Yatel Documentation

Release 0.4dev

Yatel Team

September 02, 2014

Contents

Figure 1: **Version:** 0.4dev

Contents:

CHAPTER 1

Ubuntu/Debian/Mint

Execute

```
$ apt-get install python-dev libatlas-base-dev gfortran
$ pip install yatel
```

Development version

```
$ pip install --pre yatel
```

CHAPTER 2

Windows or other xnix

- Python 2.7 http://www.python.org
- Setup tools http://pypi.python.org/pypi/setuptools
- Mercurial (if you install yatel from the repo) http://mercurial.selenic.com/
- Scipy http://scipy.org/scipylib/index.html
- NumPy http://numpy.scipy.org/

Finally open a console and execute

```
> easy_install pip
> pip install yatel
```

For development version

```
> pip install --pre yatel
```

CHAPTER 3

From repo

First install all dependencies, and then

```
$ hg clone http://bitbucket.org/yatel/yatel yatel
$ cd yatel
$ python setup.py sdist
$ pip install dist/yatel-<VERSION>.tar.gz
```

CHAPTER 4

Install as develop

```
$ hg clone http://bitbucket.org/yatel/yatel yatel
$ cd yatel
$ pip install -r requirements.txt
$ python setup.py develp
```

CHAPTER 5

Command Line Interface

For common maintenance, startup and general configuration Yatel has a comfortable set of commands for use from console.

Some use cases for better understanding exemplified below.

Yatel has three global options:

1. `-k|--ful-stack` indicates that if a command fails, show full exception output and not just the error message.
2. `-l|--log` enables log of the database to standard output.
3. `-f|--force` if a database is tried to be opened in `w` or `a` and a Yatel Network is discovered overwrite it.
4. `-h|--help` show the help os all yatel or a single command.

5.1 Commands

- `version`: Print the Yatel version and exit.

```
$ yatel version
0.3
```

- `list`: Lists all available connection strings in yatel.

```
$ yatel list
sqlite: sqlite:///${database}
memory: sqlite://
mysql: mysql://${user}:${password}@${host}:${port}/${database}
postgres: postgres://${user}:${password}@${host}:${port}/${database}
```

- `describe`: Receives a single parameter that is the connection string to the database and prints on the screen description of the network.

```
$ yatel describe sqlite:///my_nwwharehouse.db
{description}
```

- `test`: Runs all tests of Yatel. Receives a single parameter that refers to the level of verbosity [0|1|2] of the tests.

```
$ yatel test 1
....
```

- `dump`: Persists all data from a nwolap to a file in *JSON* or *XML* format. It is important to note that *JSON* is very fast but memory intensive for large networks; so *JSON* is recommended for small networks for large networks use *XML*. Dump receives two parameters:

1. URI of the database to dump
2. The name of the file where the information will be dumped. The persistence format is given by the extension of the file. To use JSON the extension must be **.json** or **.yjf** and for XML extensions are **.xml** or **.yxf**

```
$ yatel dump sqlite:///my_nwwharehouse.db dump.xml
```

- `backup`: Similar to dump and it does the same function. The only difference that the file name to be parameterized as target is not the final name but a template, between the name and the extension ALWAYS a timestamps is added to always create a new file. It is useful for automated backup tasks.

```
$ yatel backup sqlite:///my_nwwharehouse.db backup.xml
```

- `load`: Restores data from a file created by the `dump` or `backup` command. The first parameter of the command is the target database. The second parameter is the open mode of the db, `w` (erases previous contents) or `a` (adds new content to the network) and the third it's a path to the file with the data.

```
$ yatel load sqlite:///my_nwwharehouse.db a backup.xml
```

- `copy`: Copy an entire nwolap into another nwolap. The command takes as first parameter the URI of the source network, the second parameter is the open mode of the db that can be `w` (erases previous content) or `a` (adds new content to the network) and the third one it is the URI of destination network.

```
$ yatel copy sqlite:///my_nwwharehouse.db w mysql://user:password@host:port/copy_nwwharehouse
```

- **`pyshell`: Opens a Python interpreter (Ipython or Bpython if posible)** with the context set with the NWOLAP given as parameter.

```
$ yatel pyshell sqlite:///my_nwwharehouse.db

    Welcome to Yatel Interactive mode.
    Yatel is ready to use. You only need worry about your project.
    If you install IPython, the shell will use it.
    For more info, visit http://getyatel.org/
    Available modules:
        Your NW-OLAP: nw
        from yatel: db, dom, stats
        from pprint: pprint

>>>
```

- **`qbjshell`: Opens a QBJ interpreter with the context set with the NWOLAP** given as parameter.

```
$ yatel qbjshell sqlite:///my_nwwharehouse.db
Yatel QBJ Console

QBJ [sqlite://***/my_nwwharehouse.db]>
```

- `createconf`: create a new configuration to run Yatel as a service in JSON format. Receives as a parameter the name of the file to create. (For the syntax of this file see:)

```
$ yatel createconf my_new_conf.json
```

- `createwsgi`: Create a new wsgi file to deploy Yatel to a server in production mode. Receives two parameters: The first should be an absolute path (preferably), to where the configuration file was created with the command `createconf` and second the name of the wsgi file.

```
$ yatel createwsgi my_new_conf.json my_new_wsgi.py
```

- `runserver`: Runs Yatel as an HTTP service. Receives two parameters: The first is the path to the configuration file created with `createconf` command and the second `IP` and `port` where the service will be listening separated by a `:`

```
$ yatel runserver my_new_conf.json localhost:8080
```

- `createetle`: Create a new file to extract, transform, and load data (ETL) in a path specified as a parameter.

```
$ yatel createetl my_new_etl.py
```

- `describeetl`: Describe the documentation and parameters of the ETL constructor passed as a parameter.

```
$ yatel describeetl my_new_etl.py
```

- `runetl`: Runs an ETL. Receives three parameters.

  1. Destination database
  2. Open mode of the databse (`w` o `a`)
  3. ETL path

  Keep in mind that the ETL may receive more parameters in its constructor; to be passed after the path to the ETL.

```
$ yatel runetl sqlite:///my_nwwharehouse.db a my_new_etl.py param param param
```

CHAPTER 6

Quick Start

6.1 What is Yatel?

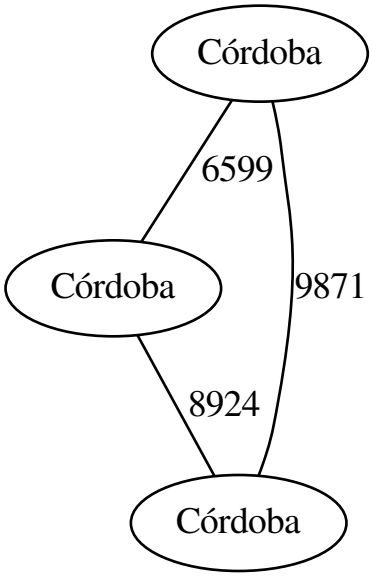
It's a reference implementation of NW-OLAP

- Wiskey-Ware License
- It is largely implementing the aforementioned process.
- Soon to arrive it's first usable version 0.3

Read more about Yatel.

6.2 Case study (example)

Suppose we have the following problem:



We have three places called Cordoba [1] [2] [3], each separated one from the other by a certain distance. We can use Yatel to state the problem and make queries:

- Which ones have an area between 200km2 and 600km2?
- Which ones speak Spanish?
- Those with the time zone utc-6?
- Who has in his name Andalucía?

6.3 Loading problem into Yatel

We load the previous model into Yatel, as follows:

```
>>> from yatel import dom, db
>>> from pprint import pprint
# postgres, oracle, mysql, and many more
>>> nw = db.YatelNetwork("memory", mode="w")
>>> elems = [
...     dom.Haplotype(0, name="Cordoba"), # left
...     dom.Haplotype(1, name="Cordoba"), # right
...     dom.Haplotype(2, name="Cordoba"), # bottom
...
...     dom.Edge(6599, (0, 1)),
...     dom.Edge(8924, (1, 2)),
...     dom.Edge(9871, (2, 0)),
```

[1] http://en.wikipedia.org/wiki/C%C3%B3rdoba,_Argentina
[2] http://en.wikipedia.org/wiki/C%C3%B3rdoba,_Veracruz
[3] http://en.wikipedia.org/wiki/C%C3%B3rdoba,_Andalusia

```
...
...     dom.Fact(0,name="Andalucia", lang="sp", timezone="utc-3"),
...     dom.Fact(1, lang="sp"),
...     dom.Fact(1, timezone="utc-6"),
...     dom.Fact(2, name="Andalucia", lang="sp", timezone="utc"),
... ]
>>> nw.add_elements(elems)
>>> nw.confirm_changes()
```

In the above code, we create a database in memory and define:

- A haplotype for each Córdoba.
- An edge to match each Córdoba by a distance.
- Facts that give us information about the haplotypes.

6.3.1 Models and Attributes

Showing the description

```
>>> descriptor =  nw.describe()
>>> pprint(dict(descriptor))
{'edge_attributes': {u'max_nodes': 2, u'weight': <type 'float'>},
 'fact_attributes': {'hap_id': <type 'int'>,
                     'lang': <type 'str'>,
                     'name': <type 'str'>,
                     'timezone': <type 'str'>},
 'haplotype_attributes': {'hap_id': <type 'int'>, 'name': <type 'str'>},
 'mode': 'r',
 'size': {u'edges': 3, u'facts': 4, u'haplotypes': 3}
}
```

Showing Haplotypes:

```
>>> for hap in nw.haplotypes():
...     print hap
<Haplotype (0) at 0x24faa50>
<Haplotype (1) at 0x24eae50>
<Haplotype (2) at 0x24fa990>
```

Showing Edges:

```
>>> for edge in nw.edges():
...     print edge
<Edge ([6599.0 [0, 1]]  ) at 0x1f64c50>
<Edge ([8924.0 [1, 2]]  ) at 0x24fa0d0>
<Edge ([9871.0 [2, 0]]  ) at 0x1f64c50>
```

Showing Facts:

```
>>> for fact in nw.facts():
...     print fact
<Fact (of Haplotype '0') at 0x24eae50>
<Fact (of Haplotype '1') at 0x24fad10>
<Fact (of Haplotype '1') at 0x24eae50>
<Fact (of Haplotype '2') at 0x24fad10>
```

6.3.2 Query

Now for the queries:

```
>>> hap = nw.haplotype_by_id(2)
>>> hap
<Haplotype (2) at 0x24fa990>
```

Edges by haplotype:

```
>>> for edge in nw.edges_by_haplotype(hap):
...     print edge
<Edge ([9871.0 [2, 0]]  ) at 0x24fa710>
<Edge ([8924.0 [1, 2]]  ) at 0x1f64c50>
```

Facts by haplotype:

```
>>> for fact in nw.facts_by_haplotype(hap):
...     print dict(fact)
{u'lang': u'sp', u'timezone': u'utc', 'hap_id': 2, u'name': u'Andalucia'}
```

Haplotypes by lang environment:

```
>>> for hap in nw.haplotypes_by_environment(lang="sp"):
...     print hap
<Haplotype (0) at 0x24fa2d0>
<Haplotype (1) at 0x25c5350>
<Haplotype (2) at 0x24fa2d0>
```

Haplotypes by timezone environment:

```
>>> for hap in nw.haplotypes_by_environment(timezone="utc-6"):
...     print hap
<Haplotype (1) at 0x24eae50>
```

Haplotypes by name environment:

```
>>> for hap in nw.haplotypes_by_environment(name="Andalucia"):
...    print hap
<Haplotype (0) at 0x25c5350>
<Haplotype (2) at 0x24eae50>
```

Edges by Andalucia environment:

```
>>> for edge in nw.edges_by_environment(name="Andalucia"):
...     print edge
<Edge ([9871.0 [2, 0]]  ) at 0x24fa7d0>
```

All environments:

```
>>> for env in nw.environments():
...     print env
<Enviroment {u'lang': u'sp', u'timezone': u'utc-3', u'name': u'Andalucia'} at 0x24faad0>
<Enviroment {u'lang': u'sp', u'timezone': None, u'name': None} at 0x24db490>
<Enviroment {u'lang': None, u'timezone': u'utc-6', u'name': None} at 0x24faad0>
<Enviroment {u'lang': u'sp', u'timezone': u'utc', u'name': u'Andalucia'} at 0x24db490>
```

6.3.3 Statistics

Here are some statistics:

```
>>> from yatel import stats

>>> stats.average(nw) # average
8464.66666667

>>> stats.std(nw, name="Andalucia")
0.0
```

6.3.4 Data Mining

Now to some data mining:

```
>>> from scipy.spatial.distance import euclidean
>>> from yatel.cluster import kmeans

>>> cbs, distortion = kmeans.kmeans(nw, nw.environments(), 2)

>>> for env in nw.environments():
...     coords = kmeans.hap_in_env_coords(nw, env)
...     min_euc = None
...     closest_centroid = None
...     for cb in cbs:
...         euc = euclidean(cb, coords)
...         if min_euc is None or euc < min_euc:
...             min_euc = euc
...             closest_centroid = cb
...     print "{} || {} || {}".format(dict(env), closest_centroid, euc)
{u'lang': u'sp', u'timezone': u'utc-3', u'name': u'Andalucia'} || [0 0 0] || 1.0
{u'lang': u'sp', u'timezone': u'utc-3', u'name': u'Andalucia'} || [0 0 0] || 1.41421356237
{u'lang': u'sp', u'timezone': None, u'name': None} || [0 0 0] || 1.0
{u'lang': u'sp', u'timezone': None, u'name': None} || [0 1 0] || 0.0
{u'lang': None, u'timezone': u'utc-6', u'name': None} || [0 0 0] || 1.0
{u'lang': None, u'timezone': u'utc-6', u'name': None} || [0 1 0] || 0.0
{u'lang': u'sp', u'timezone': u'utc', u'name': u'Andalucia'} || [0 0 0] || 1.0
{u'lang': u'sp', u'timezone': u'utc', u'name': u'Andalucia'} || [0 0 0] || 1.41421356237
```

6.3.5 References

CHAPTER 7

Yatel Objects

Yatel has a series of classes in the file `yatel.dom` that serve to abstract the information into data structures that we conceptually use in **nw-olap**

All classes defined there having behavior like immutable dictionaries and most notably of all is probably `dom.Haplotype`. So assuming two objects of either class, the following statements are equivalent `obj.attribute` o `obj["attribute"]`.

