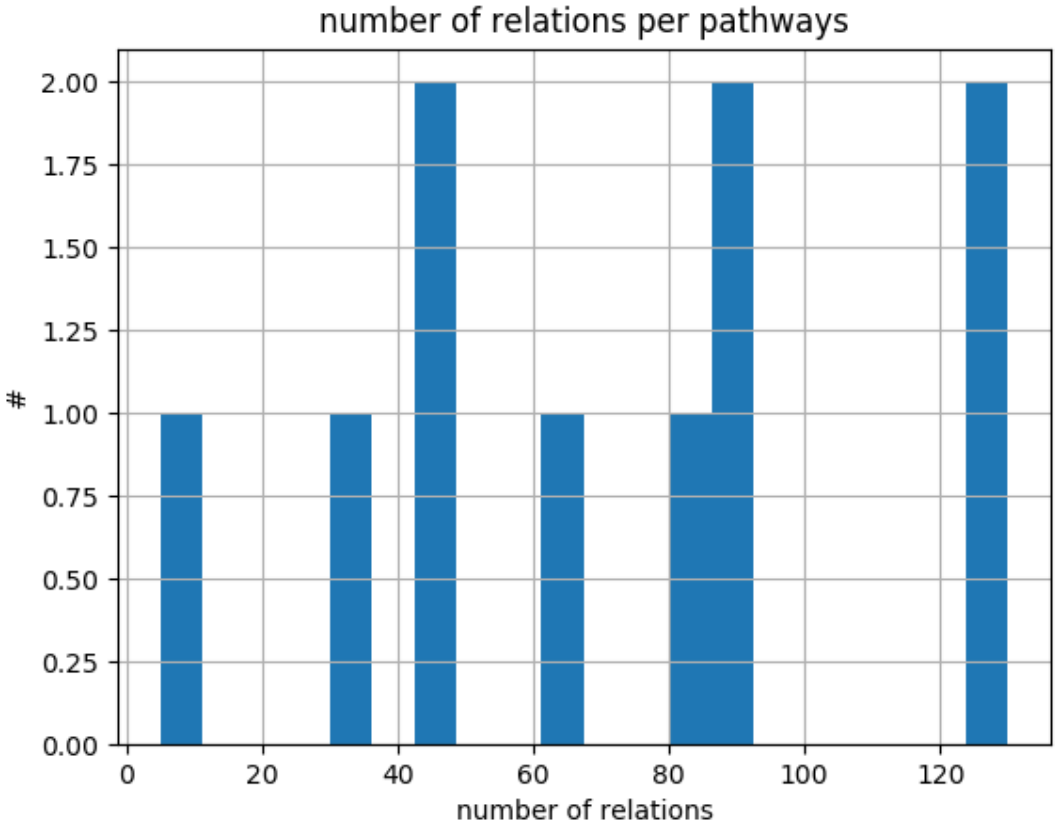
CHAPTER 3

Examples

3.1 Gallery

3.1.1 KEGG module example

Histogram of KEGG pathways relations



Out:

```
Creating directory /home/docs/.config/bioservices
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
```

```
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
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 9.623 seconds)

Download Python source code: plot_kegg_relations.py

Download Jupyter notebook: plot_kegg_relations.ipynb

Generated by Sphinx-Gallery

Download all examples in Python source code: auto_examples_python.zip

Download all examples in Jupyter notebooks: auto_examples_jupyter.zip

Generated by Sphinx-Gallery

3.2 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

3.2.1 UniProt

Here is a ipython notebook dedicated to UniProt, which can be downloaded `notebook/UniProt.ipynb` or view its results on uniprot nbviewer

3.2.2 BioModels

Here is a ipython notebook dedicated to BioModels, which can be downloaded `notebook/BioModels.ipynb` or view its results on biomodels nbviewer

3.2.3 ChEMBL

Here is a ipython notebook dedicated to ChEMBL, which can be downloaded `notebook/ChEMBL.ipynb` or view its results on chembl nbviewer

3.2.4 Entrez/Eutils

Here is a ipython notebook dedicated to EUtils, which can be downloaded `notebook/Entrez_EUtils.ipynb` or view its results on eutils nbviewer

3.2.5 KEGG

Here is a ipython notebook dedicated to KEGG, which can be downloaded `notebook/KEGG.ipynb` or view its results on KEGG nbviewer

3.2.6 MUSCLE

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

3.2.7 NCBIBlast

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

3.2.8 WikiPathway

Here is a ipython notebook dedicated to WikiPathway, which can be downloaded `notebook/WikiPathway.ipynb` or view its results on wikipathway nbviewer

Gene Mapping

This is a more general notebook related to mapping of identifiers, which combines several services.

Please download `notebook/Gene_Mapping.ipynb`.

CHAPTER 4

References

Contents

4.1 Utilities

4.1.1 Service module (REST or WSDL)

Modules with common tools to access web resources

class `Service`(*name*, *url=None*, *verbose=True*, *requests_per_sec=3*)

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 retricted. Currently implemented for REST only

All instances have an attribute called `logging` that is an instanceof 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 debugLebel is set to INFO. If verbose if 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.

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>]
```

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 = {400: 'Bad Request. There is a problem with your input', 404: 'Not found. The resource you re
some useful response codes

save_str_to_image(*data*, *filename*)
Save string object into a file converting into binary

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.

TIMEOUT

wsdl_create_factory(*name*, ***kargs*)

wsdl_methods
returns methods available in the WSDL service

wsdl_methods_info()

class RESTService(*name*, *url=None*, *verbose=True*)
Class to manipulate REST service

You can request an URL with this class that also inherits from *Service*.

For debugging:

•last_response contains

Constructor

Parameters

- **name** (*str*) – a name e.g. Kegg, Reactome, ...
- **url** (*str*) – the URL of the REST service
- **debugLevel** (*str*) – logging level. See *Service*.

getUserAgent()

http_get(*path*, *format='xml'*, *baseUrl=True*)

request(*path*, *format='xml'*, *baseUrl=True*)
Send a request via an URL to the web service.

Parameters

- **path** (*str*) – the request will be formed as self.url+/+path
- **format** (*str*) – If the expected output is in XML format then it will be converted with `easyXML()`. If the returned document is not in XML, format should be set to any other value.
- **baseUrl** (*str*) – By default, the path argument is appended to the `url` attribute (the main REST URL). However, sometimes, you would prefer to provide the entire URL yourself (e.g. in psicquic service) If so, set this baseUrl argument to False.

Note: this is a HTTP GET request

See also:

for developers see also the `_request_timeout()` if the site is down or busy.

Note: you can set the timeout of the connection, which is 1000 seconds by default by changing the `timeout`.

---

requestPost(*requestUrl*, *params*, *extra=None*)

request with a POST method.

Parameters

- **requestUrl** (*str*) – the entire URL to request
- **params** (*dict*) – the dictionary of parameter/value pairs
- **extra** (*str*) – an additional string to add after the params if needed. Could be usefule if a parameter/value can not be added to the dictionary. For instance is a parameter has several values Solved in requests module.

Todo

parameter paranName with a list of values [v1,v2] can be interpreted as paramName=v1&paramName=v2

Note: this is a HTTP POST request

Note: use only by ::*ncbiblast* service so far.

---

urlencode(*params*)

Returns a string compatible with a URL request.

Parameters **params** (*dict*) – a dictionary. Keys are parameters.

The pair of key/value are converted into a single string by concatenated the "&key=value" string for each key/value in the dictionary.

```
>>> params = {'a':1, 'b':2}
>>> urlencode(params)
"a=1&b=2"
```

Note: returns "a=1&b=2" or "b=2&a=1" since dictionary are not ordered. Note that the first parameter is not preceded by a & sign that you will need to add.

exception BioServicesError(*value*)

class REST(*name*, *url=None*, *verbose=True*, *cache=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')
```

TIMEOUT

clear_cache()

content_types = {'xml': 'application/xml', 'text': 'text/plain', 'phyloxml': 'text/x-phyloxml+xml', 'jsonp': 'text/ja

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*) – ste 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*, ***kargs*)

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

session

4.1.2 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 retreive 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()

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

4.2 Services

4.2.1 ArrayExpress

4.2.2 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 convertion from one database to all databases.

Constructor

Parameters **verbose** (*bool*) –

db2db(*input_db*, *output_db*, *inputValues*, *taxon=9606*)
Retrieves models associated to the provided Taxonomy text.

Parameters **text** (*str*) – free (Taxonomy based) text

Returns list of model identifiers

```
>>> from bioservices import BioDBNet
>>> input_db = 'Ensembl Gene ID'
>>> output_db = ['Gene Symbol']
>>> input_values = ['ENSG00000121410, ENSG00000171428']
>>> print(s.db2db(input_db, output_db, input_values, 9606)
Ensembl Gene ID Gene Symbol
ENSG00000121410 A1BG
ENSG00000171428 NAT1
```

dbFind(*input_db*, *inputValues*, *taxon='9606'*)

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

dbReport(*input_db*, *inputValues*, *taxon=9606*, *output='raw'*)
Returns report

Parameters **output** – returns dataframe if set to dataframe

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

dbWalk(*dbPath*, *inputValues*, *taxon=9606*)

getDirectOutputsForInput(*input_db*)

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")
```

4.2.3 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")
```

4.2.4 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*)
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 you 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 sevice 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(**args*, ***kargs*)

add_dataset_to_xml(**args*, ***kargs*)

add_filter_to_xml(**args*, ***kargs*)

attributes(**args*, ***kargs*)
to retrieve attributes available for a dataset:

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

configuration(**args*, ***kargs*)
to retrieve configuration available for a dataset:

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

create_attribute(**args*, ***kargs*)

create_filter(**args*, ***kargs*)

databases
list of valid datasets

datasets(**args*, ***kargs*)
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']
```

displayNames
list of valid datasets

filters(**args*, ***kargs*)
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_xml(**args*, ***kargs*)

host

hosts
list of valid hosts

lookfor(**args*, ***kargs*)

names
list of valid datasets

new_query(**args*, ***kargs*)

query(**args*, ***kargs*)
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" co
        <Dataset name="mmusculus_gene_ensembl" interface="default">
        <Filter name="ensembl_gene_id" value="ENSMUSG00000086981"/>
        <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 thiss method.

registry (**args*, ***kargs*)
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, displayNames

valid_attributes
list of valid datasets

version (**args*, ***kargs*)
Returns version of a **mart**

Parameters **mart** (*str*) – e.g. ensembl

4.2.5 BioModels

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

What is BioModels ?

URL http://www.ebi.ac.uk/biomodels-main/
Service http://www.ebi.ac.uk/biomodels-main/services/BioModelsWebServices?wsdl
Citations http://www.ncbi.nlm.nih.gov/pubmed/20587024
"BioModels 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 BioModels website, Dec. 2012

Some keywords used in this module:

identifier	Description/example
ChEBIIds	identifiers of a ChEBI terms (e.g. CHEBI:4991)
id	model identifier (e.g. BIOMD0000000408 or MODEL1201250000)
modelId	model identifier (e.g. BIOMD0000000408 or MODEL1201250000)
uniprotIds	Uniprot identifier (e.g., P41000)
taxonomyId	Taxonomy identifier (e.g. 9606)
GOId	Gene Ontology identifier (e.g. GO:0006915)
publicationIdOr-Text	publication identifier (PMID or DOI) or text which occurs in the publication's title or abstract

class BioModels (*verbose=True*)
Interface to the BioModels service

```
>>> from bioservices import *
>>> s = BioModels()
>>> model = s.getModelSBMLById('BIOMD0000000299')
```

The number of models available can be retrieved easily as well as the model IDs:

```
>>> len(s)
>>> s.modelsId
```

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

Constructor

Parameters **verbose** (*bool*) –

extra_getChEBIIds(*start=0*, *end=100*, *verbose=False*)
Retrieve existing chEBI Ids by scanning the models

Parameters

- **start** (*int*) – starting Id
- **end** (*int*) – end Id
- **verbose** (*bool*) – show the status of the analysis

Returns list of chEBI Ids that have been found in the DB.

This method may be useful to know the ChEBI Ids used in models

For instance, scanning the database for start=0 and end=200, a list of 191 chEBI Ids are returned and its takes a minute.

```
>>> s.extra_getChEBIIds(80, 84)
['CHEBI:81', 'CHEBI:83', 'CHEBI:84']
```

extra_getReactomeIds(*start=0*, *end=1000*, *verbose=False*)
Retrieve REACTOME Ids by scanning the models

Parameters

- **start** (*int*) – starting Id
- **end** (*int*) – end Id
- **verbose** (*bool*) – show the status of the analysis

Returns list of reactome IDs that have been found in the DB.

Search all models for reactome Ids in range 'REACT_start' to 'REACT_end'. Can take a while so do not be greedy and select a short range.

For instance, scanning the database for start=0 and end=3000, a list of 106 reactome Id are returned and its takes a minute or two.

```
>>> s.extra_getReactomeIds(0, 100)
['REACT_33', 'REACT_42', 'REACT_85', 'REACT_89']
```

extra_getUniprotIds(*start=10000*, *end=11000*)
Retrieve the Uniprot IDs

Parameters

- **start** (*int*) – starting ID value used to scan the database
- **end** (*int*) – ending ID value used to to scan the database

Returns list of uniprot IDs that have been found in the DB.

It may be useful to know the uniprot IDs that are available in all models. Not all of them are indeed available. This function is slow and you should use it with parcimony. This is why we set start and end IDs.

```
>>> s.extra_getUniprotIds(10000,11200)
['P10113',
 'P10415',
 'P10523',
 'P10600',
 'P10646',
 'P10686',
 'P10687',
 'P10815',
 'P11071']
```

getAllCuratedModelsId()

Retrieves the identifiers of all published **curated** models

Returns list of model identifiers

```
>>> s.getAllCuratedModelsId()
```

getAllModelsId()

Retrieves the identifiers of all published models

Returns list of model identifiers

```
>>> s.getAllModelsId()
```

Note: you cal also use the *modelsId*

getAllNonCuratedModelsId()

Retrieves the identifiers of all published **non curated** models

Returns list of model identifiers

```
>>> s.getAllNonCuratedModelsId()
```

getAuthorsByModelId(*Id*, ***kwargs*)

Retrieves the name of the authors of the publication associated with a given model.

param str Id a valid model Id. See *modelsId* attribute.

return list of names of the publication's authors

```
>>> s.getAuthorsByModelId("BIOMD0000000299")
['Leloup JC', 'Gonze D', 'Goldbeter A']
```

getDateLastModifByModelId(*Id*, ***kwargs*)

Retrieves the date of last modification of a given model.

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns date of last modification (expressed according to ISO 8601)

```
>>> s.getLastModifiedDateByModelId("BIOMD0000000256")
'2012-05-16T14:44:17+00:00'
```

Note: same as *getLastModifiedDateByModelId()*.

getEncodersByModelId(*Id*, ***kwargs*)
Retrieves the name of the encoders of a given model.

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns

list of names of the model's encoders

```
>>> s.getEncodersByModelId("BIOMD0000000256")
['Lukas Endler']
```

getLastModifiedDateByModelId(*Id*, ***kwargs*)
Retrieves the date of last modification of a given model.

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns date of last modification (expressed according to ISO 8601)

getModelById(**args*, ***kwargs*)
Retrieves the SBML form of a model (in a string) given its identifier

param str Id a valid ,odel Id

Warning: this method is now deprecated!

```
model = s.getModelById("BIOMD0000000256")
```

Instead, please use: *getModelSBMLById()*

getModelNameById(*Id*, ***kwargs*)
Retrieves the name of a model name given its identifier.

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns model name

```
>>> print(s.getModelNameById("BIOMD0000000256"))
'Rehm2006_Caspase'
```

getModelSBMLById(**args*, ***kwargs*)
Retrieves the SBML form of a model (in a string) given its identifier

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns str - SBML model in a string

```
>>> from bioservices import *
>>> s = BioModels()
>>> s.getModelSBMLById('MODEL1006230101')
```

getModelsIdByChEBI(*Id*)

Retrieves the identifiers of all models which are **associated** to some ChEBI terms. This relies on the method 'getLiteEntity' of the ChEBI Web Services (cf. http://www.ebi.ac.uk/chebi/webServices.do).

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns list of model identifiers

```
>>> s.getModelsIdByChEBI("CHEBI:4978")
['BIOMD0000000217', 'BIOMD0000000404']
```

See also:

getSimpleModelsByChEBIIds(), *getSimpleModelsRelatedWithChEBI()*, *getModelsIdByChEBIId()*

getModelsIdByChEBIId(*Id*)

Retrieves the identifiers of all the models which are **annotated** with a given ChEBI term.

Parameters **Id** (*str*) – identifier of a ChEBI term (e.g. CHEBI:4978)

Returns list of model identifiers

```
>>> from bioservices import *
>>> s = BioModels()
>>> s.getModelsIdByChEBIId('CHEBI:4978')
['BIOMD0000000404']
```

See also:

getSimpleModelsByChEBIIds(), *getSimpleModelsRelatedWithChEBI()*, *getModelsIdByChEBI()*

getModelsIdByGO(*goId*)

Retrieves the identifiers of all models related to a Gene Ontology Id.

Parameters **goId** (*str*) – a free GO text

Returns list of model identifiers

```
>>> s.getModelsIdByGO('0001756')
['BIOMD0000000201', 'BIOMD0000000275']
```

getModelsIdByGOId(*GOId*)

Retrieves the models which are annotated with the given Gene Ontology term.

Parameters **GOId** (*str*) – Gene Ontology identifier (e.g. GO:0006915)

```
>>> s.getModelsIdByGOId("GO:0006919")
['BIOMD0000000256', 'BIOMD0000000102', 'BIOMD0000000103']
```

getModelsIdByName(*name*)

Retrieves the models' identifiers which name includes the given keyword.

Parameters **name** (*str*) –

Returns array of strings - list of model identifiers

```
>>> res = s.getModelsIdByName("2009")
```

getModelsIdByPerson(*personName*)
Retrieves the identifiers of all models which have a given person as author or encoder.

Parameters **personName** (*str*) – author's or encoder's name

Returns list of model identifiers

```
>>> print(s.getModelsIdByPerson(u"Novère"))
```

Note: the use of the letter **u** in front of the string to encode special characters.

getModelsIdByPublication(*pubId*)
Retrieves the identifiers of all models related to one (or more) publication(s).

Parameters **pubId** (*str*) – publication identifier PMID or DOI or text which occurs in the publications.

```
s.getModelsIdByPublication('18308339')
['BIOMD0000000201']
```

getModelsIdByTaxonomy(*text*)
Retrieves the models which are associated to the provided Taxonomy text.

Parameters **text** (*str*) – free (Taxonomy based) text

Returns list of model identifiers

getModelsIdByTaxonomyId(*taxonomyId*)
Retrieves the models which are annotated with the given taxon.

Parameters **taxonomyId** (*str*) – Taxonomy identifier (e.g. 9606)

getModelsIdByUniprot(*text*)
Retrieves all the models which are associated to the provided UniProt text.

Parameters **text** (*str*) – a free UniProt based text

Returns list of model identifiers

```
>>> s.getModelsIdByUniprot("P10113")
['BIOMD0000000033']
```

getModelsIdByUniprotId(*Id*)
Retrieves all the models which are annotated with the given UniProt records.

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns list of model identifiers

```
>>> s.getModelsIdByUniprot("P10113")
['BIOMD0000000033']
```

getModelsIdByUniprotIds(*Ids_list*)
Retrieves all the models which are annotated with the given UniProt records.

Parameters **Ids_list** (*str*) – a valid model Id. See *modelsId* attribute.

