pymangal Documentation

Release 0.1

Timothée Poisot

July 15, 2015

Contents

`pymangal` is a library to interact with APIs returning ecological interaction networks datasets in the format specified by the `mangal` data specification. More informations on `mangal` can be found **here**.

The `pymangal` module provides way to browse, search, and get data, as well as to upload or patch them.

Data in the `mangal` database are released under the *Creative Commons 0* waiver. Anyone is free to access and use them. Note that the usual rules of good conduct among academics apply, and you are expected to credit data collectors by citing either the dataset, or the original publication. These informations are available in the `dataset` object.

CHAPTER 1

User guide

These pages will cover the use of various aspect of the `pymangal` module.

1.1 pymangal 101

This document provides an overview of what the `pymangal` module can do, and more importantly, how to do it.

1.1.1 Overview of the module

Installation

At the moment, the simplest way to install `pymangal` is to download the latest version from the *GitHub* repository, using *e.g.*:

```
wget https://github.com/mangal-wg/pymangal/archive/master.zip
unzip master.zip
cp pymangal-master/pymangal .
rm -r pymangal-master
```

Then from within the `pymangal` folder,

```
make requirements
make test
make install
```

Alternatively, `make all` will download the requirements, run the tests, and install the module. Note that by default, the `makefile` calls `python` and `pip`. If your versions of python 2 and pip are called, *e.g.*, `python27` and `pip2`, you need to pass them as variable names when calling make:

```
make all pip=pip2 python=python27
```

Creating a mangal object

Almost all of the actions you will do using `pymangal` will be done by calling various methods of the `mangal` class. The usual first step of any script is to import the module.

```
>>> import pymangal as pm
>>> api = pm.mangal()
```

Calling `dir(api)` will give you an overview of the methods and attributes.

APIs conforming to the `mangal` specification can expose either all resources, or a subset of them. To see which are available,

```
>>> api.resources
```

For each value in the previously returned list, there is an element of

```
>>> api.schemes
```

This dictionary contains the `json` scheme for all resources exposed by the API. This will both give you information about the data format, and be used internally to ensure that the data you upload or patch in the remote database are correctly formatted.

1.1.2 Getting a list of resources

`mangal` objects have a `List()` method that will give a list of entries for a type of resource. For example, one can list datasets with:

```
>>> api.List('dataset')
```

The returned object is a `dict` with keys `meta` and `objects`. `meta` is important because it allows paging through the resources, as we will see below. The actual content you want to work with is within `objects`; `objects` is an array of `dict`.

Paging and offset

To preserve bandwidth (yours and ours), `pymangal` will only return the first 10 records. The `meta` dictionary will give you the `total_count` (total number of objects) available. If you want to retrieve *all* of these objects in a single request, you can use the `page='all'` argument to the `List()` method.

```
>>> api.List('taxa', page='all')
```

If you want more that 10 records, you can pass the number of records to `page`:

```
>>> api.List('network', page=20)
```

An additional important attribute of `meta` is the `offset`. It will tell you how many objects were discarded before returning your results. For example, the following code

```
>>> t_1_to_4 = api.List('taxa', page=4, offset=0)
>>> t_5_to_8 = api.List('taxa', page=4, offset=4)
```

is (roughly, you still would have to recompose the object) equivalent to

```
>>> t_1_to_8 = api.List('taxa', page=8)
```

Filtering

There is a special page on *filtering*. When filtering, it is recommended to use `page='all'`, as it will ensure that all matched results are returned.

Note: Unless you *absolutely* need to get all the data, it makes very little sense to `List` all of the database. It is assumed that filters *will* be used most of the time.

1.1.3 Getting a particular resource

Getting a particular resource required that you know its type, and its unique identifier. For example, getting the `taxa` with `id` equal to 8 is

```
>>> taxa_8 = api.Get('taxa', 8)
```

The object is returned *as is*, *i.e.* as a python `Dict`. If there is no object with the given `id`, or no matching `type`, then the call to `Get` will fail.

1.1.4 Creating and modifying resources

There is a page dedicated to **contributing_**. Users with data that they want to add to the *mangal* database are invited to read this page, which gives informations about (1) how to register online and (2) how to prepare data for upload.

1.2 Filtering of resources

This document covers the different ways to filter resources using the `List` method.

1.2.1 General filtering syntax

Filtering follows the general syntax:

```
field__relation=target
```

`field` is the name of one of the fields in the resource model (see `mg.schemes[resource]['properties'].keys()`). `relation` is one of the ten possible values given below. Finally, `target` is the value to match. It is possible to join several filters, by joining them with `&`.

Note that if the `target` contains spaces, they will be automatically changed to `%20`, so you won't have to worry about that.