Like any dictionary, they have available methods like `items()`, `keys()` `values()` and `get(k)`.

All instances of `yatel.dom` for being immutable can be used as keys in a dictionary except that the object contains a non hasheable element. Example:

```
>>> from yatel import dom

>>> hap = dom.Haplotype(1)
>>> data = {hap: 1} # works

>>> hap = dom.Haplotype(2, attr=[1,2,3]) # warning list is unhashable
>>> data = {hap: 1} # works
...
TypeError: unhashable type: 'list'
```

7.1 Classes

- `dom.Haplotype` represents a node in a nw-olap and receives at least one parameter (`hap_id`) others being named optionally. Two haplotypes are equal if they have the same `hap_id` regardless of other attributes. Another important feature is that attributes with values None are not taken into account and removed.

  Example:

```
>>> from yatel import dom
>>> hap0 = dom.Haplotype(1)
>>> hap1 = dom.Haplotype(1, attr="foo", attr2=None)
>>> hap0 == hap1
True

>>> "attr" in hap0
False
>>> "attr" in hap1
True
```

```
>>> hap1.attr2 # invalid because is None
AttributeError: 'Haplotype' object has no attribute 'attr2'

>>> hap1.attr
"foo"
>>> hap1["attr"]
"foo"
>>hap0.get("wat?", "default")
"default"

>>> hash(hap0) == hash(hap1)
True

>>> id(hap0) == id(hap1)
False

>>> hap0 is hap1
False

>>> len(hap0)
1
>>> len(hap1)
2

>>> d = {hap0: "foo"}
>>> d[hap1] # remember same hap_id
"foo"

  >>> hap1 in d and hap0 in d
  True
```

- `dom.Edge` represents an edge in the nw-olap. Receives two parameters in its constructor: The first is the weight of arc (always converted to float) and the second is an iterable with the `hap_id` of the haplotypes associated. Two edges are equal only if they have the same weight and the same haplotypes connected. Only have two attributes that are the same as the constructor: it's weight `edge.weight` and connections in a tuple `edge.haps_id`

```
>>> from yatel import dom
>>> edge0 = dom.Edge(1, [1, 2]) # weight 1 and connect haps 1 and 2
>>> edge1 = dom.Edge(1.0, [1, 3]) # weight 1 and connect haps 1 and 3
>>> edge0 == edge1
False

>>> edge0.haps_id
(1, 2)

>>> d = {edge0: "foo", edge1: "waa"}
>>> d
{<Edge (1.0 (1, 2)) at 0x259a710>: 'foo',
<Edge (1.0 (1, 3)) at 0x259a750>: 'waa'}
```

- `dom.Fact` are the meta data of the analysis of haplotypes. Only their first parameter is required, `hap_id` of the haplotype to which they belong, and all other named parameters are optional. `dom.Fact` are equal only if they belong to the same haplotype and possesses the same attributes with the same values. Another important feature is that attributes with values None are not taken into account and removed.

```
>>> from yatel import dom
>>> fact0 = dom.Fact(0, attr0=1, attr1=None)
```

```
>>> fact1 = dom.Fact(1, attr0=1)

>>> fact0 == fact1
False

>>> fact0 is fact1
False

>>> set([fact0, fact1])
{<Fact (of Haplotype '0') at 0x22e75d0>,
 <Fact (of Haplotype '1') at 0x22e78d0>}
```

CHAPTER 8

Stats

The statistics module is one of the fundamental parts of Yatel. It's designed to support decision making through extraction of measure of positions, variation, skewness and peak analysis of arc weights in a given environment.

The features of this module are divided into 2 distinct groups:

- Transformation Functions: Is responsible for converting an environment of a given network into a *numpy array* to accelerate the calculation of statistics.
- Calculation Functions: Are used for calculating statistical measures on a haplotypes environment.

While all calculation functions use internally the transformation functions, it is often critical to the performance of processing to precalculate in an array with the values of the distances of an environment.

8.1 Transformation Functions

The transformation functions are two:

- `weights2array`: given a `dom.Edges` iterable this function returns a `numpy.ndarray` with all weight values of said arcs.

```
>>> from yatel import dom, db, stats

# Our classic network example
>>> nw = db.YatelNetwork("memory", mode="w")

>>> nw.add_elements([
...     dom.Haplotype(0, name="Cordoba", clima="calor", edad=200, frio=True), # left
...     dom.Haplotype(1, name="Cordoba", poblacion=12), # right
...     dom.Haplotype(2, name="Cordoba"), # bottom

...     dom.Edge(6599, (0, 1)),
...     dom.Edge(8924, (1, 2)),
...     dom.Edge(9871, (2, 0)),

...     dom.Fact(0, name="Andalucia", lang="sp", timezone="utc-3"),
...     dom.Fact(1, lang="sp"),
...     dom.Fact(1, timezone="utc-6"),
...     dom.Fact(2, name="Andalucia", lang="sp", timezone="utc")
... ])
... nw.confirm_changes()

# we extract all edges
```

```
edges = nw.edges()
stats.weights2array(edges)
array([ 6599.,  8924.,  9871.])
```

- `env2weightarray`: This function is responsible for converting a `db.YatelNetwork` instance into an array with all weights of the edges contained; or any of them filtered by environments. Also for reasons of implementations can receive any iterable and turn it into a numpy array.

```
>>> stats.env2weightarray(nw)
array([ 6599.,  8924.,  9871.])

# with an environment
>>> stats.env2weightarray(nw, name="Andalucia")
array([ 9871.])
```

8.2 Calculation Functions

Calculation functions are responsible for efficiently calculating statistics on the variability of a network or a network environment. The full list of functions can be found on the reference module `yatel.stats`

```
# we could calculate for example, the mean (or average) in a network
>>> stats.average(nw)
8464.666666666666667

# or in a environment
>>> stats.average(nw, name="Andalucia")
9871.0
```

For performance reasons is desirable to calculate all weights from an environment before before making many calculations (this can speed up to several hundred times the data analysis)

```
# we get the array with it's values
>>> arr = stats.env2weightarray(nw, lang="sp")

# calculate the deviation
>>> stats.std(arr)
1374.7087772405551286
```

The functions also support python iterables such as lists or tuples

```
>>> stats.average([1, 2, 3])
0.81649658092772603

# this wont return a number
>>> stats.average([])
nan
```

8.3 A More Advanced Example

While Yatel provides for the calculation of common statistics, `stats` module for its architecture facilitates data analysis of more complex environments easily integrating itself with the functionality of SciPy.

For example if we wanted to calculate One-Way ANOVA with two environments of our network.

```
# import the one-way ANOVA
>>> from scipy.stats import f_oneway

# first sample
>>> arr0 = stats.env2weightarray(nw, lang="sp")

# second sample
>>> arr1 = stats.env2weightarray(nw, name="Andalucia")

>>> f, p = f_oneway(arr0, arr1)

# value of F
>>> f
0.5232691541329888

# value of P
>>> p
0.54461284339730176
```

CHAPTER 9

The *Query By JSON* (QBJ) Language

Query By JSON (QBJ) comes up from the need of Yatel to provide an agnostic query language for NW-OLAP (OLAP Multidimensional Networks).

9.1 Features

- Declarative.
- Strong typing.
- Design based on JSON given its wide diffusion in Python (language used to implement Yatel).
- For parsing the data types we use the `yatel.typeconv` module that is also exploited in the export and import features of Yatel.
- Prior to parsing QBJ queries are validated with json-schema to avoid unnecessary calculations.
- Considered a Low-level language.

9.2 Syntax: Query and Query returns

Let's start with an example of the simplest QBJ function, *ping*

The purpose of the function *ping* is simply a response without content indicating that Yatel is listening to our queries.

1. **Simple query**

```
{
    "id": "123",
    "function": {
        "name": "ping",
        "args": [],
        "kwargs": {}
    }
}
```

   - `id` is a query identifier. It can be an integer or a string or null. This value will be returned in the query response. If you are processing it asynchronously you can use this field to discriminate processing.
   - `function` is the second, and final, mandatory key in the query. It consists of the query itself to be validated and implemented, which has its own various keys.

- `name` the name of the function to be executed, in this case *ping*
- `args` are positional arguments of the function. In this case *ping* does not have any parameter with which all of the key and the value can be avoided.
- `kwargs` are named parameters of the function and being empty can be avoided as well.

Removing the unnecessary parameters the function could be written

```
{
    'id': '123',
    'function': {
        'name': 'ping'
    }
}
```

The answer to this query has the form:

```
{
    'id': '123',
    'error': false,
    'error_msg': '',
    'stack_trace': null,
    'result': {
        'type': 'bool',
        'value': true
    }
}
```

Where:

- `id` is the same id from the query.
- `error` it's a Boolean value, will be false while the query is processed successfully.
- `error_msg` if the `error` value it is true this key will contain a description of the error that occurred.
- `stack_trace` if the `error` value it is true and the query is run in debug mode contains all the sequences of calls when the error happened.
- `result` always **null** if the `error` value true. On the other hand if no error happened `result` has the resulting value of the function (in our *ping*) which is in format `yatel.typeconv` and indicates that the result is of boolean type and its value is true.

In summary our example simply says that no error happened and as a results a Boolean value of true is returned.

2. **A query with errors**

   Suppose the call to a nonexistent function to see a result of a query with errors.

```
{
    "id": 31221220,
    "function": {
        "name": "fail!",
    }
}
```

In QBJ the function **fail!** Does not exist, therefore the result would be if we run it in debug mode the following

```
{
    'id': 31221220,
    'error': true,
    'error_msg': "'fail!'",
    'stack_trace': "Traceback (most recent call last):...",
    'result': null
}
```

Where:

- `id` it is the same from the query.
- `error` it is *true*.
- `error_msg` tells us that we sent something with the value *fail* is the result of the error.
- `stack_trace` contains the entire sequence of calls where the error within Yatel happens (cut for example) .
- `result` returns empty because an error happened during the processing of the query.

3. **Typical Yatel query**

   We will now see an example with a more typical Yatel function domain as query to obtain a haplotype by its `id`.

```
{
    "id": null,
    "function": {
        "name": "haplotype_by_id",
        "args": [
            {
                "type": "literal",
                "value": "01"
            }
        ]
    }
}
```

   In this case the function `haplotype_by_id` receives a parameter with a value of *01* to be the id of the haplotype to look for. The value of `type` is *literal* so that the value will not be changed from its json data type (string in this case) before being sent to the function. If we think of this as a call to a Python function `haplotype_by_id("01")`

```
{
    'id': null,
    'error': false,
    'error_msg': '',
    'stack_trace': null,
    'result': {
        'type': 'Haplotype',
        'value': {
            'hap_id': {'type': 'int', 'value': 1},
            'name': {'type': 'unicode', 'value': 'Amet'},
            'special': {'type': 'bool', 'value': false}
        }
    }
}
```

   The result returns a value of type `Haplotype` whose attributes are: `hap_id` integer of value *1*, `name` unicode of value *Amet* and a Boolean called `special` with value *false*

4. **Query with advanced type handling**

   The following query is a `sum` query that adds two or more values whatever pass.

```
{
    "id": "someid",
    "function": {
        "name": "sum",
        "kwargs": {
            "nw": {
                "type": "list",
                "value": [
                    {"type": "literal", "value": 1},
                    {"type": "int", "value": "2"}
                ]
            }
        }
    }
}
```

   As we see in this query the parameter `nw` is a list containing the values "1" (defined as *literal*, so Yatel takes the json type) and the second *int* with a value represented by a string "2". Yatel with this automatically converts the second element to integer type

   A shorter version of the same query would be:

```
{
    "id": "someid",
    "function": {
        "name": "sum",
        "kwargs": {
            "nw": {"type": "literal", "value": [1, 2]}
        }
    }
}
```

   The result has the form

```
{
    'id': "someid",
    'error': false,
    'error_msg': '',
    'stack_trace': null,
    'result': {'type': 'float', 'value': 3.0}
}
```

5. **Nested queries**

```
{
    "id": 1545454845,
    "function": {
        "name": "haplotype_by_id",
        "args": [
            {
                "type": "unicode",
                "function": {
                    "name": "slice",
                    "kwargs": {
                        "iterable": {"type": "unicode",
                                     "value": "id_01_"},
```

```
                        "f": {"type": "int", "value": "-3"},
                        "t": {"type": "int", "value": "-1"}
                    }
                }
            }
        ]
    }
}
```

This query really shows the QBJ potential. The first thing to note is that the main function, `haplotype_by_id`, as the first argument receives the result of function `slice`. The value of the `type` key into the argument indicates that the result of internal function if it is not a text must be converted to it.