Returns list of model identifiers

```
>>> s.getModelsIdByUniprotIds(["P10113", "P10415"])
['BIOMD0000000033', 'BIOMD0000000220']
```

getPublicationByModelId(*Id*, ***kwargs*)
Retrieves the publication's identifier of a given model.

Parameters **Id** (*str*) – a valid model Id. See *modelsId* attribute.

Returns publication identifier (can be a PMID, DOI or URL)

```
>>> s.getPublicationByModelId("BIOMD0000000256")
'16932741'
```

See also:

You can open the corresponding pubmed HTML page with `pubmed()`

getSimpleModelsByChEBIIds(*Id*)
Retrieves the models which are annotated with the given ChEBI terms.

Parameters **Id** (*str*) – identifier of a ChEBI term (e.g. CHEBI:4978)

Returns list with all models annotated with the provided ChEBI identifiers, as a TreeMap (which uses ChEBI identifiers as keys)

See also:

getSimpleModelsByChEBIIds(), *getModelsIdByChEBIId()*, *getModelsIdByChEBI()*

Warning: this method returns empty models even with example provided on BioModels website

getSimpleModelsByIds(*Id*, ***kwargs*)
Retrieves the main information about given models.

Parameters **Ids** (*str*) – a valid model Id. See *modelsId* attribute.

Returns a XML representaton (string) of the model meta information including identifier, name, publication identifier date of last modification...

```
>>> model = s.getSimpleModelsByIds("BIOMD0000000256")
```

Note: the output is a string. You can convert it to a class that ease its introspection using *easyXML()*.

getSimpleModelsByReactomeIds(*reacID*, *raw=False*)
Retrieves all the models which are annotated with the given Reactome records.

param list of reactome identifiers (e.g., REACT_1590)

param bool raw return raw data if True

return models annotated with the provided Reactome identifiers, as a TreeMap (which uses Reactome identifiers as keys)

See also:

How to retrieve REACTOME IDs in *extra_getReactomeIds()*

Note: the output is converted to ease introspection using *easyXML()*.

getSimpleModelsRelatedWithChEBI()
Retrieves all the models which are annotated with ChEBI terms.

The output of this function is a lengthy XML document containing utf-8 characters and the models (in simple models format). You can convert it and extract information such as the models ID by using the our xml parser and the following code:

```
res = s.getSimpleModelsRelatedWithChEBI()
res = self.easyXML(res.encode('utf-8'))
set([x.findall('modelId')[0].text for x in res.getchildren()])
```

Note: the output is a string. You can convert it to a class that ease its introspection using *easyXML()*.

See also:

getSimpleModelsByChEBIIds(), *getModelsIdByChEBIId()*, *getModelsIdByChEBI()*

getSubModelSBML(**args*, ***kwargs*)
Generates the minimal sub-model of a given model in the database including all selected components.

Parameters

- **modelId** – identifier of the model from which the sub-model will be extracted
- **elementsIDs** – identifiers of the selected elements. Currently only supports identifiers from compartments, species, and reactions. could be a string or list of strings

```
s.getSubModelSBML("BIOMD0000000242", "cyclinEdegradation_1")
```

modelsId
list of all model Ids

4.2.6 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 hybride" "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.

4.2.7 ChEMBL

This module provides a class *ChEMBL*

What is ChEMBL

URL https://www.ebi.ac.uk/chembldb/index.php/
REST https://www.ebi.ac.uk/chembldb/index.php/ws
FAQS https://www.ebi.ac.uk/chembldb/index.php/faq
"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*)
Interface to ChEMBL

Here is a quick example to retrieve a target given its ChEMBL Id

```
>>> from bioservices import ChEMBL
>>> s = ChEMBL(verbose=False)
>>> resjson = s.get_target_by_chemblId('CHEMBL240')
>>> resjson['proteinAccession']
'Q12809'
```

By default, most methods return dictionaries (converted from json objects returned by the ChEMBL API), however, you can also set the format to be XML.

get_all_targets(*frmt='json'*)
Get all targets in a dictionary

Parameters **frmt** (*str*) – json or xml (Default to json)

Returns Target Record in a dictionary (default is json)

The returned list contains a dictionary for each **target**. Here are some keys contained in the dictionaries:

•chemblId

•description

•geneNames

•organism

•preferredName

•proteinAccession

•synonyms

•targetType

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> res = s.get_all_targets()
```

get_alternative_compound_form(*query*, *frmt='json'*)
Get list of alternative compound forms (e.g. parent and salts) of compound

Parameters

- **query** – a valid compound ChEMBLId
- **frmt** – json or xml (Default to json)

Returns List of ChEMBLIDs which correspond to alternative forms of query compound

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> resjson = s.get_alternative_compound_form(s._alt_compound_form_example)
```

get_approved_drugs(*query*, *frmt='json'*)
Get list of approved drugs chembl compound details

Parameters

- **query** – a valid target ChEMBLId
- **frmt** – json or xml (Default to json)

Returns list of approved drugs ChEMBL compound details (dictionary)

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> resjson = s.get_approved_drugs(s._target_approved_drugs_example)
```

get_assays_bioactivities(*query*, *frmt='json'*)
Get individual assay bioactivities

Parameters

- **query** (*str*) – a valid assay ChEMBLId.
- **frmt** (*str*) – json or xml (Default to json)

Returns Bioactivity records for a given assay (dictionary)

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._assays_example)
>>> resjson = s.get_assays_bioactivities(s._assays_example)
```

get_assays_by_chemblId(*query*, *frmt='json'*)
Get assay by ChEMBLId

Parameters

- **query** (*str*) – a valid assay ChEMBLId.
- **frmt** (*str*) – json or xml (Default to json)

Returns Assay record as a dictionary (if json requested) (list of dictionaries if input is a list)

If json format is requested, a dictionary is returned. with some of the following keys:

•assayDescription

•assayOrganism

•assayStrain

•assayType

•chemblId

•journal

•numBioactivities

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._assays_example)
CHEMBL1217643
>>> resjson = s.get_assays_by_chemblId(s._assays_example)
```

get_compounds_activities(*query*, *frmt='json'*)
Get individual compound bioactivities

Parameters

- **query** (*str*) – valid chembl identifier (or list)
- **frmt** (*str*) – json or xml (Default to json)

Returns Bioactivity records in XML or dictionary (if json requested)

If json format is requested, a dictionary is returned. Here are some of the keys:

•activity_comment

•assay_description

•name_in_reference

•organism

•reference

•target_confidence

•target_name

•units

•value

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._bioactivities_example)
>>> resjson = s.get_compounds_activities(s._bioactivities_example)
```

get_compounds_by_SMILES(*query*, *frmt='json'*)
Get list of compounds by Canonical SMILES

Parameters

- **query** (*str*) – a valid compound ChEMBLId or a list of those ones.
- **frmt** (*str*) – json or xml (Default to json)

Returns Compound Record in XML or dictionary (if json requested). If query os a list/tuple, a tuple of compound records is returned.

If json format is requested, a dictionary is returned. The dictionary has a unique key 'compounds'. The value of that key is a list of compound records. For each compound record dictionary see *get_compounds_by_chemblId()*.

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._smiles_example)
>>> resjson = s.get_compounds_by_SMILES(s._smiles_example)
```

get_compounds_by_chemblId(*query*, *frmt='json'*)
Get compound by ChEMBLId

Parameters

- **query** – a compound ChEMBLId or a list of those ones. if the identifier is invalid, the number 404 is returned.
- **frmt** (*str*) – json or xml (Default to json)

Returns Compound Record (dictionary). If the query is a list of identifiers, a list of compound records is returned.

If json format is requested, a dictionary is returned. Here are some of the keys:

•acdLogd

•chemblId

•knownDrug

•molecularFormula

•molecularWeight

•passesRuleOfThree

•smiles

•stdInChiKey

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print (s._chemblId_example)
>>> resjson = s.get_compounds_by_chemblId(s._chemblId_example)
```

get_compounds_by_chemblId_drug_mechanism(*query*, *frmt='json'*)

Parameters

- **query** – valid chembl identifier (or list)
- **frmt** (*str*) – json or xml (Default to json)

See *get_compounds_by_chemblId()* for full doc.

```
>>> s.get_compounds_by_chemblId_drug_mechanism("CHEMBL3")
[{u'chemblId': u'CHEMBL1347191', u'parent': False},
{u'chemblId': u'CHEMBL1558', u'parent': False},
{u'chemblId': u'CHEMBL2', u'parent': True}]
```

get_compounds_by_chemblId_form(*query*, *frmt='json'*)

Parameters

- **query** – valid chembl identifier (or list)
- **frmt** (*str*) – json or xml (Default to json)

See *get_compounds_by_chemblId()* for full doc.

```
>>> s.get_compounds_by_chemblId_form("CHEMBL2")
[{u'chemblId': u'CHEMBL1347191', u'parent': False},
 {u'chemlId': u'CHEMBL1558', u'parent': False},
 {u'chemblId': u'CHEMBL2', u'parent': True}]
```

get_compounds_containing_SMILES(*query*, *frmt='json'*)

Deprecated. Use get_compounds_substructure

get_compounds_similar_to_SMILES(*query*, *similarity=90*, *frmt='json'*)

Get list of compounds similar to the one represented by the given Canonical SMILES.

The similarity is at a cutoff percentage score (minimum value=70%, maximum value=100%).

Parameters

- **query** (*str*) – a valid SMILES string or a list of those ones.
- **frmt** (*str*) – json or xml (Default to json)

Returns Compound records. See *get_compounds_by_chemblId()*

Each dictionary has a set of keys amongst which:

•acdAcidicPka

•acdBasicPka

•similarity

•species

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._smiles_similar_example)
>>> resjson = s.get_compounds_similar_to_SMILES(s._smiles_example, "100")
>>> resjson = s.get_compounds_similar_to_SMILES(s._smiles_similar_example)
```

get_compounds_substructure(*query*, *frmt='json'*)

Get list of compounds containing the substructure represented by the given Canonical SMILES

Parameters

- **query** (*str*) – a valid SMILES string or a list of those ones.
- **frmt** (*str*) – json or xml (Default to json)

Returns see *get_compounds_by_SMILES()*

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._smiles_example)
>>> res = s.get_compounds_substructure(s._smiles_struct_example)
```

get_image_of_compounds_by_chemblId(*query*, *dimensions=500*, *save=True*, *view=True*, *engine='rdkit'*)

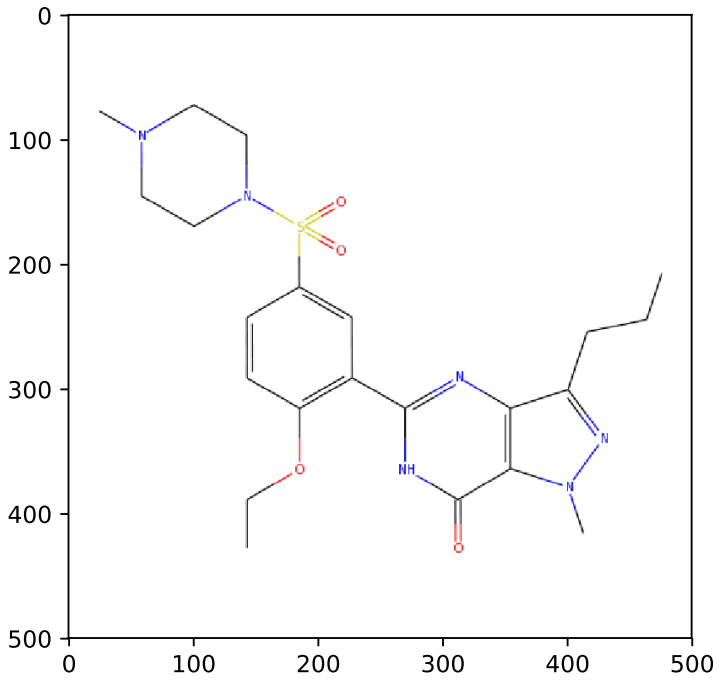
Get the image of a given compound in PNG png format.

Parameters

- **query** (*str*) – a valid compound ChEMBLId or a list/tuple of valid compound ChEMBLIds.
- **dimensions** (*int*) – optional argument. An integer z ($1 \leq z \leq 500$) giving the dimensions of the image.
- **save** –
- **view** (*bool*) –
- **engine** – Defaults to rdkit. Implemented for the future but only one value for now.
- **view** – show the image. If True the images are opened.

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_of_compounds_by_chemblId(s._image_chemblId_example)
>>> imshow(imread(res['filenames'][0]))
```



Todo

ignorecoords option

get_individual_compounds_by_inChiKey(*query*, *frmt='json'*)
Get individual compound by standard InChi Key

Parameters

- **query** (*str*) – a valid InChi key or a list.
- **frmt** (*str*) – json or xml (Default to json)

Returns Compound Record as a dictionary (or list of dictionaries)

In addition to the keys returned in *get_compounds_by_chemblId()*, the following keys are returned:

•acdAcidicPka

•acdBasicPka

•preferredCompoundName

•species

•synonyms

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._inChiKey_example)
QFFGVLORLPOAEC-SNVBAGLBSA-N
>>> resjson = s.get_individual_compounds_by_inChiKey(s._inChiKey_example)
```

get_target_bioactivities(*query*, *frmt='json'*)
Get individual target bioactivities

Parameters

- **query** (*str*) – a valid target ChEMBLId.
- **frmt** (*str*) – json or xml (Default to json)

Returns see *get_compounds_activities()*

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._target_bioactivities_example)
>>> resjson = s.get_target_bioactivities(s._target_bioactivities_example)
```

get_target_by_chemblId(*query*, *frmt='json'*)
Get target by ChEMBLId

Parameters

- **query** (*str*) – target ChEMBLId
- **frmt** (*str*) – json or xml (Default to json)

Returns Target Record in XML or dictionary (if json requested)

If json format is requested, a dictionary is returned. Here are some of dictionary's keys:

•target

•geneNames

•organism

•proteinAccession

•targetType

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._target_chemblId_example)
>>> resjson = s.get_target_by_chemblId(s._target_chemblId_example)
```

get_target_by_refseq(*query*, *frmt='json'*)
Get individual target by RefSeq Accession identifier

Parameters

- **query** (*str*) – a valid RefSeq accession Id
- **frmt** (*str*) – json or xml (Default to json)

Returns Target Record

Warning: this method works but I could not find any example to validate it. NP_001128722 provided on chembl blogspot does not work. NP_015325 from the ChEMBL API does not work either.

get_target_by_uniprotId(*query*, *frmt='json'*)
Get individual target by UniProt Accession Id

Parameters

- **query** (*str*) – a valid uniprot accession Id.
- **frmt** (*str*) – json or xml (Default to json)

Returns see *get_target_by_chemblId()*

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._target_chemblId_example)
>>> resjson = s.get_target_by_uniprotId(s._target_uniprotId_example)
```

inspect(*query*, *item_type*)
Open the URL of a query in a browser.

Parameters

- **query** (*str*) – a valid ChEMBLId of a compound, target or assay.
- **item_type** (*str*) – a valid type. Might be compound, target or assay

```
>>> from bioservices import *
>>> s = ChEMBL(verbose=False)
>>> print(s._assays_example)
>>> s.inspect(*s._inspect_example)
>>> print(s._assays_example)
>>> s.inspect(s._assays_example,'assay')
```

status()
Return the API status

Returns Response is the string 'UP' if the service is running

version()
Return version of the API on the server

4.2.8 ChemSpider

Interface to the ArrayExpress web Service.

What is ChemSpider ?

Status in progress
URL http://www.chemspider.com/
REST http://www.chemspider.com/AboutServices.aspx?
ChemSpider is a free chemical structure database providing fast access to over 28 million structures, properties and associated information. By integrating and linking compounds from more than 400 data sources, ChemSpider enables researchers to discover the most comprehensive view of freely available chemical data from a single online search. It is owned by the Royal Society of Chemistry.

—ChemSpider home page, March 2013

class ChemSpider(*verbose=False*, *token=None*, *cache=False*)
ChemSpider Web Service Interface

Status in progress you can already search for Id and compound or retrieve the chemical image of an Id

```
>>> from bioservices import *
>>> s = ChemSpider()
>>> s.find("Pyridine")
[1020]
>>> results = s.GetExtendedCompoundInfo(1020)
>>> results['averagemass']
79.0999
```

GetExtendedCompoundInfo(*Id*)

ImagesHandler(*Id*)

databases
Returns databases searched for in chemSpider

find(*query*)
return the first 100 compounds that match the query

image(*Id*)
Return string containing PNG binary image data of 2D structure image

```
>>> from bioservices import *
>>> s = ChemSpider()
>>> ret = s.image(1020)
>>> with open("test.png", "w") as f:
...     f.write(ret)
>>> s.on_web("test.png")
```

mol(*Id*)
Return record in MOL format

mol3d(*Id*)
Return record in MOL format with 3D coordinates calculated

token

4.2.9 Clinvitae

class Clinvitae

class for interfacing with the Clinvitae web service

Requests will return a list of json dicts. each dict has the following fields:

```
accessionId
gene
nucleotideChanges
description
classification
reportedClassification
url
region
proteinChange
lastUpdated
alias
source
acmgClassification
submitter
defaultNucleotideChange
_id
transcripts
lastEvaluated
```

example query on BRCA1:

```
>>> c = Clinvitae()
>>> res = c.query_gene('brca1')
>>> entry1 = res[0]
>>> print(entry1.keys())  # display fields for first entry
>>> print(entry1['accessionId']) # accession id for first entry
>>> print(entry1['lastEvaluated'])  # date first variant entry was last evaluated
>>> print(entry1['source'])  # source of first variant entry
```

all_variants(*gene*)

returns a list of unique hgvs tags reported in gene

```
>>> c = Clinvitae()
>>> res = c.all_variants('MUTYH')  # returns all reported variants in MUTYH gene
>>> print(res[0:5])
[u'NM_001048171.1:c.-2188C>T',
u'NM_001048171.1:c.462+35A>G',
u'NM_001048171.1:c.1099G>T',
u'NM_001048171.1:c.972G>C',
u'NM_001048171.1:c.1476+35C>T']
```

get_VUS(*gene*)

returns all fields for entries not classified as benign or pathogenic -> variant of unknown significance (VUS)

```
>>> c = Clinvitae()
>>> vus = c.get_VUS('brca1')
```

```
>>> len(vus)  # number of benign variants
2389
```

get_benign(*gene*)
returns all fields for entries reported as Benign by any source in Clinvitae

```
>>> c = Clinvitae()
>>> benign = c.get_benign('brca1')  # returns benign or likely benign
>>> len(benign)  # number of benign variants
187
```

get_pathogenic(*gene*)
returns all fields for entries reported as Pathogenic by any source in Clinvitae

```
>>> c = Clinvitae()
>>> pathogenic = c.get_pathogenic('brca1')  # returns pathogenic or likely pathogenic
>>> len(pathogenic)  # number of pathogenic variants
1100
```

query_gene(*gene*)
takes gene name and returns json of variants in gene (not case sensitive)

```
>>> c = Clinvitae()
>>> res = c.query_gene('brca1')
>>> entry1 = res[0]
>>> print(entry1['accessionId'])  # accession id for first entry
u'SCV000039520'
>>> print(entry1['lastEvaluated'])  # date first variant entry was last evaluated
u'2013-04-03'
>>> print(entry1['source']0  # source of first variant entry
u'ClinVar'
```

query_hgvs(*hgvs*)
Takes an hgvs (variant) tag and returns ALL reported variants in the gene in which hgvs is located

```
>>> c = Clinvitae()
>>> res = c.query_gene('NM_198578.3:c.1847A>G')  # returns all entries in LRRK2 gene
>>> entry1 = res[0]
>>> print(entry1['accessionId']) # accession id for first entry
u'SCV000056058'
>>> print(entry1['lastEvaluated']) # date first variant entry was last evaluated
u'2012-09-13'
>>> print(entry1['source']) # source of first variant entry
u'ClinVar'
```

4.2.10 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.

—From ENA web page Jan 2016

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 ChEMBL
>>> 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 i 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')
```

Todo

Taxon viewer, image retrieval.