Examples

Let's start by loading the module:

```
>>> import pymangal as pm
>>> api = pm.mangal()
```

Getting all taxa whose name contains "alba":

```
>>> api.List('taxa', filters='name__contains=alba', page='all')
```

Getting the dataset containing network "101":

```
>>> api.List('dataset', filters='networks__in=101', page='all')
```

Getting all networks with "benthic" in their name, between latitudes "-5" and "5":

```
>>> api.list('network', filters='name__contains=bentic&latitude__range=-5,5', page='all')
```

1.2.2 Type of relationships

relation	description
`startswith`	All fields starting by the target
`endswith`	All fields ending by the target
`exact`	Exact matching
`contains`	Fields that contain the target
`range`	Fields with values in the range
`gt`	Field with values greater than the target
`lt`	Field with values smaller than the target
`gte`	Field with values greater (or equal to) than the target
`lte`	Field with values smaller (or equal to) than the target
`in`	Field with the target among their values

1.2.3 Filtering through multiple resources

It is possible to combine several resources when filtering. For example, if one want to retrieve populations belonging to the taxa *Alces americanus*, the syntax is

```
taxa__name__exact=Alces%20americanus
```

Examples

List of populations whose taxa is of the genus "Alces":

```
>>> api.List('population', filters='taxa__name__startswith=Alces', page='all')
```

List of interactions involving "Canis lupus" as a predator

```
>>> api.List('interaction', filters='link_type__exact=predation&taxa_from__name__exact=Canis%20lupus
```

1.3 How to upload data

This page will walk you through the upload of a simple food web with three species. The goal is to cover the basic mechanisms. Posting data requires to be authenticated. Users can register at <http://mangal.io/dashboard/>. Authentication is done with the username and API key.

To upload data, a good knowledge of the data specification is important. `JSON` schemes are imported when connecting to the database the first time

```
>>> import pymangal as pm
>>> api = pm.mangal(usr='myUserName', key='myApiKey')
>>> api.schemes.keys()
```

Sending data into the database is done though the `Post` method of the `mangal` class. The `Post` method requires two arguments: `resource` and `data`. `resource` is the type of object you are sending in the database, and `data` is the object as a python `dict`.:

```
>>> my_taxa = {'name': 'Carcharodon carcharias', 'vernacular': 'Great white shark', 'eol': 213726, 's
>>> great_white = api.Post('taxa', my_taxa)
```

The `mangal` API is configured so that, when data are received or modified, it will *return* the database record created. It means that you can assign the result of calling `Post` to an object, for easy re-use. For example, we can now create a population belonging to this taxa:

```
>>> my_population = {'taxa': great_white['id'], 'name': 'Amity island sharks'}
>>> amity_island = api.Post('population', my_population)
```

Note: In the `rmangal` package, it is possible to pass whole objects rather than just `id` to the function to patch and post. This is not the case with `pymangal`.

1.3.1 Example: a linear food chain

In this exercice, we will upload a linear food chain made of a top predator (*Canis lupus*), a consumer (*Alces americanus*), and a primary producer (*Abies balsamea*).

The first step is to create objects containing the taxa:

```
>>> wolf = {'name': 'Canis lupus', 'vernacular': 'Gray wolf', 'status': 'confirmed'}
>>> moose = {'name': 'Alces americanus', 'vernacular': 'American moose', 'status': 'confirmed'}
>>> fir = {'name': 'Abies balsamea', 'vernacular': 'Balsam fir', 'status': 'confirmed'}
```

Now, we will take each of these objects, and send them into the database:

```
>>> wolf = api.Post('taxa', wolf)
>>> moose = api.Post('taxa', moose)
>>> fir = api.Post('taxa', fir)
```

The next step is to create interactions between these taxa:

```
>>> w_m = api.Post('interaction', {'taxa_from': wolf['id'], 'taxa_to': moose['id'], 'link_type': 'pre
>>> m_b = api.Post('interaction', {'taxa_from': moose['id'], 'taxa_to': fir['id'], 'link_type': 'herb
```

That being done, we will now create a network with the different interactions:

```
>>> net = api.Post('network', {'name': 'Isle Royale National Park', 'interactions': map(lambda x: x[
```

The last step is to put this network into a dataset:

```
>>> ds = api.Post('dataset', {'name': 'Test dataset', 'networks': [net['id']]})
```

And with these steps, we have (i) created taxa, (ii) established interactions between them, (iii) put these interactions in a network, and (iv) created a dataset.

1.3.2 Other notes

Conflicting names

The `mangal` API will check for the uniqueness of some properties before writing the data. For example, no two taxa can have the same name, of taxonomic identifiers. If this happens, the server will throw a `500` error, and the error message will tell you which field is not unique. You can then use the **filtering_** abilities to retrieve the pre-existing record.

Automatic validation

So as to avoid sending "bad" data on the database, `pymangal` conducts an automated validation of user-supplied data *before* doing anything. In case the data are not properly formatted, a `ValidationError` will be thrown, along with an explanation of (i) which field(s) failed to validate and (ii) what acceptable values were.

Resource IDs and URIs

The `pymangal` module will, internaly, take care of replacing objects identifiers by their proper URIs. If you want to make a reference to the `taxa` whose `id` is `1`, the `Post` method will automatically convert `1` to `api/v1/taxa/1/`, *i.e.* the format needed to upload.