`slice` moreover, what it does is cut the text *id_01_* from its position *-3* to *-1*.

if this were Python code the function would be somethin like

```
haplotype_by_id(
    unicode(slice(iterable="id_01_", f=int("-3"), t=int("-1")))
)
```

or what is the same

```
haplotype_by_id("01")
```

The result of this query would return a `Haplotype` from the database as follows:

```
{
    'id': "someid",
    'error': false,
    'error_msg': '',
    'stack_trace': null,
    'result': {
        'type': 'Haplotype',
        'value': {
            'hap_id': {'type': 'int', 'value': 1},
            'color': {'type': 'unicode', 'value': 'y'},
            'description': {'type': 'unicode', 'value': '...'},
            'height': {'type': 'float', 'value': 92.00891409813752},
            'number': {'type': 'int', 'value': 16}
        }
    }
}
```

9.3 Functions

QBJ includes functions to query data on a network (`haplotypes`, `edges`, `facts`, etc.); text handling (`split`, `strip`, `startswith`, `endswith`, etc.); Arithmetic and basic statistics (`sum`, `average`, `kurtosis`, `std`, etc.); data mining and treatment of local date and time as well as UTC

The complete functions list it's available *here*.

9.4 QBJ Console

Yatel provides a comfortable command line interface to use QBJ. It's opened with the command:

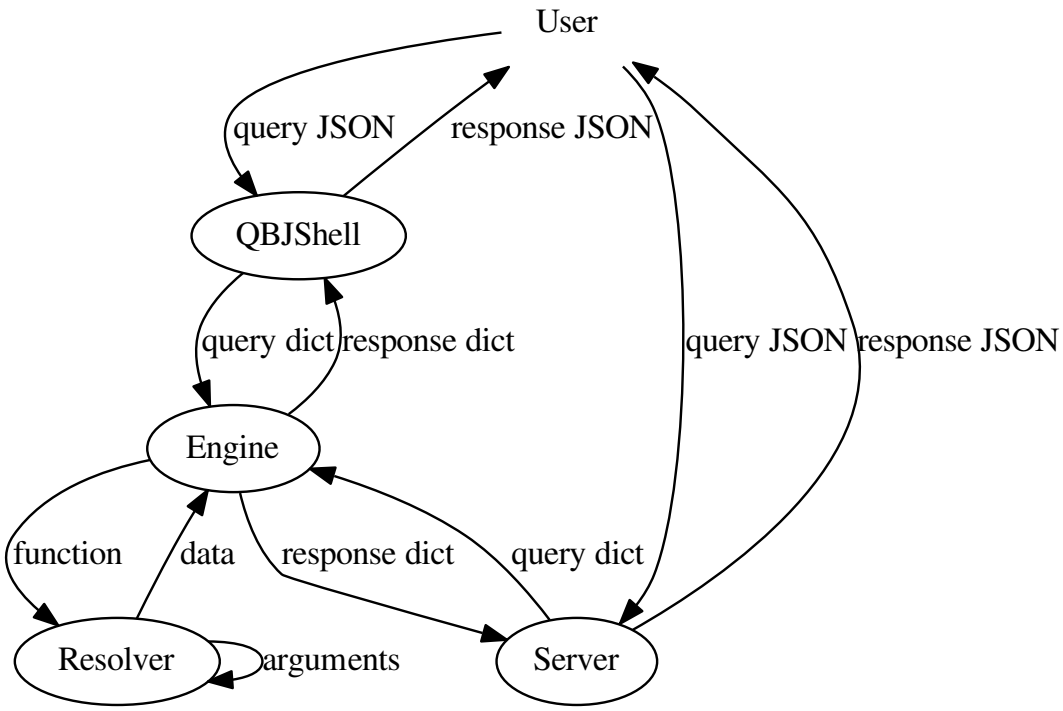
```
$ yatel qbjshell sqlite:///path_to_nw.db
```

See also:

For more information see the documentation about *command line interface*.

9.5 The Resolution Process

Warning: This section provides implementation details useful for developers or people interested in optimizing their queries



1. Yatel both the server and console always receives queries in JSON formatted in UTF-8.
2. The server or console are responsible for converting the `string` into a `dict` hereafter referred as the **query**.
3. The QBJ engine receives the **query** and one more parameter that tells if it should add the stacktrace if anything fails.
4. The engine extracts the main parameters of the query `id` and `function`.
5. The engine validates the **query** against the json-schema of QBJ
6. The engine creates a resolver for the main function and sends it with it's context (the context is the network on wich is executing)

7. The resolver extracts the parameters (*args and **kwargs) of the functions and solves each separated depending on the case by:
   (a) If the argument is a function, it generates a new resolver for that function and passes the context of the current resolver.
   (b) If the argument is just a value it's extracted.
8. Each argument is then casted to the data type specified by itself in the parameter `type` , with the `typeconv` module.
9. The function is executed with all the preprocessed arguments and the result it's returned to the engine.
10. At any step that an error is detected the engine draws it's description for the response and the stacktrace if it was required.
11. The engine simplifies the result with `typeconv` module and creates the response dictionary.
12. Finally the **query** is serialized into JSON and printed on console (if using the QBJ shell) or sent back through the server.

CHAPTER 10

Yatel ETL Framework

One of the main problems faced of data warehouses oriented to analysis is the way in which their data is loaded or updated incrementally.

The technique used is known as ETL, which roughly consists in **Extract** data from source, **Transform** them to make sense in the context of our warehouse, and finally **Load** them to our database.

Yatel provides a modest framework for creating ETL for loading NW-OLAP consistently.

10.1 Creation of a full ETL

The first step in creating a ETL is using Yatel to generate us a template on which to work in a file name, for example,

```
$ yatel createetl myetl.py
```

If we open it we will see the following code

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

'''auto created template to create a custom ETL for yatel'''

from yatel import etl, dom


#===============================================================================
# PUT YOUR ETLs HERE
#===============================================================================

class ETL(etl.BaseETL):

    # you can access the current network from the attribute 'self.nw'
    # You can access all the allready created haplotypes from attribute
    # 'self.haplotypes_cache'. If you want to disable the cache put a class
    # level attribute 'HAPLOTYPES_CACHE = False'


    def haplotype_gen(self):
        raise NotImplementedError()

    def edge_gen(self):
        raise NotImplementedError()

```

```
27      def fact_gen(self):
28          raise NotImplementedError()
29
30
31  #======================================================================================
32  # MAIN
33  #======================================================================================
34
35  if __name__ == "__main__":
36      print(__doc__)
```

Note: As a condition should be clarified that whenever the command line tools the class with the ETL to be called must be called `ETL` (Line 13).

Note: It is good practice to have only one ETL per file, to prevent problems at the time of execution and jeopardize the consistency of your data wharehouse.

- Line 6 are the imports used without exception in all the ETL
- Line 13 creates the ETL class that will contain the logic for extraction, transformation and load of the data

Should be noted that there are many methods that can be overridden (there is a sectino for ahead) but the ones that are mandatory to redefine are the generators: `haplotype_gen`, `edge_gen`, `fact_gen`.

- `haplotype_gen` (line 21) must return an iterable or in the best of cases a generator of haplotypes that you want to load into the database. For example we may decide that the haplotypes are read of a CSV using the csv module of Python:

  ```
  def haplotype_gen(self):
      with open("haplotypes.csv") as fp:
          reader = csv.reader(fp)
          for row in reader:
              hap_id = row[0] # assume that the id is in the first column
              name = row[1] # assume that the column 1 has an attribute name
              yield dom.Haplotype(hap_id, name=name)
  ```

  As is very common to use these haplotypes in the following functions, the ETL is responsible for storing them in a variable named **haplotypes_cache**. This cache is a **dict-like** whose key are `hap_id` and the values of the haplotypes themselves (cache manipulationhas it's own section ahead).

- `edge_gen` (line 24) must return an iterable or in the best of cases a generator of edges that you want to load into the database. It is normal to want to use the haplotypes cache for comparison and give the right weight to each edge. To compare each happlotype with all the rest but itself we can use the function **itertools.combinations** that comes with Python (if someone would want to compare the haplotypes with itself we can use another function **itertools.combinations.with_replacement**). Finally the weight given by the hamming distance between two haplotypes using the **weights** module in Yatel:

  ```
  def edge_gen(self):
      # we combine  haplotypes by two
      for hap0, hap1 in itertools.combinations(self.haplotypes_cache.values(), 2):
          w = weight.weight("hamming", hap0, hap1)
          haps_id = hap0.hap_id, hap1.hap_id
          yield dom.Edge(w, haps_id)
  ```

- `fact_gen` (line 27) must return an iterable or in the best of cases a generator of facts that you want to load into the database. Normally the greater complexity of the ETL is in this function. We can imagine in our case (to add some complexity to this example) that the facts com from a JSON, whose main element is an object and

its keys are equivalent to the attribute **name** of each haplotype; the values in turn are an array which each one must be a **fact** of said haplotype. A simple example would be:

```
{
    "hap_name_0": [
        {"year": 1978, "description": "something..." },
        {"year": 1990},
        {"notes": "some notes", "year": 1986},
        {"year": 2014, "active": false}
    ],
    ...
}
```

So the function to process the data must first determine what the `hap_id` for each haplotype is before creating fact. We could (by a matter of ease) save a *dict* whose value is the *name* of the haplotype (assuming it's unique) and the value of *hap_id*. To not do useless loops we can do it directly in the method `haplotype_gen` with which would be as follows:

```
def haplotype_gen(self):
    self.name_to_hapid = {}
    with open("haplotypes.csv") as fp:
        reader = csv.reader(fp)
        for row in reader:
            hap_id = row[0]
            name = row[1]
            hap = dom.Haplotype(hap_id, name=name)
            self.name_to_hapid[name] = hap_id
            yield hap
```

Now we can easily create the facts using the json module in Python.

```
def fact_gen(self):
    with open("facts.json", "rb") as fp:
        data = json.load(fp)
        for hap_name, facts_data in data.items():
            hap_id = self.name_to_hapid[hap_name]
            for fact_data in facts_data:
                yield dom.Fact(hap_id, **fact_data)
```

Finally having a destination database we can load it with our ETL with the command:

```
$ yatel runetl sqlite:///my_database.db my_etl.py
```

10.1.1 Initializer and cleanup of an ETL

It may be necessary in some cases your ETL needs some resources and it is convenient that they are freed at the finish of the process (a connection to a database for example); or otherwise create global variables to the methods.

For this cases Yatel has two extra methods than can be redefined in your ETL:

- `setup` which is executed before **all** other methods in the ETL. Added to this; also can receive positional parameters (variable parameters and those with default values are not accepted) wich can be given through the command line.
- `teardown` this method is executed at the end of all processing and is the last responsible for leaving the system in a stable estate after freeing all resources of the ETL execution.

In our example, We might want to write the time of start and end of the ETL execution (obtained with the *time* module in Python) into a file given as a parameter. This is really a better place to declare *dict* `name_to_hapid` that will be

used with the haplotypes and facts. the two functions have the form:

```
def setup(self, filename):
    self.fp = open(filename, "w")
    self.name_to_hapid = {}
    self.fp.write(str(time.time()) + "\n")

def teardown(self):
    self.fp.write(str(time.time()) + "\n")
    self.fp.close()
```

Finally to run our ETL we should use the command passing it parameters for the setup

```
$ yatel runetl sqlite:///my_database.db my_etl.py timestamps.log
```

Note: Should be pointed that all the parameters arriving to `setup` do as text and must be converted to the extent necessary.

10.1.2 Intermediate functions to generators

While it is not commonly use, the ETL has six more methods that give more atomic control of the ETL. Each one of them are executed right before and after each generator, they are:

- `pre_haplotype_gen(self)` executed right before *haplotype_gen*.
- `post_haplotype_gen(self)` executed right after *haplotype_gen*.
- `pre_edge_gen(self)` executed right before *edge_gen*.
- `post_edge_gen(self)` executed right after *edge_gen*.
- `pre_fact_gen(self)` executed right before *fact_gen*.
- `post_fact_gen(self)` executed right after *fact_gen*.

10.1.3 Error Handling

In case of encountering an error in the processing of an ETL, a method can be overridden to treat it: `handle_error(exc_type, exc_val, exc_tb)`

The parameters that `handle_error` receives are equivalent to the exit from a context manager where: *exc_type* is the error class (exception) that happened, *exc_val* its the exception itself and *exc_tb* its the error traceback.

Yes, this method Si este mètodo suspends all execution of ETL (even `teardown`)

Note: ETL **ARENT** context managers.

Note: `handle_error` should **NEVER** relaunch the exception that reaches it as parameter. If you want to silence said exception simply return `True` or a true value, otherwise the exception will propagate.

For example if we want to silence the exception only if it is TypeError

```
def handle_error(self, exc_type, exc_val, exc_tb):
    return exc_type == TypeError
```

10.1.4 Haplotypes cache

The last functionality that can be altered in a ETL is the operation of the cache haplotypes, for example if the haplotypes are too many to keep in memory at the same time we could replace the double dictionary (internal cache and the one that links names with its id) by a single cache that contains the data internally neatly.

The ETL use as cache classes that inherit from `collections.MutableMapping`.