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

Todo

download and save at the same time. Right now the fasta is retuned as a string and needs to be saved manually. It may also be an issue with very large fasta files.

get_taxon(*taxon*)

view_data(*identifier*, *fasta_range=None*)

4.2.11 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"))
>gi|34577063|ref|NP_001117.2| adenylosuccinate synthetase isozyme 2 [Homo sapiens]
MAFAETYPAASSLPNGDCGRPRARPGGNRVTVVLGAQWGDEGKGKVVDLLAQDADIVCRCQGGNNAGHTV
VVDSVEYDFHLLPSGIINPNVTAFIGNGVVIHLPGLFEEAEKNVQKGKGLEGWEKRLIISDRAHIVFDFH
QAADGIQEQQRQEQAGKNLGTTKKGIGPVYSSKAARSGLRMCDLVSDFDGFSERFKVLANQYKSIYPTLE
IDIEGELQKLKGYMEKIKPMVRDGVYFLYEALHGPPKKILVEGANAALLDIDFGTYPFVTSSNCTVGGVC
TGLGMPPQNVGEVYGVVKAYTTRVGIGAFPTEQDNEIGELLQTRGREFGVTTGRKRRCGWLDLVLLKYAH
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).

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.

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, ...

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]
MAFAETYPAASSLPNGDCGRPRARPGGNRVTVVLGAQWGDEGKGKVVDLLAQDADIVCRCQGGNNAGHTV
VVDSVEYDFHLLPSGIINPNVTAFIGNGVVIHLPGLFEEAEKNVQKGKGLEGWEKRLIISDRAHIVFDFH
QAADGIQEQQRQEQAGKNLGTTKKGIGPVYSSKAARSGLRMCDLVSDFDGFSERFKVLANQYKSIYPTLE
IDIEGELQKLKGYMEKIKPMVRDGVYFLYEALHGPPKKILVEGANAALLDIDFGTYPFVTSSNCTVGGVC
TGLGMPPQNVGEVYGVVKAYTTRVGIGAFPTEQDNEIGELLQTRGREFGVTTGRKRRCGWLDLVLLKYAH
MINGFTALALTKLDILDMFTEIKVGVAYKLDGEIIPHIPANQEVLNKVEVQYKTLPGWNTDISNARAFKE
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="nuccore",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 XML data structure parsed with *EUtilsParser*

```
>>> 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*, *retmode='json'*, ***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 an XML or json data structure that depends on the value of **:param:'databases'** (default to json)

```
>>> all_database_names = s.EInfo()
>>> # specific info about one database:
>>> ret = s.EInfo("taxonomy")
>>> ret['count']
u'1445358'
>>> ret = s.EInfo('pubmed')
>>> ret['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", retmode='xml')
>>> ret = s.parse_xml('objectify')
>>> ret.root.DbInfo.FieldList.getchildren()[2].FullName
'Filter'
```

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*, *retmode='json'*, ***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', retmode='json')
>>> # 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 possiblities (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="omim", term="aasthma+OR+alergy")
>>> ret.Query
'asthmaa OR alergies'
>>> ret.CorrectedQuery
'asthma or allergy'
>>> ret = e.ESpell(db="pubmed", term="biosservices")
>>> ret.CorrectedQuery
bioservices
```

Note: only WSDL protocol available

ESummary (*db*, *id=None*, *retmode='json'*, ***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 UIDs 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'
```

databases
Returns list of valid databases

email = None
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()*.

4.2.12 GeneProf

This module provides a class *GeneProf*

What is GeneProf ?

URL http://www.geneprof.org
REST http://ww.geneprof.org/GeneProf/api
Citations Halbritter F, Kousa AI, Tomlinson SR. GeneProf data: a resource of curated, integrated and reusable high-throughput genomics experiments. Nucleic Acids Research (2013). PubMed ID: 24174536, Full Article @ NAR.

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!

—GeneProf home page, Nov 2013

New in version 1.2.0.

Data is freely available, under the license terms of each contributing database.

class GeneProf (*verbose=True*)

Interface to the GeneProf service

Here are some GeneProf terminology used throughout the web service interface:

- Experiments are what GeneProf calls each individual data analysis project. An experiment typically consists of a set of input data (e.g. raw high-throughput sequencing reads), some experimental sample annotation, an analysis workflow and a selection of main outputs. More information here in the Experiments section
- Datasets, in GeneProf, are collections of data of the same type generated as the output of a component of an data analysis workflow. There are six generic types of datasets: FILE, SEQUENCES, GENOMIC_REGIONS, FEATURES, REFERENCE and SPECIAL. More information in Datasets section.

Warning: some of the GeneProf services requires registration. More about the API key registration on GeneProf API page

Warning: there may be a limitation on the number of request per day

New in version 1.2.0.

Constructor

Parameters **verbose** (*bool*) – prints informative messages

Most of the time, outputs are returned in one of the following format: JSON, XML, TXT or RData (R data set). The default within bioservices is set to JSON format. You can change the default behaviour by changing the attribute *default_extension*.

```
>>> from bioservices import GeneProf
>>> g = GeneProf(verbose=False)
>>> exp = g.get_list_experiments()
```

Some functionalities expect the ID of a GeneProf reference dataset as one input parameter (e.g., pub_hs_ens59_grch37) but aliases are available. The list of species alias is available in the *valid_species* attribute.

default_extension

set extension of the requests (default is json). Can be 'json' or 'xml', 'txt', 'rdata'.

get_bed_files (*Id*, *chromosome=None*, *key=None*, *filter_column=None*, *with_track_description=True*, *only_distinct=False*)

Retrieve Genomic Data as compressed BED Files (gzipped)

Retrieves genomic data as BED files in compressed gzipped format. It works only for datasets with type GENOMIC_REGIONS i.e. those containing genomic data! The dataset of interest is identified by its GeneProf accession ID (something of the form gpDS_XXX_XXX_XXX_X). You can get a list of datasets belonging to a certain experiment of interest using the metadata for an experiment service *get_metadata_experiment()*, or you can use the search datasets service to query datasets globally. The maximum size of datasets retrieved without an API key is restricted to 10,000,000 entries.

Note: chromosomes in the output BED can be dynamically renamed in order to make the names compatible with other applications (that's because, unfortunately, not all genome databases use the

same names, see also the *get_chromosome_names()* method).

Parameters

- **Id** (*str*) – The identifier of the dataset of interest. Either the entire accession ID (e.g. gpDS_11_385_44_1) or just the dataset-specific part (e.g. 11_385_44_1).
- **chromosome** (*str*) – An optional parameter that may be used to rename chromosomes in the output. The value should be comma-separated map from chromosome ID to its name in the output, where key and value are to be separated with a hyphen (-), e.g. 1-chr1,2-chr2,3-chr12. Any chromosome not mentioned in the map will not be exported, so you can use this as a filtering mechanism, too. Use the Get Chromosome Names service to get a list of all the available chromosome in a dataset with their default names.
- **filter_column** (*bool*) – The ID of a column / annotation type holding boolean flags. Only entries for which this boolean flag is true will be exported.
- **bool with-track-description** (*bool*) – Include a track description header.
- **only_distinct** (*bool*) – Export only one entry if there are multiple with the same coordinates.
- **key** (*str*) – An optional WebAPI key, required to access non-public data.

Retrieve ChIP-seq peaks for FoxA1 from dataset gpDS_11_3_7_2:

```
>>> g.get_bed_files("11_3_7_2")
```

Retrieve only the ChIP-seq peaks on chromosome 3 for FoxA1 from dataset gpDS_11_3_7_2:

```
>>> g.get_bed_files("11_3_7_2", chromosome="3-chr3")
```

Retrieve gene coordinates from the zebrafish reference dataset without a track header:

```
>>> g.get_bed_files("zebrafish", with_track_description=False)
```

Retrieve only the ChIP-seq peaks for Stat3 (identified by the column $C_11_12_125_2_14_TFBS) from a dataset containing peaks for many different factors (gpDS_11_12_125_2):

```
>>> g.get_bed_files("11_12_125_2",
        filter_column="C_11_12_125_2_14_TFBS")
```

get_chromosome_names(*Id*, *frmt='json'*, *key=None*)

Get Chromosome Names

Retrieve the IDs and names of all chromosomes in a genomic dataset. This service can only be used for genomic datasets, i.e. for datasets with type GENOMIC_REGIONS or REFERENCE.

The names in genome databases use to refer to chromosomes, even of well-known organisms, are not always the same. For example, the mitochondrial (pseudo-)chromosome is usally called 'chrMT' in Ensembl, but 'chrM' in the UCSC databases. The data as *get_bed_files()* or as WIG might therefore require you to rename the experiments in the output, before using them with other applications.

Parameters

- **Id** (*str*) – The identifier of the dataset of interest. Either the entire accession ID (e.g. gpDS_11_385_44_1) or just the dataset-specific part (e.g. 11_385_44_1).
- **key** (*str*) – optional WebAPI key, required to access non-public data.

Get all chromosomes for the mouse reference dataset in plain text format:

```
>>> g.get_chromosome_names("pub_mm_ens58_ncbim37", frmt="txt")
```

Get all chromosomes for the human reference dataset in JSON format:

```
>>> g.get_chromosome_names("pub_hs_ens59_grch37")
```

Get the chromosome names from the ChIP-seq peaks dataset gpDS_11_3_7_2 in XML format:

```
>>> g.get_chromosome_names("11_3_7_2", frmt="xml")
```

get_data(*Id*, *frmt=None*, *sep='\t'*, *gz=False*, *ats=None*, *key=None*)
Retrieves data as Plain Text Files (TXT)

Retrieve data from a GeneProf dataset as plain text (optionailly compressed as GZIP). Retrieves the entire contents of an arbitrary GeneProf dataset as a tab-delimited, plain text file. The dataset of interest is identified by its GeneProf accession ID (something of the form gpDS_XXX_XXX_XXX_X). You can get a list of datasets belonging to a certain experiment of interest using the *get_metadata_experiment()* method, or you can use the *search_datasets()* method to query datasets globally.

Parameters

- **id** (*str*) – identifier of the dataset of interest. Either the entire accession ID (e.g. gpDS_11_385_44_1) or just the dataset-specific part (e.g. 11_385_44_1).
- **gz** (*str*) – compressed the data (if format is not rdata).
- **ats** (*str*) – (default displayed columns) A selection of column IDs to be included in the output.
- **sep** (*str*) – (default is tabulation) symbol to be used as a column separator.
- **key** (*str*) – optional WebAPI key, required to access non-public data.

Retrieve data from all visible columns of the dataset gpDS_11_119_18_1 (example RNA-seq data):

```
>>> g.get_data("11_119_18_1", frmt="txt", gz=True)
```

Retrieve only the Ensembl Gene IDs and RPKM values from the same dataset:

```
>>> g.get_data("11_119_18_1", frmt="txt", gz=True,
        ats="C_ENSG,C_11_119_16_1_RPKM0,C_11_119_16_1_RPKM1,C_11_119_16_1_RPKM2,C_11_11
```

get_expression(*ref*, *Id*, *frmt='json'*, *with_sample_info=False*, *output='RPKM'*)
Alias to :meth:‘get_gene_expression

get_external_gene_id(*ref*, *idtype*, *Id*, *frmt='json'*)
Translates a GeneProf gene ID into an external identifier or name.

GeneProf uses sets of gene annotations based on those from Ensembl. With this method, you can look up an external name (official gene symbol) or one of the supported accession ID types (e.g. Ensembl

Gene IDs, RefSeq IDs, etc. – use the *get_list_idtypes()* service to find out which types are supported for a dataset) for any given internal GeneProf gene ID.

Parameters

- **Id** (*str*) – The GeneProf ID of a gene (an integer number).
- **ref** (*str*) – The identifier of a public GeneProf reference dataset. You may use aliases here. Check the *get_list_reference_datasets()* service for all available reference datasets.
- **idtype** (*str*) – The identifier an annotation column storing IDs. Check the *get_list_idtypes()* to find out which types are supported for a dataset.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.

Get the Ensembl Gene ID(s) of the mouse gene #715, as plain text:

```
>>> g.get_external_gene_id("mouse","715", "C_ENSG")
```

Get the RefSeq ID(s) of the human gene #2981, as JSON:

```
>>> g.get_external_gene_id("mouse","2981", "C_RSEQ")
```

Get the name(s) of the human gene #2981, as XML:

```
>>> g.get_external_gene_id("mouse","2981", "C_NAME")
```

get_fasta(*Id*, *key=None*)

Sequence Data as FASTA Files (FASTA, gzipped)

Retrieves the entire contents of a nucleotide sequence dataset in FASTA format. This will only work for dataset of type SEQUENCES, i.e. those containing sequence data! The dataset of interest is identified by its GeneProf accession ID (something of the form gpDS_XXX_XXX_XXX_X). You can get a list of datasets belonging to a certain experiment of interest using the *get_metadata_experiment()* service, or you can use the *search_datasets()* method to query datasets globally. The maximum size of datasets retrieved without an API key is restricted to 10,000,000 entries. With an API key, the maximum size is unlimited.

Parameters

- **Id** (*str*) – The identifier of the dataset of interest. Either the entire accession ID (e.g. gpDS_11_385_44_1) or just the dataset-specific part (e.g.11_385_44_1).
- **key** (*str*) – optional WebAPI key, required to access non-public data.

Retrieve unprocessed Tag-seq sequence data from gpDS_11_385_6_1:

```
>>> g.get_fasta("11_385_6_1")
```

get_fastq(*Id*, *key=None*)

Sequence Data as FASTQ Files (FASTQ)

Same as *get_fasta()* but for FASTQ format.

Retrieve unprocessed Tag-seq sequence data from gpDS_11_385_6_1:

```
>>> g.get_fastq("11_385_6_1")
```

get_gene_expression(*ref*, *Id*, *frmt='json'*, *with_sample_info=False*, *output='RPKM'*)

Get Gene Expression Values for a Gene

Retrieves gene expression values for a gene based on public RNA-seq data in the GeneProf databases.

GeneProf's databases contain many pre-calculated gene expression values stemming from a reanalyses of a large collection of RNA-seq (and similar) experiments. Use this web service to retrieve all the expression values for a single gene of interest by giving the name of the **reference** dataset the gene belongs to and its internal GeneProf gene **Id** – use the *get_list_reference_datasets()*, *get_gene_id()* and/or *search_genes()* methods to look up these identifiers. You may retrieve the values either as raw read counts (the total number of short reads that were aligned to the gene's locus), RPM (reads per million – the raw counts rescaled to account for differences in library size) or RPKM (reads per kilobase million – like RPM, but also accounting for transcript length bias). All gene expression values have been calculated using the Calculate Gene Expression module. Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin).

Parameters

- **Id** (*str*) – GeneProf ID of a gene (an integer number).
- **ref** (*str*) – identifier of a public GeneProf reference dataset. You may use aliases here. Check the list public references service for all available reference datasets.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.
- **type** (*str*) – type of values to obtain, one of: RAW | RPM | RPKM
- **with_sample_info** (*str*) – Include additional annotations about the tissue, cell type, etc. of the expression values.

Retrieve gene expression values for the mouse gene #715, including additional annotation data:

```
>>> g.get_gene_expression("mouse", "715", with_sample_info=True)
```

Retrieve raw read count values for the mouse gene #715

```
>>> g.get_gene_expression("mouse", "715", type="RAW")
```

Retrieve gene expression values for the mouse gene #715 as a tab-delimited text file, including additional annotation data:

```
>>> g.get_gene_expression("mouse", "715", frmt="txt", with_sample_info=True)
```

Retrieve gene expression values for the mouse gene #715 as an RData file, including additional annotation data:

```
>>> g.get_gene_expression("mouse", "715", frmt="rdata", with_sample_info=True)
```

```
>>> from bioservices import GeneProf
>>> import math
>>> from pylab import hist, title, xlabel, clf, show, ylabel
>>> g = GeneProf()
>>> res = g.get_gene_expression("mouse", "715")
>>> rpkmValues = [x["RPKM"] for x in res]
>>> logValues = [math.log(x+1,2.) for x in rpkmValues]
```

```
>>> hist(logValues)
>>> xlabel('RPKM'); ylabel('Count')
>>> title( 'Histogram: 715')
>>> show()
```

get_gene_id(*ref*, *idtype*, *Id*, *frmt='json'*)

Get the GeneProf ID of a Gene

GeneProf uses well-defined sets of gene annotations based on those from Ensembl. you can get the *get_gene_id()* of any gene in the reference annotation by matching it against an external name (official gene symbol) or one of the supported accession ID types (e.g. Ensembl Gene IDs, RefSeq IDs, etc. – use the *get_list_idtypes()* to find out which types are supported for a dataset).

Parameters

- **ref** (*str*) – The identifier of a public GeneProf reference dataset. You may use aliases here. Check the list public references service for all available reference datasets.
- **idtype** (*str*) – The identifier of an annotation column storing IDs or the term any to use any available identifier type. Use *get_list_idtypes()* to find out which types are supported for a dataset.
- **Id** (*str*) – The GeneProf ID of a gene (an integer number).
- **frmt** (*str*) – output format in json, txt, xml, rdata.

Get the GeneProf ID of the mouse gene with Ensembl ID ENSMUSG00000059552

```
>>> g.get_gene_id("mouse", "C_ENSG", "ENSMUSG00000059552")
```

Get the GeneProf IDs of all human genes with RefSeq ID NM_005657, as JSON:

```
>>> g.get_gene_id("human", "C_RSEQ", "NM_005657")
```

Get the GeneProf IDs of all human genes with any ID matching "NM_005657" (should, in this case, be same as the previous query):

```
>>> g.get_gene_id("human", "any", "NM_005657")
```

get_list_experiment_samples(*ref*, *frmt='json'*)

Returns list of Public Experiment Samples for a reference dataset.

All public data in the GeneProf databases has been annotated with the biological sample of origin, described in terms of cell type, tissue, treatment, and so on. This web service simply retrieves a list of all the public sample annotations in the database for a specific reference dataset (see the List Public Reference Datasets service).

Parameters

- **ref** (*str*) – identifier of a public GeneProf reference dataset. you may use aliases here. Check the list public references service for all available reference datasets.
- **format** (*str*) – The file format requests, one of: json, xml, txt, rdata.

Retrieve a list of all samples for mouse as XML:

```
>>> g.get_list_experiment_samples("mouse")
>>> g.get_list_experiment_samples("human", frmt="txt")
>>> g.get_list_experiment_samples("human", frmt="rdata")
```

get_list_experiments (*frmt='json'*, ***kargs*)
Retrieves a list of GeneProf experiments.

Parameters

- **frmt** (*str*) – format of the output
- **with-ats** (*bool*) – Include descriptions for all datasets' annotation types (data columns).
- **with-samples** (*bool*) – Include information about the sample annotation per experiment.
- **with-inputs** (*bool*) – Include a listing of all input datasets per experiment.
- **with-outputs** (*bool*) – Include a listing of the main output datasets per experiment.
- **with-workflow** (*bool*) – Include the analysis workflow per experiment.
- **with-all-data** (*bool*) – Include ALL datasets linked with the experiment (large response!)
- **only-user-experiments** (*bool*) – List only experiments owned by the user identified by the WebAPI key.
- **key** (*bool*) – An optional WebAPI key, required to access non-public data.

Returns if json format is requested, the output is a list of dictionaries. Each dictionary corresponds to one experiment.