CHAPTER 2

Developer guide

These page give the complete reference of the `pymangal` module. They are mostly intended for people wanting to know *how the sausage is made* (with heavy chucks of JSON).

2.1 The mangal class

The `mangal` class is where most of the action happens. Almost all user actions consist in calling various methods of this class.

2.1.1 Documentation

class `pymangal.`**`mangal`**(*url='http://mangal.io'*, *suffix='/api/v1/'*, *usr=None*, *key=None*)

Creates an object of class `mangal`

This is the main class used by `pymangal`. When called, it will return an object with all methods and attributes required to interact with the database.

Parameters

- **`url`** – The URL of the site with the API (default: `http://mangal.io`)
- **`suffix`** – The suffix of the API (default: `/api/v1/`)
- **`usr`** – Your username on the server (default: `None`)
- **`key`** – Your API key on the server (default: `None`)

Returns An object of class `mangal`

`Get`(*resource='dataset'*, *key='1'*)

Get an object identified by its key (id)

Parameters

- **`resource`** – The type of object to get
- **`key`** – The unique identifier of the object

Returns A `dict` representation of the resource

`List`(*resource='dataset'*, *filters=None*, *page=10*, *offset=0*)

Lists all objects of a given resource type, according to a filter

Parameters

- **resource** – The type of resource (default: `dataset`)
- **filters** – A string giving the filtering criteria (default: `None`)
- **page** – Either an integer giving the number of results to return, or `'all'` (default: `10`)
- **offset** – Number of initial results to discard (default: `0`)

Returns A `dict` with keys `meta` and `objects`

Note: The `objects` key of the returned dictionary is a `list` of `dict`, each being a record in the database. The `meta` key contains the `next` and `previous` urls, and the `total_count` number of objects for the request.

Patch(*resource='taxa'*, *data=None*)
Patch a resource in the database

Parameters

- **resource** – The type of object to patch
- **data** – The dict representation of the object

The value of the `owner` field will be preserved, i.e. the owner of the object is not changed when patching the data object. This is important for users to find back the objects they uploaded even though they have been curated.

This method converts the fields values to URIs automatically.

If the request is successful, this method will return the newly created object. If not, it will print the reply from the server and fail.

Post(*resource='taxa'*, *data=None*)
Post a resource to the database

Parameters

- **resource** – The type of object to post
- **data** – The dict representation of the object

The `data` may or may not contain an `owner` key. If so, it must be the URI of the owner object. If no `owner` key is present, the value used will be `self.owner`.

This method converts the fields values to URIs automatically

If the request is successful, this method will return the newly created object. If not, it will print the reply from the server and fail.

2.2 Checks of user-supplied arguments

Several methods share arguments, so it made sense to have a set of functions designed to validate the in the same place. These functions are all in `pymangal.checks`, and are used internally only by the different methods.

2.2.1 Documentation

`pymangal.checks.`**check_resource_arg**(*api*, *resource*)
Checks that the `resource` argument is correct

Parameters

- **api** – A `mangal` instance
- **resource** – A user-supplied argument (tentatively, a string)

Returns Nothing, but fails if `resource` is not valid

So as to be valid, a `resource` argument *must*

•be of type `str`

•be included in `api.resources`, which is collected from the API root

`pymangal.checks.`**check_upload_res**(*api*, *resource*, *data*)

Checks that the data to be uploaded are in the proper format

Parameters

- **api** – A `mangal` instance
- **resource** – A resource argument
- **data** – The data to be uploaded. This is supposed to be a dict.

Returns `data`, with corrections applied if need be.

The first checks are basic:

•the user must provide authentication

•the data must be given as a dict

The next check concers data validity, i.e. they must conform to the data schema in json, as obtained from the API root when calling `__init__`.

`pymangal.checks.`**check_filters**(*filters*)

Checks that the filters are valid

Parameters **filters** – A string of filters

Returns Nothing, but can modify `filters` in place, and raises ‘‘ValueError‘‘s if the filters are badly formatted.

This functions conducts minimal parsing, to make sure that the relationship exists, and that the filter is generally well formed.

The `filters` string is modified in place if it contains space.

2.3 Various helper functions

These functions are called (internally) to (i) make the code more readable and (ii) automate some parts of the data handling. These functions are all in `pymangal.helpers`.

2.3.1 Documentation

`pymangal.helpers.`**uri_from_username**(*api*, *username*)

Returns a URI from a username

Parameters

- **api** – An API object
- **username** – The username for which you want the URI, as a string

Returns The URI as a string, and raises `ValueError` if there is no known user, `TypeError` if the arguments are not in the correct type.

`pymangal.helpers.`**`prepare_data_for_patching`**(*api*, *resource*, *data*)

`pymangal.helpers.`**`check_data_from_api`**(*api*, *resource*, *data*)

`pymangal.helpers.`**`keys_to_uri`**(*api*, *resource*, *data*)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

C

G

K

L

M

P

U