```
import collections

class DoubleDictCache(collections.MutableMapping):

    def __init__(self, path):
        self.by_hap_id = {}
        self.name_to_hap_id = {}

    # all this methods have to be redefined in a mutable mapping
    def __delitem__(self, hap_id):
        hap = self.by_hap_id.pop(hap_id)
        self.name_to_hap_id.pop(hap.name)

    def __getitem__(self, hap_id):
        return self.by_hap_id[hap_id]

    def __iter__(self):
        return iter(self.by_hap_id)

    def __len__(self):
        return len(self.by_hap_id)

    def __setitem__(self, hap_id, hap):
        self.by_hap_id[hap_id] = hap
        self.name_to_hap_id[hap.name] = hap_id

    def get_hap_id(self, name):
        return self.name_to_hap_id[name]
```

To use this class level cache of the ETL we need to redefine an attribute called `HAPLOTYPES_CACHE` Para utilizar este cache a nivel de clase del ETL hay que redefinir un atributo que se llama `HAPLOTYPES_CACHE` and have the class value `DoubleDictCache`.

Note: If you want to disable the cache completely, put the value of `HAPLOTYPES_CACHE` as *None*

In our example the code would be:

```
class ETL(etl.BaseETL):

    HAPLOTYPES_CACHE = DoubleDictCache

    ...
```

Note: Note that it may be required depends on the size of your cache that suits you to implement something on a key value database (Riak o Redis), OO (ZODB) or directly Tenga en cuenta que es posible que sea necesario depende el tamaño de su cache que le convenga implementar algo sobre una base de datos llave valor (Riak o Redis), OO (ZODB) or directly a relational database lia a small SQLite

---

10.1.5 Full example

Full example code can be seen here

10.2 Life cycle of a ETL

1. First it verifies that the class inherits from :py:class:`yatel.etl.BaseETL`.
2. Cache class is extracted and if is not found disabled.
3. **If cache class is:**
   (a) `None` no cache is created.
   (b) `!= None` it verifies that is a subclass of `collections.MutableMapping` then an instance is created and asigned to the etl in `haplotypes_cache` variable.
4. The `db.YatelNetwork` instance is assigned to the variable `nw` in the ETL.
5. `setup` method of the ETL is executed passing all arguments.
6. `pre_haplotype_gen` is executed.
7. Iterating over the `dom.Haplotype` that returns `haplotype_gen` and they are added to the database. If something is returned at some point other than a `dom.Haplotype` an `TypeError` is thrown. If there is a cache each `dom.Haplotype` is assigned to the cache putting the key as *hap_id* and for value the *Haplotype*.
8. `post_haplotype_gen` is executed.
9. `pre_edge_gen` is executed.
10. Iterating over the `dom.Edge` that returns `edge_gen` and they are added to the database. If something is returned at some point other than a `dom.Edge` an `TypeError` is thrown.
11. `post_edge_gen` is executed.
12. `pre_fact_gen` is executed.
13. Iterating over the `dom.Fact` that returns `fact_gen` and they are added to the database. If something is returned at some point other than a `dom.Fact` an `TypeError` is thrown.
14. `post_fact_gen` is executed.
15. `teardown` is executed.
16. Returns `True`.

If something Fails

1. `handle_error` is executed passing it the error information. if `handle_error` returns `False` the exception is not stopped.
2. Returns `None`.

Warning: If you are running your ETL directly using the function `etl.execute` changes are not confirmed and It is your responsibility to run `nw.confirm_changes()`.
If on the other hand you are running with the command line the confirmation is run only if `etl.execute` does not fail at any time.

10.3 Running a ETL in a cronjob

It is highly recommended that before running an ETL to always backup the data for that we suggest the following scripts (for windows and posix) that facilitate this task.

Sugested *bash* (posix) script

```
#!/usr/bin/sh
# -*- coding: utf-8 -*-


DATABASE="engine://your_usr:your_pass@host:port/database";
BACKUP_TPL="/path/to/your/backup.xml";
ETL="/path/to/your/etl_file.py";
LOGFILE="/var/yatel/log.txt"

yatel backup $DATABASE $BACKUP_TPL --log --full-stack 2> $LOGFILE;
yatel runetl $DATABASE $ETL --log --full-stack 2> $LOGFILE;
```

Sugested *bat* (Windows) script

```
set BACKUP_TPL=c:\path\to\your\backup.json
set ETL=c:\path\to\your\etl_file.py
set DATABASE=sqlite://to/thing
set LOGFILE=logfile.txt

yatel backup %DATABASE% %BACKUP_TPL% --log --full-stack 2> %LOGFILE%;
yatel runetl %DATABASE% %ETL% --log --full-stack 2> %LOGFILE%;
```

CHAPTER 11

API Reference: `yatel package`

11.1 Subpackages

11.1.1 yatel.cluster package

Submodules

yatel.cluster.kmeans module

The Yatel kmeans algorithm clusters a network's environments, using as dimensions the haplotypes which exists in each environment or arbitrary values computed over them.

For more information about kmeans:

- Scipy Doc
- KMeans in wikipedia

`yatel.cluster.kmeans.`**`hap_in_env_coords`**(*nw*, *env*)

Generates the coordinates for the kmeans algorithm with the existences of haplotypes in the environment.

Parameters **nw** : `yatel.db.YatelNetwork`

env : a collection of dict or `yatel.dom.Enviroment`

Returns **array** : arrays of arrays

The returned coordinates has M elements (M is the number of haplotypes in the network) with same order of `yatel.db.YatelNetwork.haplotypes_ids` function with 2 posible values:

- **0** if the haplotype doesn´t exist in the environment.
- **0** if the haplotype exist in the environment.

`yatel.cluster.kmeans.`**`kmeans`**(*nw*, *envs*, *k_or_guess*, *whiten=False*, *coordc=None*, **args*, ***kwargs*)

Performs k-means on a set of all environments defined by *fact_attrs* of a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network source of environments to classify.

envs : iterable of `yatel.dom.Environments` or dicts

Represents all the environments to be clustered.

k_or_guess : int or ndarray

The number of centroids to generate. A code is assigned to each centroid, which is also the row index of the centroid in the code_book matrix generated.

The initial k centroids are chosen by randomly selecting observations from the observation matrix. Alternatively, passing a k by N array specifies the initial k centroids.

whiten : bool

execute `scipy.cluster.vq.whiten` function over the observation array before executing subjacent *scipy kmeans*.

coordc : None or callable

If *coordc* is `None` generates use *hap_in_env_coords* function. Otherwise `coordc` must be a callable with 2 arguments:

- `nw` network source of environments to classify.
- `env` the environment to calculate the coordinates

and must return an array of coordinates for the given network environment.

args : arguments for scipy kmeans

kwargs : keywords arguments for scipy kmeans

Returns **coodebook** : an array kxn of k centroids

A k by N array of k centroids. The i'th centroid codebook[i] is represented with the code i. The centroids and codes generated represent the lowest distortion seen, not necessarily the globally minimal distortion.

distortion : the value of the distortion

The distortion between the observations passed and the centroids generated.

Examples

```
>>> from yatel import nw
>>> from yatel.cluster import kmeans
>>> nw = db.YatelNetwork('memory', mode=db.MODE_WRITE)
>>> nw.add_elements([dom.Haplotype(1), dom.Haplotype(2), dom.Haplotype(3)])
>>> nw.add_elements([dom.Fact(1, att0=True, att1=4),
...                  dom.Fact(2, att0=False),
...                  dom.Fact(2, att0=True, att2="foo")])
>>> nw.add_elements([dom.Edge(12, 1, 2),
...                  dom.Edge(34, 2, 3),
...                  dom.Edge(1.25, 3, 1)])
>>> nw.confirm_changes()
>>> kmeans.kmeans(nw, nw.enviroments(["att0", "att2"]), 2)
(array([[1, 0, 0],
       [0, 1, 0]]),
 0.0,
 (({u'att0': True, u'att2': None},),
  ({u'att0': False, u'att2': None}, {u'att0': True, u'att2': u'foo'})))

>>> calc = lambda nw, env: [stats.average(nw, env), stats.std(nw, env)]
>>> kmeans.kmeans(nw, ["att0", "att2"], 2, coordc=calc)
(array([[ 23.   ,  11.   ],
```

```
       [  6.625,   5.375]]),
 0.0)
```

yatel.cluster.kmeans.**nw2obs**(*nw*, *envs*, *whiten=False*, *coordc=None*)

Converts any given environments defined by `fact_attrs` of a network to an observation matrix to cluster with subjacent *scipy kmeans*

Parameters **nw** : `yatel.db.YatelNetwork`

Network source of environments to classify.

envs : iterable of `yatel.dom.Enviroment` or dicts

Represent all the environment to be clustered.

whiten : bool

execute *scipy.cluster.vq.whiten* function over the observation array before executing subjacent *scipy kmeans*.

coordc : None or callable

If coordc is `None` generates use *hap_in_env_coords* function. Otherwise *coordc* must be a callable with 2 arguments:

- *nw* network source of environments to classify.
- *env* the environment to calculate the coordinates

and must return an array of coordinates for the given network environment.

Returns **obs** : a vector of envs

Each I'th row of the M by N array is an observation vector of the I'th environment of `envs`.

Examples

```
>>> from yatel import nw
>>> from yatel.cluster import kmeans
>>> nw = db.YatelNetwork('memory', mode=db.MODE_WRITE)
>>> nw.add_elements([dom.Haplotype(1), dom.Haplotype(2), dom.Haplotype(3)])
>>> nw.add_elements([dom.Fact(1, att0=True, att1=4),
...                  dom.Fact(2, att0=False),
...                  dom.Fact(2, att0=True, att2="foo")])
>>> nw.add_elements([dom.Edge(12, 1, 2),
...                  dom.Edge(34, 2, 3),
...                  dom.Edge(1.25, 3, 1)])
>>> nw.confirm_changes()
>>> kmeans.nw2obs(nw, nw.enviroments(["att0", "att2"]))
array([[1, 0, 0],
       [0, 1, 0],
       [0, 1, 0]])
```

Module contents

This package contains utilities for environment clusterization.

11.1.2 yatel.qbj package

Submodules

yatel.qbj.core module

Main logic behind QBJ.

class yatel.qbj.core.**QBJEngine**(*nw*)
Bases: object

Responsible of storing context for QBJ queries, and executes the functions required on it.

Parameters **nw** : yatel.db.YatelNetwork

Network to be used with the query.

execute(*querydict*, *stacktrace=False*)
Takes the query in querydict and executes it after validation of it's structure.

Parameters **querydict** : dict

Dictionary with query in QBJ format.

stacktrace : bool or False

True if you want a stacktrace to be generated.

Returns dict

Result of the query.

class yatel.qbj.core.**QBJResolver**(*function*, *context*)
Bases: object

Resolver of QBJ calls.

Parameters **function** : dict

Keys of function:

- name function to be called.
- args positional arguments for function name.
- kwargs named arguments for function name.

For further detail on functions arguments see yatel.qbj.functions

context : yatel.db.YatelNetwork

Network to execute functions on.

resolve()
Responsible for putting together the call to function with the respective arguments, and return its result.

yatel.qbj.functions module

QBJ functions domain.

yatel.qbj.functions.**amax**(*nw*, *env=None*, ***kwargs*)
Return the maximum in a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`amin`**(*nw*, *env=None*, ***kwargs*)
Return the minimum in a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`average`**(*nw*, *env=None*, ***kwargs*)
Compute the weighted average on a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`capitalize`**(*nw*, *string*)
Return a copy of `string` with its first character capitalized and the rest lowercased.

`yatel.qbj.functions.`**`count`**(*nw*, *iterable*, *to_count*)
Returns the number of occurrences of `to_count` in `iterable`.

`yatel.qbj.functions.`**`describe`**(*nw*)
Returns a `yatel.dom.Descriptor` object with all the information about the network.

The descriptor object is a dictionary like with keys:

edges_attributes [dict] Dictionary contains always 2 keys : `max_nodes` How many nodes connect the edge with maximun number of connections. And `weight` the time of weight attribute

fact_attributes [dict] Contains an arbitrary number of keys, with keys as attributes name, and value as attribute type.

haplotype_atributes [dict] Contains an arbitrary number of keys, with keys as attributes name, and value as attribute type.

mode [str] Actual mode of the network

size [dict] Has the number of elements in the network discrimined by type haplotypes, facts and edges.

Examples