```
from bioservices import GeneProf
g = GeneProf(verbose=False)
experiments = g.get_list_experiments(with_outputs=True)
```

get_list_idtypes (*ref*, *frmt='json'*)
list all the ID types available for a dataset.

GeneProf reference datasets provide a number of alternative ID annotations (e.g. Ensembl Gene IDs, RefSeq IDs, UniGene IDs, etc.) for each of the genes in the reference annotation. This service simply lists all the ID types available for a dataset.

Parameters

- **ref** (*str*) – identifier of a public GeneProf reference dataset (e.g. human)
- **frmt** (*str*) – format, one of: json, txt, xml, rdata.

```
>>> g.get_list_idtypes("mouse")
>>> g.get_list_idtypes("human")
```

get_list_reference_datasets (*frmt='json'*)
Retrieves a list of public reference datasets.

GeneProf provides a number of recommended reference datasets for several organisms (human, mouse, etc.). These reference datasets provide genomic sequence assemblies and genic annotations that serve as a scaffold for GeneProf's analyses, so most of GeneProf's datasets are based on one of these reference datasets. This web service simply retrieves a list of all the public, recommended reference datasets currently available in the database.

Parameters **frmt** (*str*) – format in one of: json, xml, txt, rdata.

Note: the txt and rdata format versions of the output reports a flattened version of the dataset metadata and misses out some information available in the other formats!

Retrieve a list of all reference datasets as JSON:

```
>>> g.get_list_reference_datasets()
```

get_metadata_dataset(*Id*, *frmt='json'*, ***kargs*)

Get metadata about geneprof dataset

This method retrieves metadata about a specific GeneProf dataset given the dataset's accession ID (a string of the form gpDS_XXX_XXX_XXX_XXX).

Parameters

- **Id** (*str*) – The identifier of the dataset of interest. Either the entire accession ID (e.g. gpDS_11_385_44_1) or just the dataset-specific part (e.g. 11_385_44_1).
- **frmt** (*str*) – one of json, xml, txt, rdata

Retrieve metadata about the dataset gpDS_11_12_122_1 as JSON:

```
g.get_metadata_dataset("11_12_122_1", with_ats=True)
```

get_metadata_experiment(*Id*, *frmt='json'*, ***kargs*)

Retrieves metadata (names, descriptions, IDs, etc) about experiments.

An experiment typically consists of a set of input data (e.g. raw high-throughput sequencing reads), some experimental sample annotation, an analysis workflow and a selection of main outputs.

Parameters **Id** (*str*) – accession ID (a string of the form gpXP_XXXXXX or a numeric part (e.g., 3)).

Other parameters like in *get_list_experiments()* are also available :return: dictionary with metadata

Retrieve basic metadata about experiment gpXP_000385 including workflow:

```
>>> g.get_metadata_experiment(Id="385", with_workflow=True)
```

get_metadata_usr()

Metadata about a User

get_targets_by_experiment_sample(*ref*, *Id*, *frmt='json'*, *ats='C_NAME'*, *include_unbound=False*)

Get Targets by Experiment Sample

Retrieve putative target genes for a transcription factor (or other transcriptional regulator) based on public ChIP-seq data in the GeneProf databases by querying for the targets discovered in a specific ChIP-seq experiment (identified by the ID of a public sample).

GeneProf's databases contain lots of information about putative gene regulatory interactions from a reanalyses of a large collection of ChIP-seq experiments. You use this web service to retrieve a list of putative target genes for a transcription factor (TF) or other DNA-binding protein (incl. histone modifications), by giving the identifier of a public GeneProf sample – use the *get_list_experiment_samples()* or the *search_samples()* method to look up these identifiers. The assignment of putative target genes to TFs has been done by calling enriched binding peaks on the aligned ChIP-seq reads using MACS and subsequently assigning the peaks to target genes

if they were within a permissible window of the transcription start site (as by current wizard default: 20kb up- and 1kb down-stream of the TSS; in an upcoming release of the web service, you will be able to redefine these threshold dynamically, so watch this space!). The GeneProf workflow modules corresponding to these two steps are documented here: Find Peaks with MACS and Map Regions to Genes. Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin).

Parameters

- **Id** (*str*) – the GeneProf ID of a public sample (an integer number).
- **ref** (*str*) – The identifier of a public GeneProf reference dataset. You may use aliases here. Check the list public references service for all available reference datasets.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.
- **ats** (*str*) – (default C_NAME) A selection of column IDs (from the reference) to be included in the output.
- **include_unbound** (*bool*) – default False. Include not only putative target genes in the output, but also those genes that show now evidence of regulation.

Get all the putative targets of the mouse TF Smad1 in JSON format:

```
>>> g.get_targets_by_experiment_sample("mouse", "541")
```

Get all the putative targets of the human TF MEIS1 in XML format, also include unbound genes for comparison:

```
>>> g.get_targets_by_experiment_sample("human", "784", include_unbound=True)
```

Get all the putative targets of the mouse TF Smad1 as tab-delimited text and include a column for gene name and Ensembl ID:

```
>>> g.get_targets_by_experiment_sample("mouse", "541", ats="C_NAME,C_ENSG")
```

Get all the putative targets of the human TF MEIS1 as an RData file:

```
>>> g.get_targets_by_experiment_sample("human", "784", frmt="rdata")
```

get_targets_tf(*ref*, *Id*, *frmt='json'*, *ats='C_NAME'*, *include_unbound=False*)

Retrieve putative target genes for a transcription factor

Retrieve putative target genes for a transcription factor (or other transcriptional regulator) based on public ChIP-seq data in the GeneProf databases by querying for the targets discovered in all available ChIP-seq experiments (identified by the ID of a gene).

GeneProf's databases contain lots of information about putative gene regulatory interactions from a reanalyses of a large collection of ChIP-seq experiments.

Give the name of the reference dataset the TF gene belongs to and its internal GeneProf gene ID – use the *get_list_reference_datasets()*, *get_gene_id()* or *search_genes()* or *get_gene_id()* methods to look up these identifiers. The assignment of putative target genes to TFs has been done by calling enriched binding peaks on the aligned ChIP-seq reads using MACS and subsequently assigning the peaks to target genes if they were within a permissible window of the transcription start site (as by current wizard default: 20kb up- and 1kb down-stream of the TSS;

in an upcoming release of the web service, you will be able to redefine these threshold dynamically, so watch this space!). The GeneProf workflow modules corresponding to these two steps are documented in Find Peaks with MACS and Map Regions to Genes. Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin). For some TFs there might be more than one dataset available, in which case the output returned by the web service will contain the status in all availabledatasets (distinguished by the experimental sample they belong to, see *get_list_experiment_samples()*.

Parameters

- **Id** (*str*) – The GeneProf ID of a gene/feature (an integer number).
- **ref** (*str*) – identifier of a public GeneProf reference dataset. Check the list public references service for all available reference datasets.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.
- **ats** (*str*) – (default C_NAME) A selection of column IDs (from the reference) to be included in the output.
- **include_unbound** (*bool*) – include not only putative target genes in the output, but also those genes that show now evidence of regulation.

Get all the putative targets of the mouse TF Smad1 in JSON format:

```
>>> g.get_targets_tf("mouse", "9885")
```

Get all the putative targets of the human TF MEIS1, also include unbound genes for comparison:

```
>>> g.get_targets_tf("human", "36958", include_unbound=True)
```

Get all the putative targets of the mouse TF Nanog as include a column for gene name and Ensembl ID (there are TWO ChIP-seq datasets available for this TF!):

```
>>> g.get_targets_tf("mouse", "14899", ats="C_NAME,C_ENSG")
```

Get all the putative targets of the human TF MEIS1 as an RData file:

```
>>> g.get_targets_tf("human", "36958", frmt="rdata")
```

get_tf_by_target_gene(*ref*, *Id*, *frmt='json'*, *ats='C_NAME'*, *with_sample_info=False*)
Get Transcription Factors by Target Gene

Retrieve transcription factors (and other regulatory inputs) putatively targeting a specific gene, based on public ChIP-seq data in the GeneProf databases.

GeneProf's databases contain lots of information about putative gene regulatory interactions from a reanalyses of a large collection of ChIP-seq experiments. You use this web service to retrieve a list of transcription factors and other DNA-binding proteins that might possible be regulating a gene of interest, by giving the name of the reference dataset the gene belongs to and its internal GeneProf gene ID – use the *get_list_reference_datasets()*, *get_gene_id()* or *search_genes()* to look up these identifiers. The assignment of putative target genes to TFs has been done by calling enriched binding peaks on the aligned ChIP-seq reads using MACS and subsequently assigning the peaks to target genes if they were within a permissible window of the transcription start site (as by current wizard default: 20kb up- and 1kb down-stream of the TSS; in an upcoming release of the web service, you will be able to redefine these threshold dynamically, so watch this space!). The GeneProf workflow modules corresponding to these two steps are documented here: Find Peaks with MACS and

Map Regions to Genes. Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin).

Parameters

- **Id** (*str*) – GeneProf ID of a gene (an integer number).
- **ref** (*str*) – The identifier of a public GeneProf reference dataset. see *get_list_reference_datasets()* and *valid_species* for list of public references.
- **frmt** (*bool*) – format requests, one of: json, txt, xml, rdata.
- **with_sample_info** (*bool*) – default false. Include additional annotations about the tissue, cell type, etc. of the expression values.

Get information about factors putatively targeting gene #715 in JSON format, including additional annotation data:

```
>>> g.get_tf_by_target_gene("mouse", "715", with_sample_info=True)
```

Get information about factors putatively targeting gene #715 in XML format, including additional annotation data:

```
>>> g.get_tf_by_target_gene("mouse", "715", frmt="xml",
        with_sample_info=True)
```

Get information about factors putatively targeting gene #715 as a tab-delimited text file:

```
>>> g.get_tf_by_target_gene("mouse", "715", frmt="txt")
```

Get information about factors putatively targeting gene 715 as an RData file, including additional annotation data:

```
>>> g.get_tf_by_target_gene("mouse", "715", frmt="rdata",
        with_sample_info=True)
```

get_tfas_by_gene(*ref*, *Id*, *frmt='json'*, *ats='C_NAME'*, *include_unbound=False*)

Get TFAS of a Transcription Factor

Retrieve transcription factor association strength (TFAS) scores for a transcription factor (or other transcriptional regulator) based on public ChIP-seq data in the GeneProf databases by querying for the data in all available ChIP-seq experiments (identified by the ID of a gene).

Full description: GeneProf's databases contain lots of information about putative gene regulatory interactions from a reanalyses of a large collection of ChIP-seq experiments. You use this web service to retrieve a list of TFAS scores for a transcription factor (TF) or other DNA-binding protein, by giving the name of the reference dataset the TF gene belongs to and its internal GeneProf gene ID – use the list reference datasets, get GeneProf ID and/or search genes services to look up these identifiers. 'TFAS' (= transcription factor association strength) scores are continuous values that give an indication of how strongly a transcription factor (or other DNA-binding protein) is associated with a target gene. The TFAS is calculated as a function of the intensity and the distance of all binding sites (ChIP-seq peaks) near a gene, for details, please refer to the publication by Ouyang et al. (PubMed: 19995984). We use as an intensity score the fold-change enrichment of the ChIP-seq signal over the control background as calculated by MACS in conjunction with calling peaks for the input ChIP-seq data. The GeneProf workflow modules corresponding to these two steps are documented here: Find Peaks with MACS

and Calculate TFAS. Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin). For some TFs there might be more than one dataset available, in which case the output returned by the web service will contain the status in all available datasets (distinguished by the experimental sample they belong to, see list public samples service).

Parameters

- **Id** (*str*) – The GeneProf ID of a gene/feature (an integer number).
- **Ref** (*str*) – The identifier of a public GeneProf reference dataset. You may use aliases here. Check the list public references service for all available reference datasets.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.
- **ats** (*str*) – (default C_NAME) a selection of column IDs (from the reference) to be included in the output.

Get all TFAS scores for the mouse TF Smad1 in JSON format:

```
>>> g.get_tfas_by_gene("mouse", "9885")
```

Get all TFAS scores for the human TF MEIS1 in XML format, also include unbound genes for comparison:

```
>>> g.get_tfas_by_gene("human", "36958", frmt="xml",
        include_unbound=True)
```

Get all TFAS scores the mouse TF Nanog as tab-delimited text and include a column for gene name and Ensembl ID (there are TWO ChIP-seq datasets available for this TF!):

```
>>> g.get_tfas_by_gene("mouse", "14899", frmt="txt",
        ats="C_NAME,C_ENSG")
```

Get all TFAS scores for the human TF MEIS1 as an RData file:

```
>>> g.get_tfas_by_gene("human", "36958", frmt="rdata")
```

get_tfas_by_sample(*ref*, *Id*, *frmt='json'*, *ats='C_NAME'*, *include_unbound=False*)

Get TFAS by Experiment Sample

Retrieve a list of TFAS scores for a transcription factor (TF) or other DNA-binding protein, by giving the identifier of a public GeneProf sample – use the list public samples or the *search_samples()* to look up these identifiers. TFAS (transcription factor association strength) scores are continuous values that give an indication of how strongly a transcription factor (or other DNA-binding protein) is associated with a target gene. The TFAS is calculated as a function of the intensity and the distance of all binding sites (ChIP-seq peaks) near a gene, for details, please refer to the publication by Ouyang et al. (PubMed: 19995984). We use as an intensity score the fold-change enrichment of the ChIP-seq signal over the control background as calculated by MACS in conjunction with calling peaks for the input ChIP-seq data. The GeneProf workflow modules corresponding to these two steps are documented here: Find Peaks with MACS and Calculate TFAS. Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin).

Parameters

- **Id** (*str*) – The GeneProf ID of a public sample (an integer number).

- **ref** (*str*) – identifier of a public GeneProf reference dataset. You may use aliases here. Check the list public references service for all available reference datasets.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.
- **ats** (*str*) – (default C_NAME) a selection of column IDs (from the reference) to be included in the output.

Get TFAS scores for the mouse TF Smad1 in JSON format:

```
>>> g.get_tfas_by_sample("mouse", 541)
```

Get TFAS scores for the human TF MEIS1 in XML format, also include unbound genes for comparison:

```
>>> g.get_tfas_by_sample("human", "784", frmt="xml",
        >>> include_unbound=True)
```

Get TFAS scores for the mouse TF Smad1 as tab-delimited text and include a column for gene name and Ensembl ID:

```
>>> g.get_tfas_by_sample("mouse", "541", frmt="txt",
    ats="C_NAME,C_ENSG")
```

Get TFAS scores for the human TF MEIS1 as an RData file:

```
>>> g.get_tfas_by_sample("human", "784")
```

get_tfas_scores_by_target(*ref*, *Id*, *frmt='json'*, *with_sample_info=False*)

Get TFAS Scores by Target Gene

Rerieve a list of Transcription Factor Association Strength scores quantitating the association between transcription factors (TFs) and other DNA-binding proteins and a gene of interest, by giving the name of the reference dataset the gene belongs to and its internal GeneProf gene ID – use the *get_list_reference_datasets()*, *get_gene_id()*, *search_genes()* to look up these identifiers.

TFAS scores are continuous values that give an indication of how strongly a transcription factor (or other DNA-binding protein) is associated with a target gene. The TFAS is calculated as a function of the intensity and the distance of all binding sites (ChIP-seq peaks) near a gene, for details, please refer to the publication by Ouyang et al. (PubMed: 19995984). We use as an intensity score the fold-change enrichment of the ChIP-seq signal over the control background as calculated by MACS in conjunction with calling peaks for the input ChIP-seq data. The GeneProf workflow modules corresponding to these two steps are documented here: Find Peaks with MACS and Calculate TFAS

Full details for the analysis pipeline that was used to calculate each value are available from the individual experiments the values come from (the JSON and XML output contain a link to the experiment of origin).

Parameters

- **Id** (*str*) – The GeneProf ID of a gene (an integer number).
- **ref** (*str*) – The identifier of a public GeneProf reference dataset. You may use aliases here. Check the list public references service for all available reference datasets.
- **frmt** (*str*) – format requests, one of: json, txt, xml, rdata.

- **with-sample-info** (*bool*) – (default false) Include additional annotations about the tissue, cell type, etc. of the expression values.

Get TFAS scores to gene #715 in JSON format, including additional annotation data:

```
>>> g.get_tfas_scores_by_target("mouse", "715", with_sample_info=True)
```

Get TFAS scores to gene #715 in XML format, including additional annotation data:

```
>>> g.get_tfas_scores_by_target("mouse", 715, with_sample_info=True)
```

Get TFAS scores to gene #715 as a tab-delimited text file:

```
>>> g.get_tfas_scores_by_target("mouse", 715, frmt="txt")
```

Get TFAS scores to gene #715 as an RData file, including additional annotation data:

```
>>> g.get_tfas_scores_by_target("mouse", "715", frmt="rdata",
    with_sample_info=True)
```

get_wig_files(*Id*, *chromosome=None*, *key=None*, *frag_length=-1*, *with_track_description=True*, *only_distinct=False*, *bin_size=25*)

Genomic Data as WIG Files (WIG)

Retrieves the entire contents of a genomic GeneProf dataset in WIG file format. This will only work for dataset of type GENOMIC_REGIONS, i.e. those containing genomic data! The dataset of interest is identified by its GeneProf accession ID (something of the form gpDS_XXX_XXX_XXX_X). You can get a list of datasets belonging to a certain experiment of interest using the *get_metadata_experiment()* service, or you can use the *search_datasets()* method to query datasets globally. The maximum size of datasets retrieved without an API key is restricted to 10,000,000 entries. With an API key, the maximum size is unlimited.

Note: chromosomes in the output BED can be dynamically renamed in order to make the names compatible with other applications (because not all genome databases use the same names, see also the get chromosome names service).

Parameters

- **Id** (*str*) – The identifier of the dataset of interest. Either the entire accession ID (e.g. gpDS_11_385_44_1) or just the dataset-specific part (e.g. 11_385_44_1).
- **chromosome** (*str*) – An optional parameter that may be used to rename chromosomes in the output. The value should be comma-separated map from chromosome ID to its name in the output, where key and value are to be separated with a hyphen (-), e.g. 1-chr1,2-chr2,3-chr12. Any chromosome not mentioned inthe map will not be exported, so you can use this as a filtering mech anism, too. Use the Get Chromosome Names service to get a list of all the available chromosome in a dataset with their default names.
- **with-track-description** (*bool*) – (default True) Include a track description header.
- **only_distinct** (*bool*) – Include only one entry in the coverage count if there are multiple with the same coordinates. (default False)

- **frag-length** (*int*) – (default -1) The "fragment length" to calculate the coverage with, use -1 to use the actual size of the regions.
- **bin-size** (*int*) – (default 25) The bin size / resolution of the tracks.
- **key** (*str*) – An optional WebAPI key, required to access non-public data.

Retrieve genomic coverage data from a RNA-seq assay of gene expression in human liver gpDS_11_58_16_2:

```
>>> g.get_wig_files("11_58_16_2")
```

Retrieve genomic coverage data from a ChIP-seq experiment for Smad1 (gpDS_11_12_112_2), using only distinct alignments:

```
>>> g.get_wig_files("11_12_112_2", with_track_description=False,
    only_distinct=True, frag_length=200)
```

ids_ds

ids_exp

rigid_ids_exp

search_datasets(*query*, *taxons=None*, *frmt='json'*)
search for datasets

Using this web service, you can search for experiments of interest using arbitrarily complex search queries against the names and types of these datasets.

Parameters

- **query** (*str*) – The search term to look for, e.g. a gene name or cell type. You can narrow down the fields to be search by prefixing the query with a field name. Valid fields for samples are: id, label, description, datatype, user, experiment .You can also use boolean logic in your queries using the keywords AND and OR, brackets and quotes (") for exact matches of whole phrases. Advanced search options and examples are documents on GeneProf's search page.
- **frmt** (*str*) – file format in: json, xml, txt, rdata.
- **taxons** (*str*) –

  Only return matches from experiments dealing with organisms matching these NCBI taxonomy IDs (comma-separated list).

  Search for datasets mentioning 'sox2':

```
>>> g.search_datasets('sox2')
```

  Search for datasets mentioning 'gene expression':

```
>>> g.search_datasets('gene expression')
```

Search for genomic data for 'sox2' in plain text format:

```
>>> g.search_datasets("datatype:GENOMIC_REGIONS AND sox2")
```

search_experiments(*query*, *taxons=None*, *frmt='json'*)
Search Experiments using name, description and citations.

Experiments are what GeneProf calls each individual data analysis project. An experiment typically consists of a set of input data (e.g. raw high-throughput sequencing reads), some experimental sample annotation, an analysis workflow and a selection of main outputs. Please check the manual for further information about experiments. Using this web service, you can search for experiments of interest using arbitrarily complex search queries against the names, descriptions, linked citations, linked reference dataset, and so on of those experiments. The search results are categorised by the reference dataset the experiments belong to (also see the List Public Reference Datasets service).

Parameters

- **query** (*str*) – The search term to look for, e.g. a gene name or paper title. You can narrow down the fields to be search by prefixing the query with a field name. Valid fields for experiments are: Valid search fields are: id, label, description, type, reference, user, dataset, citation, platform and sample. You can also use boolean logic in your queries using the keywords AND and OR, brackets and quotes (") for exact matches of whole phrases. Advanced search options and examples are documents on GeneProf's search page.
- **frmt** (*str*) – The file format requests, one of: json, xml, txt, rdata.
- **taxons** (*str*) – Only return matches from experiments dealing with organisms matching these NCBI taxonomy IDs (comma-separated list).

Search for experiments mentioning 'sox2' anywhere (in XML format):

```
>>> g.search_experiments("sox2")
```

Search for experiments mentioning 'cancer' in their description (in JSON format):

```
>>> g.search_experiments("citation:cancer")
```

Search for experiments mentioning 'cell stem cell' in a linked citation (in plain text format):

```
>>> g.search_experiments("citation:'stem cell'")
```

search_gene_ids(*query*, *taxons*)
This is an alias to *search_genes()* to retrieve Ids

geneIds = g.seqrch_gene_ids("nanog", "mouse")

search_genes(*query*, *taxons=None*, *frmt='json'*)
Search genes using genes' description, name and accession IDs.

use sets of gene annotations based on those from Ensembl. You can search for genes of interest using arbitrarily complex search queries against the names and identifiers (from Ensembl, RefSeq and more) of those genes. The search results are categorised by the reference dataset the genes belong to (also see the List Public Reference Datasets service).

Parameters

- **query** (*str*) – The search term to look for, e.g. a gene name or paper title. You can narrow down the fields to be search by prefixing the query with a field name. Valid fields for genes are: Valid search fields are: id, label, description, type and reference. You can also use boolean logic in your queries using the keywords AND and OR, brackets and quotes (") for exact matches of whole phrases.
- **frmt** (*str*) – file format requests, one of: json, xml, txt, rdata.

- **taxons** (*str*) – Only return matches from experiments dealing with organisms matching these NCBI taxonomy IDs (comma-separated list).

1.Search for all genes matching the query 'sox2' (in json format), (2) same but only in human (taxon 9606) (3) same bu only in human and 10090 (4) query "brca2 AND cancer AND reference" in mouse

```
>>> g.search_genes("sox2")['total_results']
8
>>> g.search_genes("sox2", taxons="9606")['total_results']
2
>>> g.search_genes("sox2", taxons="9606, 10090")['total_results']
3
>>> res = g.search_genes("brca2 AND cancer AND reference", taxons="mouse")
```

XML output example:

```
>>> g.search_genes("sox2", taxons="9606", frmt="xml")
>>> geneIds = [x.find('numeric_id').text for x in res.findAll("genes")]
```

```
>>> g.search_genes("sox2", taxons="9606")
>>> geneIds = [x["numeric_id"] for x in res[0]['genes']]
```

See also:

search_gene_ids

search_samples(*query*, *taxons=None*, *frmt='json'*)

search for public experiment samples using search terms against their annotations.

All public data in the GeneProf databases has been annotated with the biological sample of origin, described in terms of cell type, tissue, treatment, and so on. Using this web service, you can search for samples of interest using arbitrarily complex search queries against the annotations of these samples.

Parameters

- **query** (*str*) – search term to look for, e.g. a gene name or cell type. You can narrow down the fields to be search by prefixing the query with a field name. Valid fields for samples are: Valid search fields are: id, label, description , Age, Antibody, Cell_Line, Cell_Type, Description, Developmental_Stage, Gender, Gene, Label, Organism, Platform, Sample_Group, SRA_Accession, Strain, Time, Tissue, Treatment. You can also use boolean logic in your queries using the keywords AND and OR, brackets and quotes (") for exact matches of whole phrases. Advanced search options and examples are documents on GeneProf's search page.
- **frmt** (*bool*) – file format requests, one of: json, xml, txt, rdata.
- **taxons** (*str*) – Only return matches from experiments dealing with organisms matching these NCBI taxonomy IDs (comma-separated list).

Search for samples annotated 'ChIP' in any of the default search fields (in XML format):

```
>>> g.search_samples("ChIP")
```

Search for samples annotated with the gene 'sox2':

```
>>> g.search_samples("Gene:sox2")
```

Search for samples annotated 'human' in any of the default search fields in plain text format:

```
>>> g.search_samples("human")
```

valid_species = ['arabidopsis', 'at', 'ce', 'celegans', 'chick', 'chicken', 'danio', 'dm', 'dmel', 'dr', 'drosphila', 'ef

4.2.13 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
>>> s = QuickGO()
>>> res = s.Term("GO:0003824")
```

Retrieve information about a protein given its uniprot identifier, a taxonomy number. Let us also restrict the search to the UniProt database and print only 3 columns of information (protein name, GO identifier and GO name):

```
print(s.Annotation(protein="Q8IYB3", frmt="tsv", tax=9606,
    source="UniProt", col="proteinName,goID,goName"))
```

Here is the Term output for a given GO identifier:

```
>>> print(s.Term("GO:0000016", frmt="obo"))
[Term]
id: GO:0000016
name: lactase activity
def: "Catalysis of the reaction: lactose + H2O = D-glucose + D-galactose."
synonym: "lactase-phlorizin hydrolase activity" broad
synonym: "lactose galactohydrolase activity" exact
xref: EC:3.2.1.108
xref: MetaCyc:LACTASE-RXN
xref: RHEA:10079
is_a: GO:0004553 ! hydrolase activity, hydrolyzing O-glycosyl compounds
```

Constructor

Parameters **verbose** (*bool*) – print informative messages.

Annotation(*goid=None*, *protein=None*, *frmt='gaf'*, *limit=10000*, *gz=False*, *col=None*, *db=None*, *aspect=None*, *relType=None*, *termUse=None*, *evidence=None*, *source=None*, *ref=None*, *tax=None*, *qualifier=None*, *q=None*, *_with=None*)

Calling the Annotation service

Mutual exclusive parameters are goid, protein

Parameters

- **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).
- **frmt** (*str*) – one of "gaf", "gene2go", "proteinList", "fasta", "tsv" or "dict". The "dict" argument is the default and is a python dictionary.
- **gz** (*bool*) – gzips the downloaded file.
- **goid** (*str*) – GO identifiers either directly or indirectly (descendant GO identifiers) applied in annotations.
- **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.
- **relType** – not Implemented. By default, QuickGO will display annotations to GO terms that are related to that specified in the goid parameter by is_a, part_of and occurs_in relations; this parameter allows you to override that behaviour. See details
- **termUse** – if you set this parameter to slim, then QuickGO will use the supplied set of GO identifiers as a slim and will map the annotations up to these terms. See here for more details: http://www.ebi.ac.uk/QuickGO/GMultiTerm
- **db** (*str*) – protein database (identifier type). Can be UniProtKB, UniGene, Ensembl.
- **evidence** (*str*) – annotation evidence code category (Ev). Example of valid evidence are: be IDA, IC, ISS, IEA, IPI, ND, IMP, ISO, IGI should be either a string with comma separated values (e.g., IEA,IDA) or a list of strings (e.g. ["IEA","IDA"]).
- **source** (*str*) – annotation provider. Examples are 'InterPro', 'UniPathway', 'MGI', 'FlyBase', 'GOC', 'Source', 'UniProtKB', 'RGD', 'ENSEMBL', 'ZFIN', 'IntAct'.
- **ref** (*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'.
- **with** (*str*) – additional supporting information supplied in IEA, ISS, IPI, IC evidenced annotations; see GO documentation. Can refer to a specific identifier or category (for category level, use * after with type). Examples are: EC:2.5.1.30, IPR000092, HAMAP:*
- **qualifier** (*str*) – tags that modify the interpretation of an annotation. Examples are NOT, colocalizes_with, contributes_to.

Note:

•Any number of fields can be specified; they will be AND'ed together.

•Any number of values can be specified for each field; they will be OR'ed together.

•Values should be URI encoded.

•The file will be truncated if more than the specified number of annotations are found. The file is roughly 170 bytes/annotation (not gzipped).

•The file will be gzipped if the gz parameter is supplied.

```
>>> print(s.Annotation(protein='P12345', frmt='tsv', col="ref,evidence",
... ref='PMID:*'))
>>> print(s.Annotation(protein='P12345,Q4VCS5', frmt='tsv',
...     col="ref,evidence",ref='PMID:,Reactome:'))
```

Annotation_from_goid(*goid*, ***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.

Annotation_from_protein(*protein*, ***kargs*)
Returns a DataFrame containing annotation on a given protein

Parameters **protein** (*str*) – a protein name

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.

Term(*goid*, *frmt='oboxml'*)
Obtain Term information

Parameters **frmt** (*str*) – the output format (mini, obo, oboxml).

The format can be:

•mini: Mini HTML, suitable for dynamically embedding in popup boxes.

•obo: OBO format snippet.

•oboxml: OBO XML format snippet.

```
from bioservices import QuickGO
s = QuickGO()
s.Term("GO:0003824")
```

4.2.14 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 imterface 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'
```

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'
```

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.

---

2. 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:

1. gene identifiers. The database used can be one ncbi-gi, ncbi-geneid, uniprot or any KEGG organism
2. 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 suprinsingly, conv("uniprot", "hsa") takes just a few seconds.

Changed in version 1.1: the output is now a dictionary, not a list of tuples

databases
Returns list of valid KEGG databases.

dbinfo(*database=u'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.

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

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")
# 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:G00092")
# same as above
s.get("C01290+G00092")
# retrieves a human gene entry and an E.coli O157 gene entry
s.get("hsa:10458+ece:Z5100")
# retrieves amino acid sequences of a human gene and an E.coli O157 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']
```

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
```

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 O157 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 class
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 O157 gen
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

moduleIds
returns list of module Ids for the default organism.

organism must be set.

```
s = KEGG()
s.organism = "hsa"
s.moduleIds
```

organism
returns the current default organism

organismIds
Returns list of organism Ids

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 os 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'}
```

```
>>> 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)

pathwayIds
returns list of pathway Ids for the default organism.

organism must be set.

```
s = KEGG()
s.organism = "hsa"
s.pathwayIds
```

reactionIds
returns list of reaction Ids

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=u'pink'*, *keggid={}*)
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. Depdending 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")
>>> # Parses a glycan entry
>>> res = s.get("gl:G00001")
>>> # Parses a reaction entry
>>> res = s.get("rn:R00001")
>>> # Parses a rpair entry
>>> res = s.get("rp:RP00001")
>>> # Parses a rclass entry
>>> res = s.get("rc:RC00001")
>>> # Parses an enzyme entry
>>> res = s.get('ec:1.1.1.1')

>>> d = s.parse(res)
```

4.2.15 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 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 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 returns 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 tern 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
>>> 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.

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

4.2.16 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('ncd80')

#  Search for ndc80 and facet with the species field:
```

```
s.search('ncd80', 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:
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'
```

4.2.17 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=True*)
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")
```

`getParameters`()
List parameter names.

Returns An XML document containing a list of parameter names.

```
>>> from bioservices import muscle
>>> n = muscle.Muscle()
>>> res = n.getParameters()
>>> [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.

`getParametersDetails`(*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.getParametersDetails("format")
```

getResult (*jobid*, *resultType*)
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()*.

getResultTypes (*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.

getStatus (*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.

parameters
Read-only attribute that returns a list of parameters. See *getParameters()*.

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 `parametersDetails()`.

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.

4.2.18 Miriam

Interface to the MIRIAM Web Service.

What is Miriam

URL http://www.ebi.ac.uk/miriam/main/
WSDL http://www.ebi.ac.uk/miriamws/main/MiriamWebServices?wsdl
The MIRIAM Registry provides a set of online services for the generation of unique and perennial identifiers, in the form of URIs. It provides the core data which is used by the Identifiers.org resolver. The core of the Registry is a catalogue of data collections (corresponding to controlled vocabularies or databases), their URIs and the corresponding physical URLs or resources. Access to this data is made available via exports (XML) and Web Services (SOAP).
—From MIRIAM Web Site, Feb 2013

Terminology

- URI: Uniform Resource Identifiers
- URL: Uniform Resource Locators

class **Miriam**(*verbose=True*)
Interface to the MIRIAM service

```
>>> from bioservices import Miriam
>>> m = Miriam()
>>> m.getMiriamURI("http://www.ebi.ac.uk/chebi/#CHEBI:17891")
'urn:miriam:chebi:CHEBI%3A17891'
```

Constructor

Parameters

- **verbose** (*bool*) –
- **debug** (*bool*) –
- **url** (*str*) – redefine the wsdl URL

checkRegExp(*identifier*, *datatype*)
Checks if the identifier given follows the regular expression of the data collection

Parameters

- **identifier** (*str*) – internal identifier used by the data collection
- **datatype** (*str*) – name, synonym or MIRIAM URI of a data collection

Returns True if the identifier follows the regular expression, False otherwise

```
>>> m.checkRegExp("P62158", "uniprot")
True
>>> m.checkRegExp("!P62158", "uniprot")
False
```

> **Warning:** there is no sanity check on the datatype. Default output is True. So if you inverse the parameters, you may get True even tough it does not make sense. This is a feature of the Web Service.

convertURL(*url*)
Converts an Identifiers.org URL into its equivalent MIRIAM URN.

This performs a check of the identifier based on the recorded regular expression. :param str identifier: an Identifiers.org URL :return: the MIRIAM URN corresponding to the provided Identifiers.org URL or 'null' if the provided URL is invalid

```
>>> m.convertURL("http://identifiers.org/ec-code/1.1.1.1")
'urn:miriam:ec-code:1.1.1.1'
```

convertURLs(*urls*)
Converts a list of Identifiers.org URLs into their equivalent MIRIAM URNs.

This performs a check of the identifier based on the recorded regular expression.

Parameters **identifier** (*str*) – a list of Identifiers.org URL

Returns the MIRIAM URN corresponding to the provided Identifiers.org URL or 'null' if the provided URL is invalid

```
>>> m.convertURLs(['http://identifiers.org/pubmed/16333295', 'http://identifiers.org/ec
```

convertURN(*urn*)

Converts a MIRIAM URI into its equivalent Identifiers.org URL.

This takes care of any necessary conversion, for example in the case the URI provided is obsolete.

Parameters **urn** (*str*) – a MIRIAM URLs

Returns the Identifiers.org URL corresponding to the provided MIRIAM

URI or None if the provided URI does not exist

```
>>> m.serv.convertURN('urn:miriam:ec-code:1.1.1.1')
'http://identifiers.org/ec-code/1.1.1.1'
```

convertURNs(*urns*)

Converts a list of MIRIAM URIs into their equivalent Identifiers.org URLs.

This takes care of any necessary conversion, for example in the case a URI provided is obsolete. If the size of the list of URIs exceeds 200, None is returned.

Parameters **urns** (*list*) – list of MIRIAM URIs

Returns a list of Identifiers.org URLs corresponding to the provided URIs

```
>>> m.serv.convertURN(['urn:miriam:ec-code:1.1.1.1'])
['http://identifiers.org/ec-code/1.1.1.1']
```

getDataResources(*nickname*)

Retrieves all the physical locations (URLs) of the services providing the data collection (web page).

Parameters **nickname** (*str*) – name (can be a synonym) or URL or URN of a data collection name (or synonym) or URI (URL or URN)

Returns array of strings containing all the address of the main page of the resources of the data collection

```
>>> m.getDataResources("uniprot")
['http://www.ebi.uniprot.org/', 'http://www.pir.uniprot.org/', 'http://www.expasy.unipr
```

getDataTypeDef(*nickname*)

Retrieves the definition of a data collection.

Parameters **nickname** (*str*) – name or URI (URN or URL) of a data collection

Returns definition of the data collection

```
>>> m.getDataTypeDef("uniprot")
'The UniProt Knowledgebase (UniProtKB) is a comprehensive resource for protein sequence
```

getDataTypePattern(*nickname*)

Retrieves the pattern (regular expression) used by the identifiers within a data collection.

Parameters **nickname** (*str*) – data collection name (or synonym) or URI (URL or URN)

Returns pattern of the data collection

```
>>> m.getDataTypePattern("uniprot")
'^([A-N,R-Z][0-9][A-Z][A-Z, 0-9][A-Z, 0-9][0-9])|([O,P,Q][0-9][A-Z, 0-9][A-Z, 0-9][A-Z,
```

getDataTypeSynonyms(*name*)

Retrieves all the synonym names of a data collection (this list includes the original name).

Parameters **name** (*str*) – name or synonym of a data collection

Returns all the synonym names of the data collection (list of strings)

```
>>> m.getDataTypeSynonyms("uniprot")
['UniProt Knowledgebase', 'UniProtKB', 'UniProt']
```

getDataTypeURI(*name*)

Retrieves the unique (official) URI of a data collection (example: "urn:miriam:uniprot").

Parameters **name** (*str*) – name or synonym of a data collection (examples: "UniProt")

Returns unique URI of the data collection

```
>>> m.serv.getDataTypeURI("uniprot")
'urn:miriam:uniprot'
```

getDataTypeURIs(*name*)

Retrieves all the URIs of a data collection, including all the deprecated ones (examples: "urn:miriam:uniprot", "http://www.uniprot.org/", "urn:lsid:uniprot.org:uniprot", ...).

Parameters **name** (*str*) – name (or synonym) of the data collection (examples: "ChEBI", "UniProt")

Returns all the URIs of a data collection (including the deprecated ones)

```
>>> m.getDataTypeURIs("uniprot")
['urn:miriam:uniprot', 'urn:lsid:uniprot.org:uniprot', 'urn:lsid:uniprot.org', 'http://
```

getDataTypesId()

Retrieves the internal identifier (stable and perennial) of all the data collections (for example: "MIR:00000005").

Returns list of the identifier of all the data collections

```
>>> m.getDataTypesId()
```

getDataTypesName()

Retrieves the list of names of all the data collections available.

Returns list of the name of all the data collections

```
>>> m = self.getDataTypesName()
```

getJavaLibraryVersion()

getLocation(*uri*, *resource*)

Retrieves the physical location (URL) of a web page providing knowledge about a specific entity, using a specific resource.

Parameters

- **uri** (*str*) – MIRIAM URI of an entity (example: 'urn:miriam:obo.go:GO%3A0045202')
- **resource** (*sr*) – internal identifier of a resource (example: 'MIR:00100005')

Returns physical location of a web page providing knowledge about the given entity, using a specific resource

```
>>> m.getLocation("UniProt", "MIR:00100005")
['http://www.uniprot.org/uniprot/P62158', 'http://purl.uniprot.org/uniprot/P62158', 'ht
```

Warning: getLocation needs a proper example

getLocations(*nickname*, *Id=None*)

Retrieves the (non obsolete) physical locationS (URLs) of web pageS providing knowledge about an entity.

Parameters

- **nickname** (*str*) – name (can be a synonym) or URI of a data collection (examples: "Gene Ontology", "UniProt"). If Id is None, nickname parameter is a MIRIAM URI of an entity (example: 'urn:miriam:obo.go:GO%3A0045202')
- **id** (*str*) – identifier of an entity within the given data collection (examples: "GO:0045202", "P62158").

Returns physical locationS of web pageS providing knowledge about the given entity. If the URI is not recognised or the data collection does not exist, an empty array is returned. If the identifier is invalid for the data collection, None is returned. All special characters in the data entry part of the URLs are properly encoded.