```
>>> nw = db.YatelNetwork(...)
>>> nw.describe()
... {
...     u'edge_attributes': {
...         u'max_nodes': 2,
...         u'weight': <type 'float'>
...     },
...     u'fact_attributes': {
...         u'align': <type 'int'>,
...         u'category': <type 'str'>,
```

```
...         u'coso': <type 'str'>,
...         u'hap_id': <type 'int'>,
...     }
...     u'haplotype_attributes': {
...         u'color': <type 'str'>,
...         u'description': <type 'str'>,
...         u'hap_id': <type 'int'>,
...     }
...     u'mode': 'r',
...     u'size': {u'edges': 10, u'facts': 20, u'haplotypes': 5}
... }
```

yatel.qbj.functions.**div**(*nw*, *dividend*, *divider*)
Return division of `dividend` by `divider`.

yatel.qbj.functions.**edges**(*nw*)
Iterates over all `yatel.dom.Edge` instances stored in the database.

REQUIRE MODE: r

Returns iterator

Iterator of `yatel.dom.Edge` instances.

yatel.qbj.functions.**edges_by_environment**(*nw*, *env=None*, ***kwargs*)
Iterates over all `yatel.dom.Edge` instances of a given environment please see *yatel.db.YatelNetwork.haplotypes_enviroment* for more documentation about environment.

REQUIRE MODE: r

Parameters **env** : dict

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribute.

kwargs : dict

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribte.

Returns iterator

Iterator of `yatel.dom.Edge`.

yatel.qbj.functions.**edges_by_haplotype**(*nw*, *hap*)
Iterates over all the edges of a given `yatel.dom.Haplotype`.

REQUIRE MODE: r

Parameters **hap** : `yatel.dom.Haplotype`

Haplotype to search with.

Returns iterator

Iterator of `yatel.dom.Edge`.

yatel.qbj.functions.**endswith**(*nw*, *string*, *suffix*, *start=None*, *end=None*)
Return True if `string` ends with the specified `suffix`, otherwise return False. `suffix` can also be a tuple of suffixes to look for. With optional `start`, test beginning at that position. With optional `end`, stop comparing at that position.

yatel.qbj.functions.**env2weightarray**(*nw*, *env=None*, ***kwargs*)
This function always return a **numpy.ndarray** with this conditions:

- If `nw` is instance of **numpy.ndarray** the same array is returned.
- If `nw` is instance of `yatel.db.YatelNetwork` and an environment is given return all the edges in this environment.
- If `nw` is instance of `yatel.db.YatelNetwork` and no environment is given then return all edges.
- In the last case the function tries to convert `nw` to **numpy.ndarray** instance.

`yatel.qbj.functions.`**`environments`**(*nw*, *facts_attrs=None*)

Iterates over all combinations of environments of the given attrs.

REQUIRE MODE: r

Parameters **fact_attrs** : iterable

Collection of existing fact attribute names.

Returns iterator

Iterator of dictionaries with all valid combinations of values of a given `fact_attrs` names

Examples

```
>>> for env in nw.enviroments(["native", "place"]):
...     print env
{u'place': None, u'native': True}
{u'place': u'Hogwarts', u'native': False}
{u'place': None, u'native': False}
{u'place': u'Mordor', u'native': True}
{u'place': None, u'native': None}
...
```

`yatel.qbj.functions.`**`facts`**(*nw*)

Iterates over all `yatel.dom.Fact` instances stored in the database.

`yatel.qbj.functions.`**`facts_by_environment`**(*nw*, *env=None*, ***kwargs*)

Iterates over all `yatel.dom.Fact` instances of a given environment please see *yatel.db.YatelNetwork.haplotypes_environment* for more documentation about environment.

REQUIRE MODE: r

Parameters **env** : dict

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribute.

kwargs : dict of keywords arguments

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribute.

Returns iterator

Iteratorterator of `yatel.dom.Fact`.

`yatel.qbj.functions.`**`facts_by_haplotype`**(*nw*, *hap*)

Return a iterator of all facts of a given `yatel.dom.Haplotype`.

Parameters **hap** : `yatel.dom.Haplotype`

Haplotype to search with.

Returns iterator

Iterator of yatel.dom.Fact.

yatel.qbj.functions.**find**(*nw*, *string*, *subs*, *start=None*, *end=None*)
Return the lowest index in string where the substring sub is found such that sub is wholly contained in string[start:end]. Return -1 on failure. Defaults for start and end and interpretation of negative values is the same as for slices.

yatel.qbj.functions.**floor**(*nw*, *dividend*, *divider*)
Return mod from division operation between dividend and divider.

yatel.qbj.functions.**haplotype_by_id**(*nw*, *hap_id*)
Return a dom.Haplotype instace stored in the dabase with the given hap_id.

REQUIRE MODE: r

Parameters **hap_id** : id of the haplotypes type table.

Returns yatel.dom.Haplotype

yatel.dom.Haplotype instance.

yatel.qbj.functions.**haplotypes**(*nw*)
Iterates over all yatel.dom.Haplotype instances stored in the database.

REQUIRE MODE: r

Returns iterator

iterator of yatel.dom.Haplotypes instances.

yatel.qbj.functions.**haplotypes_by_environment**(*nw*, *env=None*, ***kwargs*)
Return an iterator of yatel.dom.Haplotype related to a yatel.dom.Fact with attribute and value specified in env and kwargs.

REQUIRE MODE: r

Parameters **env** : dict

Keys are yatel.dom.Fact attributes name, and value is a possible value of the given attribute.

kwargs : a dict of keywords arguments

Keys are yatel.dom.Fact attributes name, and value is a possible value of the given attribute.

Returns iterator

Iterator of yatel.dom.Haplotype.

Examples

```
>>> from yatel import db, dom
>>> nw = db.YatelNetwork("sqlite", mode=db.MODE_WRITE, database="nw.db")
>>> nw.add_elements([dom.Haplotype("hap1"),
...                  dom.Haplotype("hap2"),
...                  dom.Fact("hap1", a=1, c="foo"),
...                  dom.Fact("hap2", a=1, b=2),
...                  dom.Edge(1, ("hap1", "hap2"))])
>>> nw.confirm_changes()
>>> tuple(nw.haplotypes_enviroment(a=1))
```

```
(<Haplotype 'hap1' at 0x2463250>, <Haplotype 'hap2' at 0x2463390>)
>>> tuple(nw.haplotypes_enviroment({"c": "foo"}))
(<Haplotype 'hap1' at 0x2463250>, )
>>> tuple(nw.haplotypes_enviroment({"a": 1}, b=2))
(<Haplotype 'hap2' at 0x2463390>, )
```

yatel.qbj.functions.**help**(*nw*, *fname=None*)
Returns a list of all functions if `fname` is not specified or `None`, otherwise documentation for `fname`.

yatel.qbj.functions.**index**(*nw*, *iterable*, *value*, *start=None*, *end=None*)
Return the lowest index in `iterable` where `value` is found. Returns `-1` if not found.

Parameters **nw** : `yatel.db.YatelNetwork`

network source of data.

iterable : iterator

iterable object.

value :

Value to look for.

start : int or None

Starting point.

end : int or None

Finishing point.

yatel.qbj.functions.**isalnum**(*nw*, *string*)
Return true if all characters in `string` are alphanumeric and there is at least one character, false otherwise.

yatel.qbj.functions.**isalpha**(*nw*, *string*)
Return true if all characters in `string` are alphabetic and there is at least one character, false otherwise.

yatel.qbj.functions.**isdigit**(*nw*, *string*)
Return true if all characters in `string` are digits and there is at least one character, false otherwise.

yatel.qbj.functions.**islower**(*nw*, *string*)
Return `true` if all cased characters in `string` are lowercase and there is at least one cased character, `false` otherwise.

yatel.qbj.functions.**isspace**(*nw*, *string*)
Return `true` if there are only whitespace characters in `string` and there is at least one character, `false` otherwise.

yatel.qbj.functions.**istitle**(*nw*, *string*)
Return true if `string` is a titlecased string and there is at least one character.

yatel.qbj.functions.**isupper**(*nw*, *string*)
Return `true` if all cased characters in `string` are uppercase and there is at least one cased character, `false` otherwise.

yatel.qbj.functions.**join**(*nw*, *joiner*, *to_join*)
Concatenate a list or tuple of words with intervening occurrences of `joiner`.

yatel.qbj.functions.**kmeans**(*nw*, *envs*, *k_or_guess*, *whiten=False*, *coords=None*, **args*, ***kwargs*)

yatel.qbj.functions.**kurtosis**(*nw*, *env=None*, ***kwargs*)
Computes the kurtosis (Fisher's definition) of a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`lower`**(*nw*, *string*)
Return a copy of `string`, with upper case letters converted to lower case.

`yatel.qbj.functions.`**`lstrip`**(*nw*, *string*, *chars=None*)
Return a copy of `string` with leading characters removed. If `chars` is omitted or `None`, whitespace characters are removed. If given and not `None`, `chars` must be a string; the characters in the string will be stripped from the beginning of the string.

`yatel.qbj.functions.`**`max`**(*nw*, *env=None*, ***kwargs*)
Return the maximum in a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`median`**(*nw*, *env=None*, ***kwargs*)
Compute the median on a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`min`**(*nw*, *env=None*, ***kwargs*)
Return the minimum in a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`minus`**(*nw*, *minuend*, *sust*)
Return subtraction of `sust` from `minuend`.

`yatel.qbj.functions.`**`mode`**(*nw*, *env=None*, ***kwargs*)
Calculates mode on a network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`now`**(*nw*, **args*, ***kwargs*)
Return the current local date and time.

`yatel.qbj.functions.`**`percentile`**(*nw*, *q*, *env=None*, ***kwargs*)
Compute the q-th percentile of the network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`ping`**(*nw*)
Always return True

`yatel.qbj.functions.`**`pow`**(*nw*, *radix*, *exp*)
Return exponentiation of `radix` to `exp`.

`yatel.qbj.functions.`**`range`**(*nw*, *env=None*, ***kwargs*)
Computes the distance between the maximum and minimum.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`replace`**(*nw*, *string*, *old*, *new*, *count=None*)
Return a copy of `string` with all occurrences of `old` replaced by `new`. If `count` is given, the first `count` occurrences are replaced.

`yatel.qbj.functions.`**`rfind`**(*nw*, *string*, *subs*, *start=None*, *end=None*)
Return the highest index in `string` where the substring `sub` is found.

`yatel.qbj.functions.`**`rsplit`**(*nw*, *string*, *s=None*, *maxsplit=None*)
Return a list of the words of `string`, scanning `s` from the end. With `s` as separator and maximum number of split by `maxsplit`.

Parameters **string** : str

String to split.

s : str ot None

Used as separator if given, if `None` uses whitespace characters as separators.

maxsplit : int or None

Maximum number of split on `string` and the remainder of the string is returned as the first element of the list, if `None` no limit.

`yatel.qbj.functions.`**`rstrip`**(*nw*, *string*, *chars=None*)
Return a copy of *string* with trailing characters removed. If `chars` is omitted or `None`, whitespace characters are removed. If given and not `None`, `chars` must be a string; the characters in the string will be stripped from the end of the string.

`yatel.qbj.functions.`**`size`**(*nw*, *iterable*)
Returns size of `iterable`.

`yatel.qbj.functions.`**`slice`**(*nw*, *iterable*, *f*, *t=None*)
Returns `iterable` from F-th element to T-th element.

Parameters **nw** : `yatel.db.YatelNetwork`

network source of data.

iterable : iterator

iterable object.

f : int

Starting point.

t : int or None

Finishing point.

yatel.qbj.functions.**sort**(*nw*, *iterable*, *key=None*, *dkey=None*, *reverse=False*)

Sorts `iterable` using one of it's keys as reference.

Parameters **nw** : `yatel.db.YatelNetwork`

network source of data.

iterable : iterator

iterable object.

key : str or None

A key of objects in `iterable` to sort.

dkey : str or None

Defalut key to sort.

yatel.qbj.functions.**split**(*nw*, *string*, *s=None*, *maxsplit=None*)

Return a list of the words of `string`. With `s` as separator and maximum number of split by `maxsplit`.

Parameters **string** : str

String to split.

s : str ot None

Used as separator if given, if `None` uses whitespace characters as separators.

maxsplit : int or None

Maximum number of split on *string* and the remainder of the string is returned as the final element of the list, if `None` no limit.

yatel.qbj.functions.**startswith**(*nw*, *string*, *prefix*, *start=None*, *end=None*)

Return True if `string` starts with the `prefix`, otherwise return False. `prefix` can also be a tuple of prefixes to look for. With optional `start`, test `string` beginning at that position. With optional `end`, stop comparing string at that position.

yatel.qbj.functions.**std**(*nw*, *env=None*, ***kwargs*)

This function always return a **numpy.ndarray** with this conditions:

- If `nw` is instance of **numpy.ndarray** the same array is returned.
- If `nw` is instance of `yatel.db.YatelNetwork` and an environment is given return all the edges in this environment.
- If `nw` is instance of `yatel.db.YatelNetwork` and no environment is given then return all edges.
- In the last case the function tries to convert `nw` to **numpy.ndarray** instance.

yatel.qbj.functions.**strip**(*nw*, *string*, *chars=None*)

Return a copy of `string` with leading and trailing characters removed. if `chars` is `None` whitespaces are removed otherwise the characters in the string will be stripped from the both ends.

yatel.qbj.functions.**sum**(*nw*, *env=None*, ***kwargs*)

Sum of the elements on the network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`swapcase`**(*nw*, *string*)
Return a copy of `string`, with lower case letters converted to upper case and vice versa.

`yatel.qbj.functions.`**`time`**(*nw*, **args*, ***kwargs*)
Return `time` object of current local date and time..

`yatel.qbj.functions.`**`times`**(*nw*, *t0*, *t1*)
Return multiplication of `t0` by `t1`

`yatel.qbj.functions.`**`title`**(*nw*, *string*)
Returns a copy of `string` in which first characters of all the words are capitalized.

`yatel.qbj.functions.`**`today`**(*nw*, **args*, ***kwargs*)
Return the current local date.

`yatel.qbj.functions.`**`upper`**(*nw*, *string*)
Return a copy of `string`, with lower case letters converted to upper case.

`yatel.qbj.functions.`**`utcnow`**(*nw*, **args*, ***kwargs*)
Return the current UTC date and time.

`yatel.qbj.functions.`**`utctime`**(*nw*, **args*, ***kwargs*)
Return `time` object of current UTC date and time.

`yatel.qbj.functions.`**`utctoday`**(*nw*, **args*, ***kwargs*)
Return `date` object of current UTC date and time.

`yatel.qbj.functions.`**`var`**(*nw*, *env=None*, ***kwargs*)
Compute the variance of the network.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`variation`**(*nw*, *env=None*, ***kwargs*)
Computes the coefficient of variation.

Parameters **nw** : `yatel.db.YatelNetwork`

Network to which apply the operation.

env : `yatel.dom.Enviroment` or dict like

Environment for filtering.

`yatel.qbj.functions.`**`xroot`**(*nw*, *radix*, *root*)
Computes nth root with given `radix` and `root`.

yatel.qbj.schema module

Defines the schema to validate all incomming QBJ.

- http://www.tutorialspoint.com/json/json_schema.htm

`yatel.qbj.schema.`**`DEFINITIONS`** **= {'TYPE_ARRAY_DEF': {'items': {'$ref': '#/definitions/TYPE_DEF'}, 'type': 'array**
Extra definitions for the schema validation.

yatel.qbj.schema.**QBJ_SCHEMA = {'description': 'Defines the schema to validate all incomming QBJ.\n\n- http://www.tut**
Schema to validate QBJ queries.

yatel.qbj.schema.**validate**(*to_validate*, **args*, ***kwargs*)
Validates that the query structure given as JSON is correct.

Parameters **to_validate** : str

String in JSON format.

Raises **ValidationError**

When `to_validate` does not have the corresponding structure.

yatel.qbj.shell module

Interactive shell for QBJ.

class yatel.qbj.shell.**QBJShell**(*nw*, *debug*)
Bases: `cmd.Cmd`

Write your QBJQuery and end it with ';'

Special Commands:

- help: Print this help.
- exec <PATH>: execute a query from a qbj file.
- fhelp: Print a list of all available QBJ-functions
- fhelp <FNAME>: print help about FNAME function.
- indent <INT>: change indentation of response (Default: None).

default(*line*)
Execute a QBJ query.

do_EOF(*args*)
Exit on system end of file character.

do_exec(*fpath*)
Excecute query from .qbj file.

do_exit(*args*)
Exits from the console.

do_fhelp(*args*)
Provides help of the given function if it cant all help is returned.

do_help(*args*)
Provides the entire help.

do_indent(*indent*)
Sets the level of indentation of output.

emptyline()
Do nothing on empty input line.

Module contents

This package contains all necesary function and classes to execute queries using JSON

11.1.3 yatel.weight package

Submodules

yatel.weight.core module

Base structure for weight calculations in Yatel.

class yatel.weight.core.**BaseWeight**
Bases: object

Base class of all weight calculators.

classmethod names()
Abstract Method.

Names of the registered calculators.

Raises **NotImplementedError**

weight(*hap0*, *hap1*)
Abstract Method.

A *float* distance between 2 yatel.dom.Haplotype instances.

weights(*nw*, *to_same=False*, *env=None*, ***kwargs*)
Calculates the distance between all combinations of existing haplotypes of the network environment or a collection.

Parameters **calcname** : string

Registered calculator name (see: *yatel.weight.calculators*)

nw : yatel.db.YatelNetwork or

yatel.dom.Haplotype yatel.db.YatelNetwork instance or iterable of yatel.dom.Haplotype instances

to_same : bool

If True calculate the distance between the same haplotype.

env : dict or None

Enviroment dictionary only if nw is yatel.db.YatelNetwork instance.

kwargs :

Variable parameters to use as enviroment filters only if nw is yatel.db.YatelNetwork instance.

Returns Iterator

Like (hap_x, hap_y), float where hap_x is the origin node, hap_y is the end node and float is the weight between them.

yatel.weight.euclidean module

Euclidean distance implementation of Yatel.

- http://en.wikipedia.org/wiki/Euclidean_distance

class yatel.weight.euclidean.**Euclidean**(*to_num=None*)
Bases: yatel.weight.core.BaseWeight

Calculates "ordinary" distance/weight between two haplotypes given by the Pythagorean formula.

Every attribute value is converted to a number by a to_num function. The default behavior of to_num is a sumatory of base64 ord value of every attribute value. Example:

```
def to_num(attr):
    value = 0
    for c in str(attr).encode("base64"):
        value += ord(c)
    return value

to_num("h") # 294
```

For more info about euclidean distance: http://en.wikipedia.org/wiki/Euclidean_distance

classmethod names()
Synonims names to call this weight calculation.

weight(*hap0*, *hap1*)
A float distance between 2 yatel.dom.Haplotype instances

yatel.weight.euclidean.**to_num_default**(*attr*)
The default behavior of to_num is a sumatory of base64 ord value of every attribute value.

yatel.weight.hamming module

Hamming distance implementation of Yatel.

- http://en.wikipedia.org/wiki/Hamming_distance

class yatel.weight.hamming.**Hamming**
Bases: yatel.weight.core.BaseWeight

Calculate the hamming distance between two haplotypes, by counting the number of differences in attributes.

The distance is incremented by "1" by two reasons:

1.haplotype0.attr_a != haplotype1.attr_a

2.attr_a exist in haplotype0 but not exist in haplotype1.

Examples

```
>>> from yatel import dom, weigth
>>> h0 = dom.Haplotype("0", attr_a="a", attr_b="b", attr_c=0)
>>> h1 = dom.Haplotype("1", attr_a="a", attr_c="0")
>>> hamming = weight.Hamming()
>>> dict(hamming(h0, h1))
{(<haplotype0>, <haplotype1>): 2.0}
```

classmethod names()
Synonims names to call this weight calculation.

weight(*hap0*, *hap1*)
A float distance between 2 dom.Haplotype instances

yatel.weight.levenshtein module

Levenshtein and Damerau Levenshtein distance implementation of Yatel.

- http://en.wikipedia.org/wiki/Levenshtein_distance
- http://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance

class yatel.weight.levenshtein.**DamerauLevenshtein**(*to_seq=None*)
Bases: yatel.weight.levenshtein.Levenshtein

Calculates the Damerau-Levenshtein distance between haplotypes.

This distance is the number of additions, deletions, substitutions, and transpositions needed to transform the first haplotypes as sequences into the second.

Transpositions are exchanges of **consecutive** characters; all other operations are self-explanatory.

This implementation is O(N*M) time and O(M) space, for N and M the lengths of the two sequences.

Note: Previously the haplotypes attribute values are *base64* encoded.

classmethod names()
Synonims names to call this weight calculation.

weight(*hap0*, *hap1*)
A float distance between 2 *dom.Haplotype* instances

class yatel.weight.levenshtein.**Levenshtein**(*to_seq=None*)
Bases: yatel.weight.core.BaseWeight

The Levenshtein distance between two haplotypes is defined as the minimum number of edits needed to transform one haplotype as squence (sumatory of attribute values) into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.

Note: Previously the haplotypes attribute values are encoded with to_seq funcion.

classmethod names()
Synonims names to call this weight calculation.

weight(*hap0*, *hap1*)
A float distance between 2 dom.Haplotype instances

yatel.weight.levenshtein.**to_seq_default**(*obj*)
Converts a given object to a normalized base64 of self.

Module contents

This package contains several modules and functions to calculate distances between haplotypes.

Esentially contains some known algorithms to calculate distances between elements that can be used as edge weights.

yatel.weight.**weight**(*calcname*, *hap0*, *hap1*)
Calculates the weight between yatel.dom.Haplotype instances by the given calculator.

Parameters **calcname** : string

Registered calculator name (see: *yatel.weight.calculators*)

hap0 : yatel.dom.Haplotype

A Haplotype

hap1 : yatel.dom.Haplotype

A Haplotype

Examples

```
>>> from yatel import dom, weight
>>> hap0 = dom.Haplotype(1, att0="foo", att1=34)
>>> hap1 = dom.Haplotype(2, att1=65)
>>> weight.weight("hamming", hap0, hap1)
2
```

yatel.weight.**weights**(*calcname*, *nw*, *to_same=False*, *env=None*, ***kwargs*)

Calculates the distance between all combinations of existing haplotypes in the network enviroment or a collection by the given calculator algorithm.

Parameters **calcname** : string

Registered calculator name (see: *yatel.weight.calculators*)

nw : `yatel.db.YatelNetwork` or `yatel.dom.Haplotype`

`yatel.db.YatelNetwork` instance or iterable of `yatel.dom.Haplotype` instances.

to_same : bool

If `True` calculate the distance between the same haplotype.

env : dict or None

Enviroment dictionary only if `nw` is `yatel.db.YatelNetwork` instance.

kwargs :

Variable parameters to use as enviroment filters only if `nw` is `yatel.db.YatelNetwork` instance.

Returns Iterator

Like `(hap_x, hap_y), float` where `hap_x` is the origin node, `hap_y` is the end node and `float` is the weight between them.

Examples

```
>>> from yatel import db, dom, weight
>>> nw = db.YatelNetwork('memory', mode=db.MODE_WRITE)
>>> nw.add_elements([dom.Haplotype(1, att0="foo", att1=34),
...                  dom.Haplotype(2, att1=65),
...                  dom.Haplotype(3)])
>>> nw.add_elements([dom.Fact(1, att0=True, att1=4),
...                  dom.Fact(2, att0=False),
...                  dom.Fact(2, att0=True, att2="foo")])
>>> nw.add_elements([dom.Edge(12, 1, 2),
...                  dom.Edge(34, 2, 3)])
>>> nw.confirm_changes()

>>> dict(weight.weights("lev", nw))
{(<Haplotype '1' at 0x2823c10>, <Haplotype '2' at 0x2823c50>): 5,
 (<Haplotype '1' at 0x2823c10>, <Haplotype '3' at 0x2823d50>): 7,
 (<Haplotype '2' at 0x2823c50>, <Haplotype '3' at 0x2823d50>): 4}
```