```
>>> m.getLocations("UniProt","P62158")
>>> m.getLocations("urn:miriam:obo.go:GO%3A0045202")
```

getLocationsWithToken(*nickname*, *token*)

Retrieves the list of (non obsolete) generic physical locations (URLs) of web pageS providing the dataset of a given data collection.

Parameters

- **nickname** (*str*) – name (can be a synonym) or URI of a data collection (examples: "Gene Ontology", "UniProt", "urn:miriam:biomodels.db")
- **token** (*str*) – placeholder which will be put in the URLs at the location of the data entry identifier (default: $id)

Returns list of (non obsolete) generic physical locations (URLs) of web pageS providing the dataset of a given data collection

```
>>> m.getLocationsWithToken("uniprot", "P43403")
```

getMiriamURI(*name*)

Transforms a MIRIAM URI into its official equivalent (to transform obsolete URIs into current valid ones).

Parameters **uri** (*str*) – deprecated URI (URN or URL), example: "http://www.ebi.ac.uk/chebi/#CHEBI:17891"

Returns the official URI corresponding to the deprecated one (for example: "urn:miriam:obo.chebi:CHEBI%3A17891") or 'null' if the URN does not exist

```
>>> m.getMiriamURI("http://www.ebi.ac.uk/chebi/#CHEBI:17891")
'urn:miriam:chebi:CHEBI%3A17891'
```

getName(*uri*)

Retrieves the common name of a data collection

Parameters **uri** (*str*) – URI (URL or URN) of a data collection

Returns the common name of the data collection

```
>>> m.getName('urn:miriam:uniprot')
'UniProt Knowledgebase'
```

getNames(*uri*)

Retrieves all the names (with synonyms) of a data collection.

Parameters **uri** (*str*) – URI (URL or URN) of a data collection

Returns the common name of the data collection and all the synonyms

```
>>> m.serv.getNames('urn:miriam:uniprot')
['UniProt Knowledgebase', 'UniProtKB', 'UniProt']
```

getOfficialDataTypeURI(*nickname*)

Retrieves the official URI (it will always be a URN) of a data collection.

Parameters **nickname** (*str*) – name (can be a synonym) or MIRIAM URI (even deprecated one) of a data collection (for example: "ChEBI", "http://www.ebi.ac.uk/chebi/", ...)

Returns the official URI of the data collection

```
>>> m.getOfficialDataTypeURI("chEBI")
'urn:miriam:chebi'
```

getResourceInfo(*Id*)

Retrieves the general information about a precise resource of a data collection.

Parameters **Id** (*str*) – identifier of a resource (example: "MIR:00100005")

Returns general information about a resource

```
>>> m.getResourceInfo("MIR:00100005")
'MIRIAM Resources (data collection)'
```

getResourceInstitution(*Id*)

Retrieves the institution which manages a precise resource of a data collection.

Parameters **Id** (*str*) – identifier of a resource (example: "MIR:00100005")

Returns institution which manages a resource

```
>>> m.getResourceInstitution("MIR:00100005")
'European Bioinformatics Institute'
```

getResourceLocation(*Id*)

Retrieves the location of the servers of a location.

Parameters **Id** (*str*) – identifier of a resource (example: "MIR:00100005")

Returns location of the servers of a resource

```
>>> m.getResourceLocation("MIR:00100005")
'United Kingdom'
```

getServicesInfo()

Retrieves some information about these Web Services.

Returns information about the Web Services

```
>>> m.getServicesInfo()
```

getServicesVersion()

Retrieves the current version of MIRIAM Web Services.

Returns Current version of the web services

getURI(*name*, *Id*)

Retrieves the unique URI of a specific entity (example: "urn:miriam:obo.go:GO%3A0045202").

If the data collection does not exist (or is not recognised), an empty String is returned.

If the identifier is invalid for the given data collection, 'null' is returned.

Parameters

- **name** (*str*) – name of a data collection (examples: "ChEBI", "UniProt")
- **id** (*str*) – identifier of an entity within the data collection (examples: "GO:0045202", "P62158")

```
>>> m.getURI("UniProt", "P62158")
'urn:miriam:uniprot:P62158'
```

getURIs(*names*, *Ids*)

Retrieves the unique URIs for a list of specific entities (example: "urn:miriam:obo.go:GO%3A0045202").

If a data collection does not exist (or is not recognised), an empty String is returned for this data collection.

If an identifier is invalid for the given data collection, 'null' is returned for this data collection.

If the provided lists do not have the same size, 'null' is returned.

If the size of any list exceeds 200, 'null' is returned.

Returns list of MIRIAM URIs

```
>>> m.serv.getURIs(["UniProt", "GO"], ["P62158", "GO:0045202"])
['urn:miriam:uniprot:P62158', 'urn:miriam:obo.go:GO%3A0045202']
```

Todo

: chracter is not encoded correclty

isDeprecated(*uri*)
Says if a URI of a data collection is deprecated.

Returns answer ("true" or "false") to the question: is this URI deprecated?

```
>>> im.isDeprecated("urn:miriam:uniprot")
False
```

4.2.19 NCBIblast

Interface to the NCBIBLAST web service

What is NCBIBLAST ?

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=True*)
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

databases
Returns accepted databases. Alias to *parametersDetails('database')*

getParameters()
List parameter names.

Returns An XML document containing a list of parameter names.

```
>>> from bioservices import ncbiblast
>>> n = ncbiblast.NCBIblast()
>>> res = n.getParameters()
>>> [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.

getResult(*jobid*, *resultType*)
Get the job result of the specified type.

param str jobid a job identifier returned by *run()*.

pariam str resultType type of result to retrieve. See *getResultTypes()*.

The output from the tool itself. Use the 'format' parameter to retireve 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, 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'.

em_rel_vrl

getResultTypes(*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.

getStatus(*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.

parameters
Read-only attribute that returns a list of parameters. See *getParameters()*.

parametersDetails (*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.parametersDetails("matrix")
[u'BLOSUM45',
 u'BLOSUM50',
 u'BLOSUM62',
 u'BLOSUM80',
 u'BLOSUM90',
 u'PAM30',
 u'PAM70',
 u'PAM250']
```

run (*program=None*, *database=None*, *sequence=None*, *stype='protein'*, *email=None*, ***kargs*)
Submit a job with the specified parameters.

Compulsary arguments

param str program BLAST program to use to perform the search (e.g., blastp)

param str sequence query sequence. The use of fasta formatted sequence is recommended.

param list database 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".

param str email a valid email address. Will be checked by the service itself.

Optional arguments. If not provided, a default value will be used

param str type query sequence type in 'dna', 'rna' or 'protein' (default is protein).

param str matrix scoring matrix to be used in the search (e.g., BLOSUM45).

param bool gapalign perform gapped alignments.

param int alignments maximum number of alignments displayed in the output.

param exp E-value threshold.

param bool filter low complexity sequence filter to process the query sequence before performing the search.

param int scores maximum number of scores displayed in the output.

param int dropoff amount score must drop before extension of hits is halted.

param match_scores match/miss-match scores to generate a scoring matrix for nucleotide searches.

param int gapopen penalty for the initiation of a gap.

param int gapext penalty for each base/residue in a gap.

param seqrange region of the query sequence to use for the search. Default: whole sequence.

return 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 *parametersDetails()*.

For instance,:

```
from bioservices import NCBIblast
n = NCBIblast()
n.parameterDetails("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" than should bie 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.

4.2.20 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,O15117,Q96FE5')
res_many = o.get_interactions(['P00533','O15117','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

4.2.21 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*)
Interface to the PathwayCommons service

```
>>> from bioservices import *
>>> pc2 = PathwayCommons(verbose=False)
>>> res = pc2.get("http://identifiers.org/uniprot/Q06609")
```

Constructor

Parameters **verbose** (*bool*) – prints informative messages

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")
```

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 subnetworks 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_direction` 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")
```

idmapping(*ids*)

Identifier mapping tool

Unambiguously maps, e.g., HGNC gene symbols, NCBI Gene, RefSeq, ENS, and secondary UniProt identifiers to the primary UniProt accessions, or - ChEBI and PubChem IDs to primary ChEBI. You can mix different standard ID types in one query.

Note: this is a specific id-mapping (not general-purpose) for reference proteins and small molecules; the mapping tables were derived exclusively from Swiss-Prot (DR fields) and ChEBI data

---

Parameters **ids** (*str*) – list of Identifiers or a single identifier string.

Returns a dictionary

```
>>> from bioservices import PathwayCommons
>>> pc2 = PathwayCommons(verbose=False)
>>> pc2.idmapping("BRCA2")
{u'BRCA2': u'P51587'}
>>> pc2.idmapping(["TP53", "BRCA2"])
{"BRCA2":"P51587","TP53":"P04637"}
```

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

```
>>> from bioservices import PathwayCommons
>>> pc2 = PathwayCommons(vverbose=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)
```

top_pathways(*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.

Parameters

- **datasource** (*str*) – filter by data source (same as search)
- **organism** (*str*) – organism filter

Returns 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()
```

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 (pathwayComponent) 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 propery 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
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")
```

4.2.22 PDB module

Interface to the PDB web Service.

What is PDB ?

URL http://www.rcsb.org/pdb/
REST http://www.rcsb.org/pdb/software/rest.do
An Information Portal to Biological Macromolecular Structures
—PDB home page, Feb 2013
Status in progress not for production

class PDB(*verbose=False*, *cache=False*)
Interface to part of the PDB service

Status in progress not for production. You can get all ID and retrieve uncompressed file in PDB/FASTA formats for now. New features will be added on request.

```
>>> from bioservices import PDB
>>> s = PDB()
>>> res = s.get_file("1FBV", "pdb")
```

Constructor

Parameters **verbose** (*bool*) – prints informative messages (default is off)

get_current_ids()
Get a list of all current PDB IDs.

get_file(*identifier*, *frmt*, *compression=False*, *headerOnly=False*)
Download a file in a specified format

Parameters

- **identifier** (*int*) – a valid Identifier. See *get_current_ids()*.

- **fileFormat** (*str*) – a valid format in "pdb", "cif", "xml"

```
>>> from bioservices import PDB
>>> s = PDB()
>>> res = s.get_file("1FBV", "pdb")
>>> import tempfile
>>> fh = tempfile.NamedTemporaryFile()
>>> fh.write(res)
>>> # manipulate the PDB file with your favorite tool
>>> # close the file ONLY when finished (this is temporary file)
>>> # fh.close()
```

reference: http://www.rcsb.org/pdb/static.do?p=download/http/index.html

get_ligands(*identifier*)

List the ligands that can be found in a PDB entry

Parameters **identifier** – a valid PDB identifier (e.g., 4HHB)

Returns

xml document

```
>>> from bioservices import PDB
>>> s = PDB()
>>> s.get_ligands("4HHB")
```

Then,

```
x = s.get_ligands("4HHB")
from pyquery import PyQuery as pq
d = pq(x)
```

get_xml_query(*query*)

Send an XML query

query = '<?xml version="1.0" encoding="UTF-8"?> <orgPdbQuery> <version>B0907</version> <queryType>org.pdb.query.simple.ExpTypeQuery</queryType> <description>Experimental Method Search : Experimental Method=SOLID-STATE NMR</description> <mvStructure.expMethod.value>SOLID-STATE NMR</mvStructure.expMethod.value> </orgPdbQuery> '

search(*query*)

<?xml version="1.0" encoding="UTF-8"?> <orgPdbQuery> <version>B0907</version> <queryType>org.pdb.query.simple.ExpTypeQuery</queryType> <description>Experimental Method Search : Experimental Method=SOLID-STATE NMR</description> <mvStructure.expMethod.value>SOLID-STATE NMR</mvStructure.expMethod.value> </orgPdbQuery>

4.2.23 PICR module

This module provides a class *PICR* that allows an access to the REST interface of the PICR web service. There is also a SOAP web service but we implemented only the REST interface since they both provide access to the same functionalities.

What is PICR ?

URL http://www.ebi.ac.uk/Tools/picr
REST http://www/ebi.ac.uk/Tools/picr/rest
Citations http://www.biomedcentral.com/1471-2105/8/401
"The Protein Identifier Cross-Reference (PICR) service is a web application that provides interactive and programmatic (SOAP and REST) access to a mapping algorithm based on 100% sequence identity to proteins from over 98 distinct source databases. Mappings can be limited by source database, taxonomic ID and activity status in the source database. Users can copy/paste or upload files containing protein identifiers or sequences in FASTA format to obtain mappings using the interactive interface. "

—From the PICR home page, Dec 2012

class `PICR`(*verbose=False*, *cache=False*)
Interface to the PICR (Protein Identifier Cross reference) service

```
>>> from bioservices import PICR
>>> p = PICR()
>>> res = p.getMappedDatabaseNames() # get list of valid database
>>> results = p.getUPIForSequence(p._sequence_example,                ["IPI", "ENSEMBL", "
```

Constructor

Parameters **`verbose`** (*bool*) – prints informative messages (default is False)

`databases`
Get a human-readable list of databases (obtained from XML returned by *getMappedDatabaseNames()*)

`getMappedDatabaseNames`()
Returns names of the databases

Returns An XML containing the databases available.

See also:

databases to obtain a human readable list

`getUPIForAccession`(*accession*, *database*, *taxid=None*, *version=None*, *onlyactive=True*, *includeattributes=True*)
Get Protein identifier given an accession number

Parameters

- **`accession`** (*str*) – the accession to map [required]
- **`version`** (*str*) – the version of accession to map [optional]
- **`database`** – the database to map to (string). At least one database is required, but multiple databases can be queried at once using a list.
- **`taxid`** – the NEWT taxon ID to limit the mappings [optional]
- **`onlyactive`** (*bool*) – if true, only active mappings will be returned. If false, results may include deleted mappings. [optional, default is true]
- **`includeattributes`** (*bool*) – if true, extra attributes such as sequence and taxon IDs will be returned if available. If false, no extra information returned. [optional, default is false]

Note: parameter names are case sensitive

Note: If version is not specified but the accession is of the form P29375.1, the accession and version will automatically be split to accession=P29375 and version-1.

Note: If a taxid is submitted, includeattributes will be true.

Example:

```
>>> from bioservices import *
>>> s = PICR()
>>> s.getUPIForAccession("P29375", ["IPI", "ENSEMBL"])
>>> s.getUPIForAccession("P29375-1", ["IPI", "ENSEMBL"])
```

getUPIForBLAST(*blastfrag*, *database*, *taxid=None*, *version=None*, *onlyactive=True*, *includeattributes=False*, ***kargs*)

Get Protein identifier given a sequence similarity (BLAST)

Parameters

- **blastfrag** (*str*) – the AA fragment to map [required]
- **database** (*str*) – the database to map to. At least one database is required, but multiple databases can be queried at once using a list.
- **taxid** – the NEWT taxon ID to limit the mappings [optional]
- **onlyactive** (*bool*) – if true, only active mappings will be returned. If false, results may include deleted mappings. [optional, default is true]
- **includeattributes** (*bool*) – if true, extra attributes such as sequence and taxon IDs will be returned if available. If false, no extra information returned. [optional, default is false]

Other options (related to BLAST analysis) can be provided as optional argument. See this link for details:

http://www.ebi.ac.uk/Tools/picr/RESTDocumentation.do

As an example, you can provide the matrix argument:

Parameters **matrix** (*str*) – specifies which protein scoring matrix to use. [optional, defaults to BLOSUM62]

Note: Parameter names are case sensitive.

Note: If version is not specified but the accession is of the form P29375.1, the accession and version will automatically be split to accession=P29375 and version-1.

Note: If a taxid is submitted, includeattributes will be true.

```
>>> res = s.getUPIForBLAST(s._blastfrag_example, "SWISSPROT")
>>> res = s.getUPIForBLAST(s._blastfrag_example, "SWISSPROT",
      program="blastp", matrix="BLOSUM80")
```

Todo

add missing parameters such as filtertype, blast parameters, identity Value and so on.

getUPIForSequence(*sequence*, *database*, *taxid=None*, *onlyactive=True*, *includeattributes=False*)

Get Protein identifier given an exact sequence

Parameters

- **sequence** – the sequence to map [required]
- **database** – the database to map to. At least one database is required, but multiple databases can be queried at once.
- **taxid** – the NEWT taxon ID to limit the mappings [optional]
- **onlyactive** – if true, only active mappings will be returned. If false, results may include deleted mappings. [optional, default is true]
- **includeattributes** – if true, extra attributes such as sequence and taxon IDs will be returned if available. If false, no extra information returned. [optional, default is false]

Note: Parameter names are case sensitive.

Note: Some servers, browsers and other clients may have restrictions on the length of the query string, so long sequences might cause errors. If this is the case, use a POST request rather than a GET.

Note: If a taxid is submitted, includeattributes will be true.

```
>>> from bioservices import PICR
>>> p = PICR()
>>> sequence = p._sequence_example
>>> databases = ["IPI", "ENSEMBL", "SWISSPROT"]
>>> results = p.getUPIForSequence(sequence, databases)
```

4.2.24 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 nuber

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

4.2.25 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 necesseraly mean protein-protein interactions. In order to be effective, the query format has been standarised.

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 incude 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

Continued on next page

Table 4.1 – continued from previous page

Field Name	Searches on	Implicit*	Example
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"
udate	Update date	Yes	udate:[20100101 TO 20120101]
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 interaction having stoichiometry available	Yes	stc:true
param	Interaction parameters. Only true or false, just to be able to filter interaction 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 possiblity 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:O95169
uniprotkb:P43403
intact:EBI-716238
intact:EBI-1211276
psi-mi:ndub8_human(display_long)|uniprotkb:NADH-ubiquinone oxidoreductase ASHI
.
.
```

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()
```

activeDBs
returns the active DBs only

convert (*data*, *db=None*)

convertAll (*data*)

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 databae is kept. If in the list of DB:ID pairs no known database is found, then we keep the first one whatsover.

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 cleaing by ignoring score, db and interaction len(set([(x[0],x[1]) for x in retnew]))

preCleaning (*data*)
remove entries ehre 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

registry
returns the registry of psicquic

registry_actives
returns active state of each service

registry_counts
returns number of entries in each service

registry_names
returns all services available (names)

registry_restexamples
retuns REST example for each service

registry_restricted
returns restricted status of services

registry_resturls
returns URL of REST services

registry_soapurls
returns URL of WSDL service

registry_versions
returns version of each service

4.2.26 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 are 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 (*version='1.0'*, *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*")
```

See *search() entry()* methods for more information about format.

Rhea constructor

Parameters

- **version** (*str*) – the current version of the interface (1.0)
- **verbose** (*bool*) – True by default

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

entry(*id*, *frmt*)

Retrieve a concrete reaction for the given id in a given format

Parameters

- **id** (*int*) – the id of a reaction
- **format** – can be rxn, biopax2, or cmlreact

Returns An XML document containing the reactions with undefined direction, with links to the corresponding bi-directional ones. The format is easyXML. If frmt is rnx,

```
>>> print(r.entry(10281, frmt="rxn"))
```

The output is in XML format. This page from the Rhea web site explains what are the data fields of the XML file.

search(*query*, *frmt=None*)

Search for reactions

Parameters

- **query** (*str*) – the search term using format parameter
- **format** (*str*) – the biopax2 or cmlreact format (default)

Returns An XML document containing the reactions with undefined direction, with links to the corresponding bi-directional ones. The format is easyXML.

```
>>> r = Rhea()
>>> r.search("caffeine")  # id 10280
>>> r.search("caffeine", frmt="biopax2")  # id 10280
```

The output is in XML format. This page from the Rhea web site explains what are the data fields of the XML file.

4.2.27 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*)
Reactome interface

some data can be download on the main website

`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
- `identfier` (*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:
    f.write(res.decode("base64"))
    f.close()
```

Todo

Saving the image above returns a blank image ...

list_by_query(*classname*, ***kargs*)
Get list of objecs from Reactome database

Parameters

- **class name** (*str*) –
- **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

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

identifiers(*genes*)
s.identfiers("TP53") .. warning:: works for oe gene only for now

4.2.28 Readseq

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

What is Readseq ?

URL http://www.ebi.ac.uk/Tools/sfc/readseq/
Service
Citations http://www.ncbi.nlm.nih.gov/pubmed/18428689
Readseq reads and converts biosequences between a selection of common biological sequence formats, including EMBL, GenBank and fasta sequence formats.
Readseq homepage – Sep 2014

class Readseq(*verbose=True*)
Interface to the Readseq service

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

Constructor

Parameters **verbose** (*bool*) –

get_parameter_details(*parameter*)
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.

get_parameters()
Get a list of the parameter names.

Returns a list of strings giving the names of the parameters.

get_result(*jobid*, *parameters=None*)
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.

Returns the result data for the specified type, base64 encoded. Depending on the SOAP library and programming language used the result may be returned in decoded form. For some result types (e.g. images) this will be binary data rather than a text string.

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.

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).

4.2.29 RNASEQ Analysis

Interface to web service that analysed 200,000 RNA-seq runs in 185 organisms provided by EMBL-EBI Gene Expression Team.

RNASEQ_EBI ?

URL http://www.ebi.ac.uk/~rpetry/geteam/rnaseq/apispec.pdf
Citation http://www.ebi.ac.uk/~rpetry/geteam/rnaseq/apispec.pdf

class RNASEQ_EBI(*verbose=False*, *cache=False*)
Interface to the RNA-SEQ ANALYSIS API service

Example

```
>>> from bioservices import RNASEQ_EBI
>>> r = RNASEQ_EBI()
>>> r.organisms
>>> r.get_run_by_organism('homo_sapiens')
```

See http://www.ebi.ac.uk/~rpetry/geteam/rnaseq/apispec.pdf for the original documentation

Constructor

Parameters **verbose** – set to False to prevent informative messages

get_run(*run_id*, *frmt='json'*, *mapping_quality=70*)

Parameters

- **run_id** – a valid run identifier (e.g., SRR1042759)
- **frmt** – json or tsv
- **mapping_quality** – Min. percentage of reads mapped to genome reference

get_run_by_organism(*organism*, *frmt='json'*, *mapping_quality=70*, *condition=None*)

Parameters

- **organism** – Check the `organism` attributes for valid names
- **frmt** – json or tsv
- **mapping_quality** – Min. percentage of reads mapped to genome reference

- **condition** – check if it exists in EFO (e.g. cancer) (http://www.ebi.ac.uk/efo)

Returns If Pandas is installed and frmt is set to *tsv*, the returned object is a Pandas DataFrame. If Pandas is not available, a list of list is returned. The first element being the names of the field, and each sub list an entry. If *frmt* is set to json, a list of entries is returned, each of them being a dictionary.

Examples:

```
from bioservices import RNASEQ_EBI
r = RNASEQ_EBI()
results = r.get_run_by_organism('oryza_longistaminata', frmt='tsv')
results = r.get_run_by_organism('homo_sapiens', frmt='tsv',
                condition="central nervous system")
```

Table 4.2: Returned fields

name	description
ASSEMBLY_USED	Genome reference assembly name
BIOREP_ID	ENA Run ID or a unique label for tech. replicates in RUN_IDS
ENA_LAST_UPDATED	Date ENA record for was last updated
FTP_LOCATION	FTP location of the CRAM file
LAST_PROCESSED_DATE	Date the run(s) were last analysed
ORGANISM	Organism of samples in SAMPLE_IDS
MAPPING_QUALITY	Percentage of reads mapped to the genome reference
REFERENCE_ORGANISM	Genome reference organism
RUN_IDS	List of ENA Run ID's corresponding to BIOREP_ID
SAMPLE_ATTRIBUTE_TYPE	Matched sample attribute type
SAMPLE_ATTRIBUTE_VALUE	Matched sample attribute value
SAMPLE_IDS	List of BioSamples DB ID's corresponding to BIOREP_ID
STATUS	Processing status
STUDY_ID	ENA Study ID

get_run_by_study(*study*, *frmt='json'*, *mapping_quality=70*)
Access to the RUNS for a given study

Parameters

- **study** – a valid study name eg SRP1042759
- **frmt** – json or tsv
- **mapping_quality** – Min. percentage of reads mapped to genome reference

Returns See *get_run_by_organism()*

Example:

```
r.get_run_by_study("SRP033494", mapping_quality=90, frmt='tsv')
```

get_sample_attribute_coverage_per_study(*study_id*, *frmt='json'*)
Return attributes of a given RUN ID

Param a run ID

Parameters **frmt** – tsv or json

Returns list of entries with the fields as described in *get_sample_attribute_per_run()*

Example:

```
r.get_sample_attribute_per_study("SRP020492")
```

get_sample_attribute_per_run(*run_id*, *frmt='json'*)
Return attributes of a given RUN ID

Param a run ID

Parameters **frmt** – tsv or json

Returns list of entries with the following fields

Table 4.3: Returned fields

name	description
FO_URL	URL of EFO term matching VALUE
RUN_ID	ENA Run ID
STUDY_ID	ENA Study ID
TYPE	Sample Attribute Type
VALUE	Sample Attribute Value
NUM_OF_RUNS	Number of runs annotated with TYPE/VALUE
PCT_OF_ALL_RUNS	Runs annotated with TYPE/VALUE as a percentage of all runs

Example:

```
r.get_sample_attribute_per_run("SRR805786")
```

get_sample_attribute_per_study(*study_id*, *frmt='json'*)
Return attributes of a given RUN ID

Param a run ID

Parameters **frmt** – tsv or json

Returns list of entries with the fields as described in *get_sample_attribute_per_run()*

Example:

```
r.get_sample_attribute_per_study("SRP020492")
```

get_studies_by_organism(*organism*, *frmt='json'*)

Parameters

- **organism** – Check the `organism` attributes for valid names
- **frmt** – json or tsv

If you have Pandas install this code will return the list of valid studies for a given orgnanism:

```
res = r.get_studies_by_organism("arabidopsis_thaliana", frmt='tsv')
studies = res['STUDY_ID'].values
```

Otherwise, if you use the *tsv* format and do not have Pandas installed:

```
res = r.get_studies_by_organism("arabidopsis_thaliana", frmt='tsv')
studies = [x[0] for x in res[1:]]
```

Of with json output:

```
res = r.get_studies_by_organism("arabidopsis_thaliana", frmt='tsv')
studies = [x['STUDY_ID'] for x in res]
```

Note that in addition to study names, each study stores a set of fields as follows:

Table 4.4: Returned fields

name	description
ASSEMBLY_USED	Genome reference assembly name
EXONS_FPKM_COUNTS_FTP_LOCATION	FTP location of exon FPKM counts
EXONS_RAW_COUNTS_FTP_LOCATION	FTP location of exon raw counts
GENES_FPKM_COUNTS_FTP_LOCATION	FTP location of gene FPKM counts
GENES_RAW_COUNTS_FTP_LOCATION	FTP location of genes raw counts
GTF_USED	Ensembl GTF file used for quantification
LAST_PROCESSED_DATE	Date the run(s) were last analysed
ORGANISM	Organism studied in STUDY_ID
REFERENCE_ORGANISM	Genome reference organism
SOFTWARE_VERSIONS_FTP_LOCATION	FTP location of pipeline tools info
STATUS	processing status
STUDY_ID	ENA Study I

get_study(*study*, *frmt='json'*)
Retrieve a study data

Parameters **study** – valid study name (see *get_studies_by_organism()*)

Returns a dictionary if frmt is set to jsonwith fields as described in *get_studies_by_organism()*

Example:

```
r.get_study("SRP033494")
```

organisms
return list of valid organisms

4.2.30 UniChem

This module provides a class *UniChem*

What is UniChem

URL https://www.ebi.ac.uk/unichem/info/webservices
REST https://www.ebi.ac.uk/unichem/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()
```

Constructor UniChem

Parameters **verbose** – set to False to prevent informative messages

get_all_compound_ids_from_all_src_id(*src_compound_id*, *src_id*, *target=None*)
Obtain a list of all src_compound_ids from all sources (including BOTH current AND obsolete assignments) to the same structure as a currently assigned query src_compound_id.

The output will include query src_compound_id if it is a valid src_compound_id with a current assignment. Note also, that by adding an additional (optional) argument (a valid src_id), then results will be restricted to only the source specified with this optional argument.

Parameters

- **src_compound_id** (*str*) – a valid compound identifier (or list)
- **source** – one of the valid database ids. See `source_ids`.
- **target** – if provided, return answer for a specific target database only. Otherwise return answer for all database found in `source_ids`.

Returns list of three element arrays, containing 'src_compound_id' and 'src_id', and 'Assignment', or (if optional 'to_src_id' is specified) list of two element arrays, containing 'src_compound_id' and 'Assignment'.

```
>>> res = s.get_all_compound_ids_from_src_id("CHEMBL12", "chembl")
>>> s.get_all_compound_ids_from_src_id("CHEMBL12", "chembl", "chebi")
[{u'assignment': u'1', u'src_compound_id': u'49575'}]
```

The second call may return an empty list if there is no target from chebi.

get_all_src_ids()
Obtain all src_ids of database currently in UniChem

Returns list of 'src_id's.

```
>>> uni.get_all_src_ids()
```

get_auxiliary_mappings(*src_id*)
For a single source, obtain a mapping between all current src_compound_ids to their corresponding auxiliary data if any.

Some instances of UniChem may contain sources that create URLs for compound-specific pages by using strings or identifiers (called 'auxiliary data' here) that are different to the src_compound_ids for the source. This is not very common, but is dealt with in UniChem by use of an additional mapping step for these sources. This function returns such mapping.

> **Warning:** this method may return very large data sets. you will need to change TIMEOUT to a larger value.

Parameters **src_id** (*int*) – corresponding database identifier (name or id).

Returns list of two element arrays, containing 'src_compound_id' and 'auxiliary data'.

```
>>> uni.get_auxiliary_mappings(15)
```

get_compound_ids_from_src_id(*src_compound_id*, *src_id*, *target=None*)
Obtain a list of all src_compound_ids from all sources which are CURRENTLY assigned to the same structure as a currently assigned query src_compound_id.

The output will include query src_compound_id if it is a valid src_compound_id with a current assignment. Note also, that by adding an additional (optional) argument (a valid src_id), then results will be restricted to only the source specified with this optional argument.

Parameters

- **src_compound_id** (*str*) – a valid compound identifier (list is possible as well)
- **src_id** (*str*) – one of the valid database ids. See `source_ids`.
- **target** (*str, int*) – database identifier (name or id) to map to.

Returns list of dictionaries with the 'src_compound_id' and 'src_id' keys. or (if optional *target* is specified, a list with only 'src_compound_id' keys).

```
>>> get_compound_ids_from_src_id("CHEMBL12", "chembl")
>>> get_compound_ids_from_src_id("CHEMBL12", "chembl", "chebi")
[{'src_compound_id': '49575'}]
```

get_mapping(*source*, *target*)

Obtain a full mapping between two sources. Uses only currently assigned src_compound_ids from both sources.

Parameters

- **source** – name of the source database
- **target** – name of the target database

Returns a dictionary. Keys are the source identifiers. Values are the target identifiers.

```
>>> get_mapping("kegg_ligand", "chembl")
```

get_source_information(*src_id*)

Description: Obtain all information on a source by querying with a source id

Parameters **src_id** (*int*) – valid identifiers (values or keys of `source_ids` e.g. chebi, chembl,0,1). could also be a list of those identifiers.

Returns

dictionary (or list of dictionaries) with following keys:

- src_id (the src_id for this source),
- src_url (the main home page of the source),
- name (the unique name for the source in UniChem, always lower case),
- name_long (the full name of the source, as defined by the source),
- name_label (A name for the source suitable for use as a 'label' for the source within a web-page. Correct case setting for source, and always less than 30 characters),
- description (a description of the content of the source),
- base_id_url_available (an flag indicating whether this source provides a valid
- base_id_url for creating cpd-specific links [1=yes, 0=no]).
- base_id_url (the base url for constructing hyperlinks to this source [append an
- identifier f rom this source to the end of this url to create a valid url to a
- specific page for this cpd], unless aux_for_url=1),
- aux_for_url (A flag to indicate whether the aux_src field should be used to create hyperlinks instead of the src_compound_id [1=yes, 0=no]

```
>>> res = get_source_information("chebi")
```

get_src_compound_id_url(*src_compound_id*, *src_id*, *to_src_id*)

Obtain a list of URLs for all src_compound_ids

Obtain a list of URLs for all src_compound_ids from a specifiedsource (the 'to_src_id'), which are CURRENTLY assigned to the same structure as a currently assigned query src_compound_id. Method only applicable for sources which support direct URLs to src_compound_id pages. Method also applicable for 'to_src_id's where the hyperlink is constructed from auxiliary data [and not from the src_compound_id] as per example2 below.

Parameters

- **src_compound_id** (*str*) – a valid compound identifier
- **src_id** (*int*) – corresponding database identifier (name or id).
- **to_src_id** (*str*) – database identifier (name or id) to map to.
- **to_src_id** –

Returns list of URLs.

```
>>> uni.get_src_compound_id_url("CHEMBL12", "chembl", "drugbank")
>>> # equivalent to
>>> uni.get_src_compound_id_url("CHEMBL12", 1, 2)
```

get_src_compound_ids_all_from_inchikey(*inchikey*)

Description: Obtain a list of all src_compound_ids (from all sources) which have current AND obsolete assignments to a query InChIKey

Parameters **inchikey** (*str*) – input source identified by its InChiKey (or list) (or list of list of dictionaries if input is a list).

Returns list of two element arrays, containing 'src_compound_id' and 'src_id'. and 'Assignment'.

```
>>> uni.get_src_compound_ids_all_from_inchikey("AAOVKJBEBIDNHE-UHFFFAOYSA-N")
```

get_src_compound_ids_all_from_obsolete(*obsolete_src_compound_id*, *src_id*, *to_src_id=None*)

Obtain a list of all src_compound_ids from all sources with BOTH current AND obsolete to the same structure with an obsolete assignment to the query src_compound_id.

The output will include query src_compound_id if it is a valid src_compound_id with an obsolete assignment. Note also, that by adding an additional (optional) argument (a valid src_id), then results will be restricted to only the source specified with this optional argument.

Parameters

- **src_compound_id** (*str*) – a valid compound identifier
- **src_id** (*int*) – corresponding database identifier (name or id).
- **to_src_id** (*int*) – database identifier (name or id) to map to.

Returns list of four element arrays, containing 'src_compound_id', 'src_id', 'assignment' and 'UCI', or (if optional 'to_src_id' is specified) list of three element arrays, containing 'src_compound_id', 'Assignment' and 'UCI'.

```
>>> from bioservices import UniChem
>>> u = UniChem()
>>> u.get_src_compound_ids_all_from_obsolete("DB07699", "2")
>>> u.get_src_compound_ids_all_from_obsolete("DB07699", "2", "1")
```

get_src_compound_ids_all_from_src_compound_id(*src_compound_id*, *src_id*, *target=None*)

get_src_compound_ids_from_inchikey(*inchikey*)

Obtain a list of all src_compound_ids (from all sources) which are CURRENTLY assigned to a query InChIKey

Parameters **inchikey** (*str*) – input source identified by its InChiKey (or list)

Returns list of dictionaries containing 'src_compound_id' and 'src_id' keys (or list of list of dictionaries if input is a list).

```
>>> uni.get_src_compound_ids_from_inchikey("AAOVKJBEBIDNHE-UHFFFAOYSA-N")
```

get_src_compound_ids_from_src_compound_id(*src_compound_id*, *src_id*, *target=None*)

get_structure(*src_compound_id*, *src_id*)

Obtain structure(s) CURRENTLY assigned to a query src_compound_id.

Parameters

- **src_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")
```

get_structure_all(*src_compound_id*, *src_id*)

Obtain structure(s) with current AND obsolete assignments

Parameters

- **src_compound_id** (*str*) – a valid compound identifier
- **src_id** (*int*) – corresponding database identifier (name or id).

Returns dictionary with 'standardinchi', 'standardinchikey' and 'assignment' keys

```
>>> uni.get_structure_all("CHEMBL12", "chembl")
```

get_verbose_src_compound_ids_from_inchikey(*inchikey*)

Obtain all src_compound_ids (from all sources)

which are CURRENTLY assigned to a query InChIKey. However, these are returned as part of the following data structure: A list of sources containing these src_compound_ids, including source description, base_id_url, etc. One element in this list is a list of the src_compound_ids currently assigned to the query InChIKey.