```
>>> dict(weight.weights("ham", nw, to_same=True, att0=False))
{(<Haplotype '2' at 0x1486c90>, <Haplotype '2' at 0x1486c90>): 0}
```

11.1.4 yatel.yio package

Submodules

yatel.yio.core module

Base structure for Yatel parsers.

class yatel.yio.core.**BaseParser**
Bases: object

Base class for parsers.

dump(*nw*, *fp*, **args*, ***kwargs*)
Abstract Method.

Serializes data from a yatel network to a file.

Raises **NotImplementedError**

dumps(*nw*, **args*, ***kwargs*)
Serializes a yatel db to a formatted string.

Parameters **nw** : yatel.db.YatelNetwork

Network source of data.

Returns **string** : str

Json formatted string.

classmethod **file_exts**()

load(*nw*, *fp*, **args*, ***kwargs*)
Abstract Method.

Deserializes data from a file and adds it to the yatel network.

Raises **NotImplementedError**

loads(*nw*, *string*, **args*, ***kwargs*)
Deserializes a formatted string to add it into the yatel database.

Parameters **nw** : yatel.db.YatelNetwork

Network destination for data.

string : str

String to be deserialize.

classmethod **version**()
Returns version of parser.

yatel.yio.core.**YF_STR_VERSION = '0.5'**
Parser version number (string).

yatel.yio.core.**YF_VERSION = ('0', '5')**
Parser version number (tuple).

yatel.yio.yjf module

Persists Yatel databases in json format.

class yatel.yio.yjf.**JSONParser**
Bases: yatel.yio.core.BaseParser

JSON parser to serialize and deserialize data.

dump(*nw*, *fp*, **args*, ***kwargs*)
Serializes data from a Yatel network to a JSON file-like stream.

Parameters **nw** : yatel.db.YatelNetwork

Network source of data.

fp : file-like object

Target for serialization.

classmethod file_exts()
Returns extensions used for JSON handling.

load(*nw*, *fp*, **args*, ***kwargs*)
Deserializes data from a JSON file-like stream and adds it to the yatel network.

Parameters **nw** : yatel.db.YatelNetwork

Network target of data.

fp : file-like object

Source of data to deserialize.

yatel.yio.yxf module

Persists Yatel databases in XML format.

class yatel.yio.yxf.**XMLParser**
Bases: yatel.yio.core.BaseParser

XML parser to serialize and deserialize data.

dump(*nw*, *fp*, **args*, ***kwargs*)
Serializes data from a Yatel network to a XML file-like stream.

Parameters **nw** : yatel.db.YatelNetwork

Network source of data.

fp : file-like object

Target for serialization.

classmethod file_exts()
Returns extensions used for XML handling.

load(*nw*, *fp*, **args*, ***kwargs*)
Deserializes data from a XML file-like stream and adds it to the Yatel network.

Parameters **nw** : yatel.db.YatelNetwork

Network target of data.

fp : file-like object

Source of data to deserialize.

Module contents

Utilities to persist yatel into diferent file formats.

yatel.yio.**PARSERS = {'xml': <class 'yatel.yio.yxf.XMLParser'>, 'json': <class 'yatel.yio.yjf.JSONParser'>, 'yxf': <class 'y**
Container of the different parsers supported by Yatel.

yatel.yio.**SYNONYMS = frozenset([('json', 'yjf'), ('xml', 'yxf')])**
Synonyms of the names used by the parser.

yatel.yio.**dump**(*ext*, *nw*, *stream=None*, **args*, ***kwargs*)
Serializes from a Yatel network to a file or string.

Parameters **ext** : str

Extension of target data.

nw : yatel.db.YatelNetwork

Source database.

stream : file or str

Target of data, can be string or a file.

yatel.yio.**load**(*ext*, *nw*, *stream*, **args*, ***kwargs*)
Deserializes from a stream to Yatel network.

Parameters **ext** : str

Extension of source data.

nw : yatel.db.YatelNetwork

Target database.

stream : file or str

Source of data, can be string or file.

11.2 Submodules

11.3 yatel.cli module

Launcher of Yatel Command Line Interface (cli) tools.

class yatel.cli.**Backup**(**args*, ***kwargs*)
Bases: Command

Like dump but always creates a new file with the format backup_file<TIMESTAMP>.EXT.

class yatel.cli.**Copy**(**args*, ***kwargs*)
Bases: Command

Copy a Yatel network to another database.

class yatel.cli.**CreateConf**(**args*, ***kwargs*)
Bases: Command

Creates a new configuration file for Yatel.

class yatel.cli.**CreateETL**(**args*, ***kwargs*)
Bases: Command

Creates a template file to write your own ETL.

class yatel.cli.**CreateWSGI**(**args*, ***kwargs*)
Bases: Command

Creates a new WSGI file for a given configuration.

class yatel.cli.**Database**(*mode*)
Bases: object

This class parses and validates the open mode of a database.

class yatel.cli.**Describe**(**args*, ***kwargs*)
Bases: Command

Prints information about the network.

class yatel.cli.**DescribeETL**(**args*, ***kwargs*)
Bases: Command

Return a list of parameters and documentation about the ETL. The argument is in the format path/to/module.py The BaseETL subclass must be named after ETL.

class yatel.cli.**Dump**(**args*, ***kwargs*)
Bases: Command

Exports the given database to a file. The extension of the file determines the format.

class yatel.cli.**List**(**args*, ***kwargs*)
Bases: Command

Lists all available connection strings in yatel.

class yatel.cli.**Load**(**args*, ***kwargs*)
Bases: Command

Import the given file to the given database.

class yatel.cli.**PyShell**(**args*, ***kwargs*)
Bases: Shell

Run a python shell with a Yatel Network context.

class yatel.cli.**QBJShell**(**args*, ***kwargs*)
Bases: Command

Runs interactive console to execute QBJ queries.

class yatel.cli.**RunETL**(**args*, ***kwargs*)
Bases: Command

Runs one or more ETL inside of a given script. The first argument is in the format path/to/module.py second onwards parameters are of the setup method of the given class.

class yatel.cli.**Runserver**(**args*, ***kwargs*)
Bases: Command

Run Yatel as a development http server with a given config file.

class yatel.cli.**Test**(**args*, ***kwargs*)
Bases: Command

Run all Yatel test suites.

class yatel.cli.**Version**(**args*, ***kwargs*)
Bases: Command

Show Yatel version and exit.

yatel.cli.**command**(*name*)
Clean way to register class based commands.

11.4 yatel.client module

Client library for yatel

class yatel.client.**QBJClient**(*url*, *nwname*)
Bases: object

Handles query execution and parsing its response.

execute(*query*)
Executes the query given in the server.

parse_response(*response*)
Response parser.

exception yatel.client.**QBJClientError**(*qbjresponse*)
Bases: exceptions.Exception

Custom exception for failed executions.

class yatel.client.**QBJResponse**
Bases: tuple

Response structure tuple.

error
Alias for field number 4

error_msg
Alias for field number 5

id
Alias for field number 0

json
Alias for field number 2

response
Alias for field number 1

stack_trace
Alias for field number 6

yatel
Alias for field number 3

11.5 yatel.db module

Database abstraction layer.

yatel.db.**EDGES = 'edges'**
The name of the edges table

yatel.db.**ENGINES = ('sqlite', 'memory', 'mysql', 'postgres')**
Available engines

yatel.db.**ENGINE_URIS = {'sqlite': 'sqlite:///${database}', 'mysql': 'mysql://${user}:${password}@${host}:${port}/${datab**
Connection uris for the existing engines

yatel.db.**ENGINE_VARS = {'sqlite': ['database'], 'mysql': ['user', 'password', 'host', 'port', 'database'], 'postgres': ['user',**
Variables of the uris

yatel.db.**FACTS = 'facts'**
The name of the facts table

yatel.db.**HAPLOTYPES = 'haplotypes'**
The name of the haplotypes table

yatel.db.**MODES = ('r', 'w', 'a')**
The 3 modes to open the databases

yatel.db.**MODE_APPEND = 'a'**
Constant of append mode

yatel.db.**MODE_READ = 'r'**
Constant of read-only mode

yatel.db.**MODE_WRITE = 'w'**
Constant of write mode (Destroy the existing database)

yatel.db.**PYTHON_TYPES = {<class 'Float'>: <function <lambda> at 0x7fa9a96f27d0>, <class 'DateTime'>: <function <lam**
This dictionary maps sqlalchemy Column types to functions, converts the given Column class to python type. To retrieve all suported columns use db.PYTHON_TYPES.keys()

yatel.db.**SQL_ALCHEMY_TYPES = {<type 'datetime.datetime'>: <function <lambda> at 0x7fa9a96f2050>, <type 'datetime.**
This dictionary maps Python types to functions, converts the given type instance to a correct sqlalchemy column type. To retrieve all suported types use db.SQL_ALCHEMY_TYPES.keys()

yatel.db.**TABLES = ('haplotypes', 'facts', 'edges')**
A collection with the 3 table names

class yatel.db.**YatelNetwork**(*engine*, *mode='r'*, *log=None*, ***kwargs*)
Bases: object

Abstraction layer for yatel network databases.

add_element(*elem*)
Add single instance of yatel.dom.Haplotype or yatel.dom.Fact or yatel.dom.Edge. The network must be in **w** or **a** mode.

REQUIRE MODE: wla

Parameters elems : instance of yatel.dom.Haplotype or yatel.dom.Fact or yatel.dom.Edge.

Element to add.

Examples

```
>>> nw = db.YatelNetwork("sqlite", mode="w", log=False, database="nw.db")
>>> nw.add_element(dom.Fact(3, att0="foo"))
```

add_elements(*elems*)
Add multiple instaces of yatel.dom.Haplotype or yatel.dom.Fact or yatel.dom.Edge instance. The network must be in **w** or **a** mode.

Parameters elems : iterable of `yatel.dom.Haplotype` or `yatel.dom.Fact` or `yatel.dom.Edge` instances.

Elements to be added in the network.

Examples

```
>>> nw = db.YatelNetwork("sqlite", mode="w", log=False, database="nw.db")
>>> nw.add_element([dom.Haplotype(3), dom.Fact(3, att0="foo")])
```

confirm_changes()

Creates the subjacent structures to store the elements added and changes to read mode.

Examples

```
>>> from yatel import db, dom
>>> nw = db.YatelNetwork("sqlite", mode="w", log=False, database="nw.db")
>>> nw.add_element(dom.Haplotype(3, att0="foo"))
>>> nw.confirm_changes()
>>> nw.haplotype_by_id(3)
<Haplotype '3' at 0x37a9890>
```

describe()

Returns a `yatel.dom.Descriptor` object with all the information about the network.

The descriptor object is a dictionary like with keys:

edges_attributes [dict] Dictionary contains always 2 keys : `max_nodes` How many nodes connect the edge with maximun number of connections. And `weight` the time of weight attribute

fact_attributes [dict] Contains an arbitrary number of keys, with keys as attributes name, and value as attribute type.

haplotype_atributes [dict] Contains an arbitrary number of keys, with keys as attributes name, and value as attribute type.

mode [str] Actual mode of the network

size [dict] Has the number of elements in the network discrimined by type haplotypes, facts and edges.

Examples

```
>>> nw = db.YatelNetwork(...)
>>> nw.describe()
... {
...     u'edge_attributes': {
...         u'max_nodes': 2,
...         u'weight': <type 'float'>
...     },
...     u'fact_attributes': {
...         u'align': <type 'int'>,
...         u'category': <type 'str'>,
...         u'coso': <type 'str'>,
...         u'hap_id': <type 'int'>,
...     }
...     u'haplotype_attributes': {
```

```
...         u'color': <type 'str'>,
...         u'description': <type 'str'>,
...         u'hap_id': <type 'int'>,
...     }
...     u'mode': 'r',
...     u'size': {u'edges': 10, u'facts': 20, u'haplotypes': 5}
... }
```

edges()
Iterates over all `yatel.dom.Edge` instances stored in the database.

REQUIRE MODE: r

Returns iterator

Iterator of `yatel.dom.Edge` instances.

edges_by_environment(*env=None*, ***kwargs*)
Iterates over all `yatel.dom.Edge` instances of a given environment please see *yatel.db.YatelNetwork.haplotypes_enviroment* for more documentation about environment.

REQUIRE MODE: r

Parameters **env** : dict

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribute.

kwargs : dict

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribte.

Returns iterator

Iterator of `yatel.dom.Edge`.

edges_by_haplotype(*hap*)
Iterates over all the edges of a given `yatel.dom.Haplotype`.

REQUIRE MODE: r

Parameters **hap** : `yatel.dom.Haplotype`

Haplotype to search with.

Returns iterator

Iterator of `yatel.dom.Edge`.

environments(*facts_attrs=None*)
Iterates over all combinations of environments of the given attrs.

REQUIRE MODE: r

Parameters **fact_attrs** : iterable

Collection of existing fact attribute names.

Returns iterator

Iterator of dictionaries with all valid combinations of values of a given `fact_attrs` names

Examples

```
>>> for env in nw.enviroments(["native", "place"]):
...     print env
{u'place': None, u'native': True}
{u'place': u'Hogwarts', u'native': False}
{u'place': None, u'native': False}
{u'place': u'Mordor', u'native': True}
{u'place': None, u'native': None}
...
```

execute(*query*)
Execute a given `query` to the backend.

REQUIRE MODE: r

Parameters **query** : a query for the backend

A valid query for the backend.

facts()
Iterates over all `yatel.dom.Fact` instances stored in the database.

facts_by_environment(*env=None*, ***kwargs*)
Iterates over all `yatel.dom.Fact` instances of a given environment please see *yatel.db.YatelNetwork.haplotypes_environment* for more documentation about environment.

REQUIRE MODE: r

Parameters **env** : dict

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribute.

kwargs : dict of keywords arguments

Keys are `yatel.dom.Fact` attributes name, and value is a possible value of the given attribute.

Returns iterator

Iteratorterator of `yatel.dom.Fact`.

facts_by_haplotype(*hap*)
Return a iterator of all facts of a given `yatel.dom.Haplotype`.

Parameters **hap** : `yatel.dom.Haplotype`

Haplotype to search with.

Returns iterator

Iterator of `yatel.dom.Fact`.

haplotype_by_id(*hap_id*)
Return a `dom.Haplotype` instace stored in the dabase with the given `hap_id`.

REQUIRE MODE: r

Parameters **hap_id** : `id` of the haplotypes type table.

Returns `yatel.dom.Haplotype`

`yatel.dom.Haplotype` instance.

haplotypes()
Iterates over all yatel.dom.Haplotype instances stored in the database.

REQUIRE MODE: r

Returns iterator

iterator of yatel.dom.Haplotypes instances.

haplotypes_by_environment(*env=None*, ***kwargs*)
Return an iterator of yatel.dom.Haplotype related to a yatel.dom.Fact with attribute and value specified in env and kwargs.

REQUIRE MODE: r

Parameters **env** : dict

Keys are yatel.dom.Fact attributes name, and value is a possible value of the given attribute.

kwargs : a dict of keywords arguments

Keys are yatel.dom.Fact attributes name, and value is a possible value of the given attribute.

Returns iterator

Iterator of yatel.dom.Haplotype.

Examples

```
>>> from yatel import db, dom
>>> nw = db.YatelNetwork("sqlite", mode=db.MODE_WRITE, database="nw.db")
>>> nw.add_elements([dom.Haplotype("hap1"),
...                  dom.Haplotype("hap2"),
...                  dom.Fact("hap1", a=1, c="foo"),
...                  dom.Fact("hap2", a=1, b=2),
...                  dom.Edge(1, ("hap1", "hap2"))])
>>> nw.confirm_changes()
>>> tuple(nw.haplotypes_enviroment(a=1))
(<Haplotype 'hap1' at 0x2463250>, <Haplotype 'hap2' at 0x2463390>)
>>> tuple(nw.haplotypes_enviroment({"c": "foo"}))
(<Haplotype 'hap1' at 0x2463250>, )
>>> tuple(nw.haplotypes_enviroment({"a": 1}, b=2))
(<Haplotype 'hap2' at 0x2463390>, )
```

mode
Returns mode of the database.

uri
Returns uri of the database.

validate_read()
Raise a YatelNetworkError if the network is not in read mode.

Raises **YatelNetworkError**

if the network is not in read mode.

exception yatel.db.**YatelNetworkError**
Bases: exceptions.Exception

Error to use when some Yatel logic fails in the database.

yatel.db.**copy**(*from_nw*, *to_nw*)

Copy all the network in `from_nw` to the network `to_nw`.

`from_nw` must be in read-only mode and `to_nw` in write or append mode. Is your responsability to call `to_nw.confirm_changes()` after the copy.

Parameters **from_nw** : `yatel.db.YatelNetwork`

Network in **r** mode.

to_nw : `yatel.db.YatelNetwork`

Network in **w** or **a** mode.

Examples

```
>>> from yatel import db, dom
>>> from_nw = db.YatelNetwork("memory", mode=db.MODE_WRITE)
>>> from_nw.add_elements([dom.Haplotype("hap1"),
...                       dom.Haplotype("hap2"),
...                       dom.Fact("hap1", a=1, c="foo"),
...                       dom.Fact("hap2", a=1, b=2),
...                       dom.Edge(1, ("hap1", "hap2"))])
>>> from_nw.confirm_changes()
>>> to_nw = db.YatelNetwork("sqlite", mode=db.MODE_WRITE, database="nw.db")
>>> db.copy(from_nw, to_nw)
>>> to_nw.confirm_changes()
>>> list(from_nw.haplotypes()) == list(to_nw.haplotypes())
True
```

yatel.db.**exists**(*engine*, ***kwargs*)

Returns `True` if exists a `yatel.db.YatelNetwork` database in that connection.

Parameters **engine** : str

A value of the current engine used (see valid `yatel.db.ENGINES`)

kwargs : a dict of variables for the engine.

Returns **existsdb** : bool

This function return `False` if:

- The database does not exists.
- The hap_id column has diferent types in `haplotypes`, `facts` or `edges` tables.
- The `edges` table hasn't a column `weight` with type float.

Examples

```
>>> from yatel import db, dom
>>> db.exists("sqlite", mode="r", database="nw.db")
False
>>> from_nw = db.YatelNetwork("memory", mode=db.MODE_WRITE)
>>> from_nw.add_elements([dom.Haplotype("hap1"),
...                       dom.Haplotype("hap2"),
...                       dom.Fact("hap1", a=1, c="foo"),
...                       dom.Fact("hap2", a=1, b=2),
...                       dom.Edge(1, ("hap1", "hap2"))])
```

```
>>> from_nw.confirm_changes()
>>> db.exists("sqlite", mode="r", database="nw.db")
True
```

yatel.db.**parse_uri**(*uri*, *mode='r'*, *log=None*)

Creates a dictionary to use in creation of a YatelNetwork.

```
parsed = db.parse_uri("mysql://tito:pass@localhost:2525/mydb",
                      mode=db.MODE_READ, log=None)
nw = db.YatelNetwork(**parsed)
```

is equivalent to

::

nw = db.YatelNetwork("mysql", database="mydb", user="tito", password="pass", host="localhost", port=2525, mode=db.MODE_READ, log=None)

yatel.db.**qfilter**(*query*, *flt*)

Filters a Yatel query by a given filter.

Parameters **query** : iterator of Yatel DOM

Data to apply filter on.

flt : Lambda expression

Filter expression.

yatel.db.**to_uri**(*engine*, ***kwargs*)

Create a correct uri for a given engine ignoring all unused parameters.

Parameters **engine: str**

The engine name.

kwargs : dict

Variables for the engine.

Examples

```
>>> from yatel import db
>>> db.to_uri("sqlite", database="nw.db")
'sqlite:///nw.db'
>>> db.to_uri("mysql", database="nw", host="localhost", port=3306,
...           user="root", password="secret")
'mysql://root:secret@localhost:3306/nw'
```

11.6 yatel.dom module

Domain Object Model for Yatel.

class yatel.dom.**Descriptor**(*mode*, *fact_attributes*, *haplotype_attributes*, *edge_attributes*, *size*)

Bases: yatel.dom.YatelDOM

Represents detailed information of a network.

class yatel.dom.**Edge**(*weight*, *haps_id*)
Bases: yatel.dom.YatelDOM

Represents a relation between 2 or more *haplotypes*.

class yatel.dom.**Environment**(***attrs*)
Bases: yatel.dom.YatelDOM

Represents an iterable dictionary of dictionaries with valid combinations of values of the attributes given when the instance is created.

class yatel.dom.**Fact**(*hap_id*, ***attrs*)
Bases: yatel.dom.YatelDOM

Fact represents a *metadata* of the *haplotype*.

For example if you gather in two places the same *haplotype*, the characteristics of these places correspond to different *facts* of the same *haplotype*.

class yatel.dom.**Haplotype**(*hap_id*, ***attrs*)
Bases: yatel.dom.YatelDOM

Represents an individual class or group with similar characteristics to be analized.

class yatel.dom.**YatelDOM**(***attrs*)
Bases: _abcoll.Mapping

Base class for yatel objects, handling arbitrary keys.

11.7 yatel.etl module

Functionality to create and execute an ETL. ETLs

class yatel.etl.**BaseETL**
Bases: object

Defines the basic structure of an ETL and methods to be implemented.

HAPLOTYPES_CACHE
alias of dict

edge_gen()
Creation of data to edge like style.

fact_gen()
Creation of data to fact like style.

haplotype_gen()
Creation of data to haplotype like style.

yatel.etl.**etlcls_from_module**(*filepath*, *clsname*)
Return a class of a given filepath.

yatel.etl.**execute**(*nw*, *etl*, **args*)
Execute an ETL instance.

yatel.etl.**get_template**()
Return the template of a base ETL as a string.

yatel.etl.**scan_dir**(*dirpath*)
Retrieve all python files from a given directory.

yatel.etl.**scan_file**(*filepath*)
Retrieve all *yatel.etl.BaseETL* subclass of a given file.

11.8 yatel.server module

Http server for querying, using QBJ or YQL (Yatel Query Languaje).

yatel.server.**CONF_SCHEMA = {'type': 'object', 'properties': {'CONFIG': {'additionalProperties': True, 'type': 'object', '**
JSON schema to validate configuration

yatel.server.**WSGI_BASE_TPL = <string.Template object at 0x7fa9a9673f90>**
Template for WSGI configuration

class yatel.server.**YatelHttpServer**(***config*)
Bases: Flask

Yatel server class.

add_nw(*nwname*, *nw*, *enable_qbj*)
Adds the given nw to the server.

nw(*name*)
Returns the context network.

qbj(*nwname*)
It handles the server query calls.

yatel.server.**from_dict**(*data*)
Returns a server created with a dictionary as configuration.

data keys:

- Path to the configuration file.
- IP and port where the service will be listening separated by a :

yatel.server.**get_conf_template**()
Returns a JSON configuration template as a String.

yatel.server.**get_wsgi_template**(*confpath*)
Returns WSGI configuration template as a String.

Parameters **confpath: string**

Path to configuration file.

yatel.server.**validate_conf**(*confdata*)
Validates that the configuration structure given as JSON is correct.

11.9 yatel.stats module

Statistic functions to calculate weight statistics over Yatel enviroments.

yatel.stats.**amax**(*nw*, *env=None*, ***kwargs*)
Return the maximum in a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**amin**(*nw*, *env=None*, ***kwargs*)
Return the minimum in a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**average**(*nw*, *env=None*, ***kwargs*)
Compute the weighted average on a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**env2weightarray**(*nw*, *env=None*, ***kwargs*)
This function always return a **numpy.ndarray** with this conditions:

- If nw is instance of **numpy.ndarray** the same array is returned.
- If nw is instance of yatel.db.YatelNetwork and an environment is given return all the edges in this environment.
- If nw is instance of yatel.db.YatelNetwork and no environment is given then return all edges.
- In the last case the function tries to convert nw to **numpy.ndarray** instance.

yatel.stats.**kurtosis**(*nw*, *env=None*, ***kwargs*)
Computes the kurtosis (Fisher's definition) of a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**max**(*nw*, *env=None*, ***kwargs*)
Return the maximum in a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**median**(*nw*, *env=None*, ***kwargs*)
Compute the median on a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**min**(*nw*, *env=None*, ***kwargs*)
Return the minimum in a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**mode**(*nw*, *env=None*, ***kwargs*)
Calculates mode on a network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**percentile**(*nw*, *q*, *env=None*, ***kwargs*)
Compute the q-th percentile of the network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**range**(*nw*, *env=None*, ***kwargs*)
Computes the distance between the maximum and minimum.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**std**(*nw*, *env=None*, ***kwargs*)
Compute the standard deviation of the network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**sum**(*nw*, *env=None*, ***kwargs*)
Sum of the elements on the network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**var**(*nw*, *env=None*, ***kwargs*)
Compute the variance of the network.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**variation**(*nw*, *env=None*, ***kwargs*)
Computes the coefficient of variation.

Parameters **nw** : yatel.db.YatelNetwork

Network to which apply the operation.

env : yatel.dom.Enviroment or dict like

Environment for filtering.

yatel.stats.**weights2array**(*edges*)
Create a **numpy.ndarray** with all the weights of yatel.dom.Edge

11.10 yatel.typeconv module

Contains functions to convert various support types of Yatel to more easily serializable types.

yatel.typeconv.**HASHED_TYPES = (<type 'dict'>, <class 'yatel.dom.Haplotype'>, <class 'yatel.dom.Fact'>, <class 'yatel.do**
Dictionary of yatel domain object model.

yatel.typeconv.**LITERAL_TYPE = 'literal'**
Constant to retrieve value as is.

yatel.typeconv.**NAMES_TO_TYPES = {'set': <type 'set'>, 'int': <type 'int'>, 'float': <type 'float'>, 'datetime': <type 'date**
This dictionary maps names to data type.

yatel.typeconv.**TO_PYTHON_TYPES = {<type 'complex'>: <type 'complex'>, <type 'datetime.time'>: <function <lambda>**
This dictionary maps types to python types.

yatel.typeconv.**TO_SIMPLE_TYPES = {<type 'complex'>: <function <lambda> at 0x7fa9a969a848>, <type 'datetime.time**
This dictionary maps types to it's most simple representation.

yatel.typeconv.**TYPES_TO_NAMES = {<type 'datetime.time'>: 'time', <type 'list'>: 'list', <type 'long'>: 'long', <type 'con**
This dictionary maps data types to their name.

yatel.typeconv.**np2py**(*obj*)
Converts a numpy number to it´s closest respresentation of Python traditional objects.

yatel.typeconv.**parse**(*obj*)
Parses an objects type and value, according to the dictionary maps.

yatel.typeconv.**simplifier**(*obj*)
Translates obj given to a Python dictionary.

Returns dictionary: dict

a dictionary representation of obj.

11.11 Module contents

Yatel allows the creation of user-profile-distance-based of OLAP Network and their multidimensional analysis through a process of exploration.

In the process of analyzing data from heterogeneous sources - like data regarding biology, social studies, marketing, etc. -, it is often possible to identify individuals or classes (groups of individuals that share some characteristic). This individuals or groups are identified by attributes that were measured and stored in the data data base. For instance, in a biological analysis, the profile can be defined by some certain properties of the nucleic acid, in a social analysis by the data from people and in a sales analysis by the data from sales point tickets.

CHAPTER 12

Indices and tables

- *genindex*
- *modindex*
- *search*

y