Parameters **inchikey** (*str*) – input source identified by its InChiKey

Returns

list containing

- src_id (the src_id for this source),
- src_url (the main home page of the source),
- name (the unique name for the source in UniChem, always lower case),
- name_long (the full name of the source, as defined by the source),
- name_label (A name for the source suitable for use as a 'label' for the source within a web-page. Correct case setting for source, and always less than 30 characters),
- description (a description of the content of the source),
- base_id_url_available (an flag indicating whether this source provides a valid
- base_id_url for creating cpd-specific links [1=yes, 0=no]).
- base_id_url (the base url for constructing hyperlinks to this source [append an
- identifier from this source to the end of this url to create a valid url to a specific pag e for this cpd], unless aux_for_url=1),
- aux_for_url (A flag to indicate whether the aux_src field should be used to create hyperlinks instead of the src_compound_id [1=yes, 0=no] ,
- src_compound_id (a list of src_compound_ids from this source which are currently assigned to the query InChIKey.
- aux_src (a list of src-compound_id keys mapping to corresponding auxiliary data (url_id:value), for creating links if aux_for_url=1. Only shown if aux_for_url=1).

```
>>> uni.get_verbose_src_compound_ids_from_inchikey("QFFGVLORLPOAEC-SNVBAGLBSA-N")
>>> # Note that this one is not valid anymore
>>> uni.get_verbose_src_compound_ids_from_inchikey("ZUITABIAKMVPG-UHFFFAOYSA-N")
```

4.2.31 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

Identifiers mapping between databases:

```
>>> from bioservices import UniProt
>>> u = UniProt(verbose=False)
>>> u.mapping("ACC", "KEGG_ID", 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")
```

Constructor

Parameters **verbose** – set to False to prevent informative messages

get_df(*entries*, *nChunk=100*, *organism=None*)
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** –

Returns dataframe with indices being the uniprot id (e.g. DIG1_YEAST)

Todo

cleanup the content of the data frame to replace strings separated by ; into a list of strings. e.g. the Gene Ontology IDs

---

Warning: requires pandas library

get_fasta(*id_*)
Returns FASTA string given a valid identifier

See also:

bioservices.apps.fasta for dedicated tools to manipulate FASTA

get_fasta_sequence(*id_*)
Returns FASTA sequence (Not FASTA)

Parameters **id** (*str*) – Should be the entry name

Returns returns fasta sequence (string)

Warning: this is the sequence found in a fasta file, not the fasta content itself. The difference is that the header is removed and the formatting of end of lines every 60 characters is removed.

mapping(*fr='ID'*, *to='KEGG_ID'*, *query='P13368'*)
This is an interface to the UniProt mapping service

Parameters

- **fr** – the source database identifier. See _mapping.
- **to** – the targetted database identifier. See _mapping.
- **query** – a string containing one or more IDs separated by a space It can also be a list of strings.

- **format** – The output being a dictionary, this parameter is deprecated and not used anymore

Returns a list. The first element is the source database Id. The second is the targetted source identifier. Following elements are alternate of one the entry and its mapped Id. If a query has several mapped Ids, the query is repeated (see example with PDB mapping here below) e.g., [”From:ID”, “to:PDB_ID”, “P43403”]

```
>>> u.mapping("ACC", "KEGG_ID", 'P43403')
defaultdict(<type 'list'>, {'P43403': ['hsa:7535']})
>>> u.mapping("ACC", "KEGG_ID", 'P43403 P00958')
defaultdict(<type 'list'>, {'P00958': ['sce:YGR264C'], 'P43403': ['hsa:7535']})
>>> u.mapping("ID", "PDB_ID", "P43403")
defaultdict(<type 'list'>, {'P43403': ['1FBV', '1M61', '1U59',
'2CBL', '2OQ1', '2OZO', '2Y1N', '3ZNI', '4A4B', '4A4C', '4K2R']})
```

There is a web page that gives the list of correct database identifiers. You can also look at the `_mapping` attribute.

URL http://www.uniprot.org/mapping/

Changed in version 1.1.1: to return a dictionary instaed 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=False*, *sort='score'*, *limit=None*)
a specialised version of *search()*

This is equivalent to:

```
u = uniprot.UniProt()
u.search(query, frmt="tab", include=False, sor="score", limit=None)
```

Returns a dictionary.

retrieve(*uniprot_id*, *frmt='xml'*, *database='uniprot'*)
Search for a uniprot ID in UniProtKB database

Parameters

- **uniprot** (*str*) – a valid UniProtKB ID or a list of identifiers.
- **frmt** (*str*) – expected output format amongst xml, txt, fasta, gff, rdf

Returns is 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="xml")
>>> fasta = u.retrieve([u'P29317', u'Q5BKX8', u'Q8TCD6'], frmt='fasta')
>>> print(fasta[0])
```

search(*query*, *frmt='tab'*, *columns=None*, *include=False*, *sort='score'*, *compress=False*, *limit=None*, *offset=None*, *maxTrials=10*, *database='uniprot'*)
Provide some interface to the uniprot search interface.

Parameters

- **query** (*str*) – query must be a valid uniprot query. See http://www.uniprot.org/help/text-search, http://www.uniprot.org/help/query-fields See also example below
- **frmt** (*str*) – a valid format amongst html, tab, xls, asta, 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 fomat is tab or xls. For UnitProtKB, some possible columns are: id, entry name, length, organism. Some column name must be followed by database name (e.g., "database(PDB)"). Again, see uniprot website for more details. See also `_valid_columns` for the full list of column keyword.
- **include** (*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:9606', frmt='list')
>>> u.search("zap70+and+taxonomy:9606", frmt="tab", limit=3,
...    columns="entry name,length,id, genes")
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:9606", limit=3, columns="id,database(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="tab", limit=3,
...    columns="entry name,length,id, genes")
Entry name  Length  Entry   Gene names
```

Warning: this function request seems a bit unstable (UniProt web issue ?) so we repeat the request if it fails

Warning: some columns although valid may not return anything, not even in the header: 'score', 'taxonomy', 'tools'. this is a uniprot feature, not bioservices.

searchUniProtId(*uniprot_id*, *frmt='xml'*)

uniref(*query*)
Calls UniRef service

Returns if you have Pandas installed, returns a dataframe (see example)

```
>>> u = UniProt()
>>> df = u.uniref("member:Q03063")  # of just A03063
>>> df.Size
```

4.2.32 wsdbfetch

Interface to WSDbfetch web service

What is WSDbfetch

URL http://www.ebi.ac.uk/Tools/webservices/services/dbfetch
Service http://www.ebi.ac.uk/Tools/webservices/services/dbfetch_rest
"WSDbfetch 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 `WSDbfetch`(*verbose=False*)
Interface to WSDbfetch service

```
>>> from bioservices import WSDbfetch
>>> w = WSDbfetch()
>>> data = w.fetchBatch("uniprot" ,"zap70_human", "xml", "raw")
```

The actual URL used is http://www.ebi.ac.uk/ws/services/WSDbfetchDoclit?wsdl from biocatalogue (this one having let functionalities: http://www.ebi.ac.uk/ws/services/WSDbfetch?wsdl).

Constructor

Parameters **verbose** (*bool*) – print informative messages

fetchBatch(*db*, *ids*, *format='default'*, *style='default'*)
Fetch a set of entries in a defined format and style.

Parameters

- **db** (*str*) – the name of the database to obtain the entries from (e.g. 'uniprotkb').
- **query** (*list*) – list of identifiers (e.g. 'wap_rat, wap_mouse').
- **format** (*str*) – the name of the format required.
- **style** (*str*) – the name of the style required.

Returns

The format of the response depends on the interface to the service used:

- WSDBFetchServerService and WSDBFetchDoclitServerService: The entries as a string.
- WSDBFetchServerLegacyService: An array of strings containing the entries.

```
from bioservices import WSDbfetch
u = WSDbfetch()
u.fetchBatch("uniprot" ,"wap_mouse", "xml")
```

```
```

fetchData(*query*, *format='default'*, *style='default'*)
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.
- **style** (*str*) – the name of the style required.

Returns

The format of the response depends on the interface to the service used:

- WSDBFetchServerService and WSDBFetchDoclitServerService: The entries as a string.
- WSDBFetchServerLegacyService: An array of strings containing the entries. Generally this will contain only one item which contains the set of entries.

```
from bioservices import WSDbfetch
u = WSDbfetch()
u.fetchData('uniprot:zap70_human')
```

getDatabaseInfo(*db*)
Get details describing specific database (data formats, styles)

Parameters **db** (*str*) – a valid database.

Returns The output can be introspected and contains several attributes (e.g., displayName).

```
>>> res = u.getDatabaseInfo("uniprotkb")
>>> res.displayName
'UniProtKB'
>>> print(res.description.encode('utf-8'))
u'The UniProt Knowledgebase (UniProtKB) is the central access point for extensive curat
```

getDatabaseInfoList()
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.

getDbFormats(*db*)
Get list of format names for a given database.

Parameters **db** (*str*) –

getFormatStyles(*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.getFormatStyles("uniprotkb", "fasta")
['default', 'raw', 'html']
```

getSupportedDBs()
Get a list of database names usable with WSDbfetch.

Buffered in _supportedDB.

getSupportedFormats()
Get a list of database and format names usable with WSDbfetch.

Deprecated since version use: Of getDbFormats(db), getDatabaseInfo(db) or getDatabaseInfoList().

getSupportedStyles()
Get a list of database and style names usable with WSDbfetch.

Deprecated since version use: Of getFormatStyles(db, format), getDatabaseInfo(db) or getDatabaseInfoList() is recommended.

Returns: An array of strings containing the database and style names.

supportedDBs
Alias to getSupportedDBs.

supportedFormats
Get a list of database and format names usable with WSDbfetch.

Deprecated since version use: Of getDbFormats(db), getDatabaseInfo(db) or getDatabaseInfoList().

supportedStyles
Get a list of database and style names usable with WSDbfetch.

Deprecated since version use: Of getFormatStyles(db, format), getDatabaseInfo(db) or getDatabaseInfoList() is recommended.

Returns: An array of strings containing the database and style names.

4.2.33 Wikipathway

Interface to the WikiPathway service

What is WikiPathway ?

URL http://www.wikipathways.org/index.php/WikiPathways
WSDL http://www.wikipathways.org/index.php/Help:WikiPathways_Webservice/API
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 Wikipathway

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:

•getXrefList: Neither WSDL or REST seemed to work

•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.
- **WSAuth auth** (*object*) – The authentication info.

Returns WSPathwayInfo The pathway info for the created pathway (containing identifier, revision, etc.).

displaySavedPathwayInBrowser(*filename*)
Show a saved document in a browser.

Parameters **filename** (*str*) –

Returns Nothing

Note: Method from Bioservices. Not a WikiPathways function.

---

findInteractions(*query*, *organism=None*, *interactionOnly=True*, *raw=False*)
Find interactions defined in WikiPathways pathways.

Parameters

- **query** (*str*) – The name of an entity to find interactions for (e.g. 'P53')
- **organism** (*str*) – The name of the organism to refine the search (default is the :attr: 'organism' attribute).
- **interactionOnly** (*bool*) – Returns only the interactions (default). If false, returns also scores, revision, pathways

- **raw** (*bool*) – If True, returns the output of the request without post processing (also ignoring organism)

Returns Depends on the parameters **raw** and **interactionOnly**. By default, the output from WikiPathways is processed and only the interactions are returned (for the default organism). You can change this behaviour by changing the default arguments (raw, organism, interactionOnly)

Warning: Interface different from the service, unless raw is set to True

findPathwaysByLiterature(*query*)

Find pathways by their literature references.

Parameters **query** (*str*) – The query, can be a pubmed id, author name or title keyword.

Returns Array of WSSearchResult descr=The search results. {{{descr}}}

```
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/

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')
```

findPathwaysByXref(*ids*)

Find pathways by searching on the external references of DataNodes.

This function supports only 1 ids and 1 code at a time. To specify multiple ids and codes parameters to query for multiple xrefs at once, the REST syntax should be used. In that case, the number of ids and codes parameters should match, they will be paired to form xrefs, e.g.:

```
>>> from bioservices import RESTService
>>> r = RESTService("test", url=s.url[:-5])
>>> r.request('/findPathwaysByXref?ids=1234&ids=ENSG00000130164&codes=L&codes=EnHs')
>>> r.request('/findPathwaysByXref?ids=1234&codes=L')
```

Parameters

- **string ids** (*str*) – One DataNode identifier(s) (e.g. 'P45985'). Datanodes can be (gene/protein/metabolite identifiers).
- **codes** (*str*) – One code of the database system to limit the search to. **Not implemented**.

Returns List of WSSearchResult. An array of search results with DataNode GraphId stored in the 'field' hash.

```
>>> s.findPathwaysByXref(ids="P45985")
```

Warning: **codes** is not available. Does not seem to work in WSDLinterface.

getColoredPathway (*pathwayId*, *filetype='svg'*, *revision=0*)
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.

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")
```

getOntologyTermsByOntology (*ontologyTerm*)
Get a list of ontology terms from a given ontology.

Parameters **ontology** (*str*) – The ontology term (for possible values, see the Ontology Tags section on the pathway page at WikiPathways website.

Returns List of WSOntologyTerm The ontology terms.

```
>>> from bioservices import WikiPathways
>>> s = WikiPathway()
>>> s.getOntologyTermsByOntology("Disease")
```

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.

```
s.getPathway("WP2320")
```

getPathwayAs(*pathwayId*, *filetype='owl'*, *revisionNumb=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

Warning: Argument pathwayId and filetype are inversed as compared to the WSDL prototype (if you want to call it directly)

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:** Does not seem to work with WSDL. Replaced by a REST version but 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(verbose=False)
>>> s.getPathwayInfo("WP2320")
```

getPathwaysByOntologyTerm(*ontologyTermId*)
Get a list of pathways tagged with a given ontology term.

Parameters **ontologyTermId** (*str*) – the ontology term identifier.

Returns List of WSPathwayInfo. The pathway information.

```
>>> from bioservices import WikiPathways
>>> s = Wikipathway()
>>> s.getPathwaysByOntologyTerm('DOID:344')
```

getPathwaysByParentOntologyTerm(*ontologyTermId*)
Get a list of pathways tagged with any ontology term that is the child of the given Ontology term.

Parameters **ontologyTermId** (*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: yyyymmddMMHHSS (string or number)

Returns The changed pathways

```
s.getRecentChanges(20110101000000)
```

listPathways(*organism=None*)
Get a list of all available pathways.

Parameters **organism** (*str*) – a valid organism (default is the *organism* attribute)

Returns List of pathways for the selected organism.

login(*usrname*, *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 excecute 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.

organism
Read/write attribute for the organism

organisms = None
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*, *revisionNumb*)
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*, *revisionNumb=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

---

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.

4.3 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.

4.3.1 Peptides

class Peptides(*verbose=False*)

```
>>> p = Peptides()
>>> p.get_fasta_sequence("Q8IYB3")
>>> p.get_peptide_position("Q8IYB3", "VPKPEPIPEPKEPSPE")
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*)

4.3.2 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
MKNKVVVVTGVPGVGGTTLTQKTIEKLKEEGIEYKMVNFGTVMFEVAKEEGLVEDRDQMR
KLDPDTQKRIQKLAGRKIAEMAKESNVIVDTHSTVKTPKGYLAGLPIWVLEELNPDIIVI
VETSSDEILMRRLGDATRNRDIELTSDIDEHQFMNRCAAMAYGVLTGATVKIIKNRDGLL
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 encompase:

- **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
- **Identifiifers** is the begining 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.

PE
: returns PE keyword found in the header if any

SV
: returns SV keyword found in the header if any

accession

dbtype

df

entry
: returns entry only

fasta
returns FASTA content

gene_name
returns gene name from GN keyword found in the header if any

get_fasta(*id_*)
Fetches FASTA from uniprot and loads into attrbiute *fasta*

Parameters **id** (*str*) – a given uniprot identifier

Returns the FASTA contents

header
returns header only

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

name

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

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
```

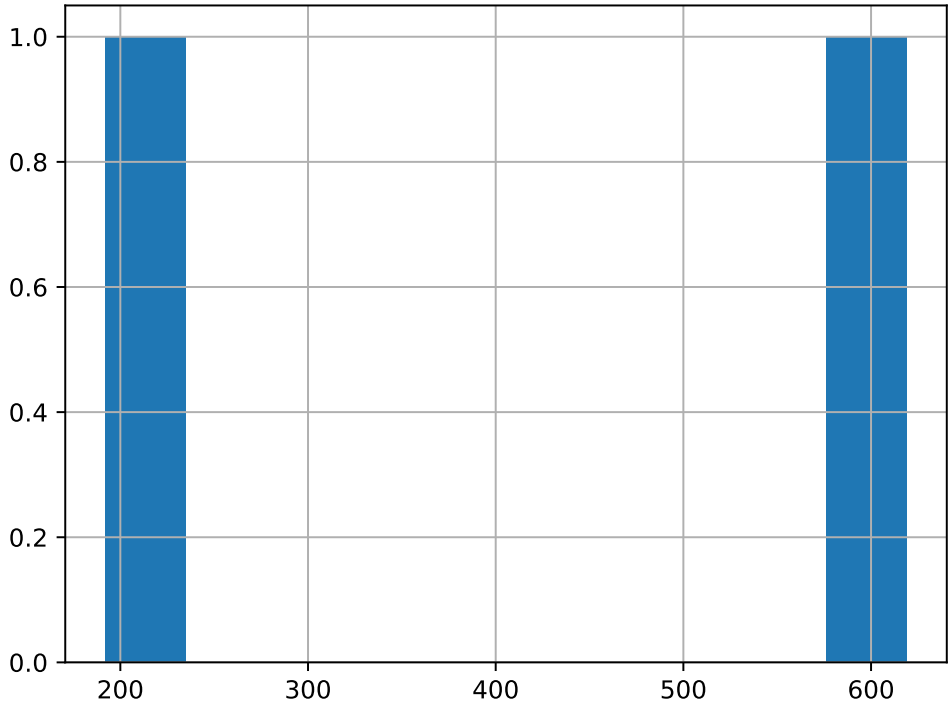
Each FASTA is stored in *fasta*, which is a dictionary where each values is an instance of *FASTA*:

```
>>> print(mf._fasta["P43408"].fasta)
>sp|P43408|KADA_METIG Adenylate kinase OS=Methanotorris igneus GN=adkA PE=1 SV=2
MKNKVVVVTGVPGVGGTTLTQKTIEKLKEEGIEYKMVNFGTVMFEVAKEEGLVEDRDQMR
KLDPDTQKRIQKLAGRKIAEMAKESNVIVDTHSTVKTPKGYLAGLPIWVLEELNPDIIVI
VETSSDEILMRRLGDATRNRDIELTSDIDEHQFMNRCAAMAYGVLTGATVKIIKNRDGLL
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()
```



df

fasta
Returns all FASTA instances

hist_size(**args*, ***kwds*)

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

CHAPTER 5

Others

5.1 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

5.2 FAQS

5.2.1 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

5.2.2 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)

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")
```

5.2.3 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.

5.2.4 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.

5.3 Whats' new, what has changed

5.3.1 Revision 1.4

- 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_eutils
        * 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

5.3.2 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 * clinvitae: 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 clinvitae 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 urrlib2
        * 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)
    * chembldb 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.
    * chembldb:
      · 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 ba 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

5.3.3 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 ito 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 where performing a search
        * added the BioDBnet service.
        * added Pathway Common
        * 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 time-out/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

5.3.4 Revision 1.1

- **1.1.3**
  - **fix bug in chembldb.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 loggin.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

5.3.5 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**
    - 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.

5.3.6 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/chembldb 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

5.3.7 Revision 0.9

- **0.9.7:**
    - **add new feature in KEgg module to instrospect kgml data sets**
        * add biogrid test and documentation.
        * chembldb improvments
        * 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, PSICQUIC, Wikipath

5.3.8 Up to Revision 0.5

- 0.4.9: finalise wikipathway
- 0.4.8: finalise doc of half of the services.
- 0.4.7: add psicquic 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 servie 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 rhea service + test. Updating the documentation.
- 0.3.0: add reactome + uniprot.
- 0.2.0: finalise biomodels and add picr service + test for biomdodel service..
- 0.1.0: add database and kegg modules + its documentation and tests

5.4 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:

- Sven-Maurice Althoff, Christian Knauth implemented the `bioservices.muscle` module.
- Patrick Short implemented the `bioservices.clinvitae` module

And thank you also to the contributions from users who have sent communication via emails or via the ticket system.

Note that originally code (and earlier tickets) were hosted elsewhere.

A

B

C

D

E

F

G

H

S

T

U

V

W