KINKi Documentation

Release 0.1

Jacopo Grazzini

November 14, 2018

Contents

1 Overview **1**
1.1 About . . . 1
1.2 Documentation . . . 1

2 KINKi components **3**
2.1 `metadata` Module . . . 3
2.2 Data sketching and status modelling . . . 4
2.3 Online data access and fetching . . . 77
2.4 In-house data access and storage . . . 135
2.5 Database and API client . . . 136
2.6 Data analysis and visualisation . . . 147
2.7 Miscellaneous utilities and third-party functionalities . . . 156
2.8 Indices and tables . . . 194

Python Module Index **195**

CHAPTER 1

Overview

1.1 About

KINKi Is Not Kowalski - Tools for social sensing and heterogeneous data collection, exploration and analysis

version: 0.1

since: 2014

license: .. literalinclude:: ../LICENSE

author: Jacopo Grazzini, European Union

1.2 Documentation

The background and rationale for KINKi implementation and development are developed in a three-parts document entitled *"Collaborative research software for crowd-sourced data exploration: from context to practice"*.

- document on *KINKi* vision: the Big picture is provided in the `Guidelines for scientific evidence provision for policy support based on Big Data and open technologies` (doi: http://dx.doi.org/10.2788/329540);
- draft document on *KINKi* ecosystem: `Open-source implementation for crowd-sourced data collection, analysis and visualisation`;
- draft document on *KINKi* applications: `Showcases and best practices for analysis of data from social online platforms`.

Online API documentation on *readthedocs* (this site): http://kinki-.readthedocs.io/en/latest/.

CHAPTER 2

KINKi components

2.1 metadata Module

KINKi project metadata.

About

credits: grazzja

since: Mon Aug 25 10:31:17 2014

version: 0.1

This file was automatically generated using the metadata generator *program distributed with KINKi project – See original* metadata.json *file.*

Description

Information describing the project: *project* name, *url* link, *package* name, *date* of implementation, *author* name and *contact* email address, main contributors (*credits*) identifiers, *license* and *copyright* of the software, current *version* as well as a short *desciption*.

```
>>> from kinki.metadata import metadata
>>> print metadata
...    project      : KINKi
...    description  : Tools for social sensing and heterogeneous data␣
↪exploration
...    url          : https://github.com/gjacopo/KINKi
...    package      : kinki
...    author       : Jacopo Grazzini
...    contact      : jacopo.grazzini@ec.europa.eu
...    license      : European Union Public Licence (EUPL)
...    copyright    : European Union
...    organization : European Commission (EC - DG ESTAT)
...    version      : 0.1
```

In addition, it makes it possible to call indifferently metadata fields like attributes or keys:

```
>>> metadata['version'] == metadata.__version__
True
>>> metadata.__license__ == metadata.license
True
>>> metadata['description'] == metadata.description
True
```

Note

This file will also be used to generate automatically the corresponding variables further used in the documentation.

From any submodule of the KINKi project, it is possible to load the metadata, e.g. in `kinki.sketch.location` it is done the following way:

```
from kinki.metadata import metadata

metadata = metadata.copy()
metadata.update({
                'subpackage': 'sketch',
                'date': 'Fri Apr 04 14:40:34 2014',
                'credits':  ['grazzja']
                })
```

hence it will accompagny any subpackage's metadata:

```
>>> from kinki.sketch.location import metadata
>>> print metadata
...    project      : KINKi
...    description  : Tools for social sensing and heterogeneous data␣
↪exploration
...    url          : https://github.com/gjacopo/KINKi
...    package      : kinki
...    subpackage   : sketch
...    author       : Jacopo Grazzini
...    contact      : jacopo.grazzini@ec.europa.eu
...    license      : European Union Public Licence (EUPL)
...    copyright    : European Union
...    organization : European Commission (EC - DG ESTAT)
...    credits      : ['grazzja']
...    date         : Fri Apr 04 14:40:34 2014
...    version      : 0.1
```

2.2 Data sketching and status modelling

2.2.1 Purpose

The `sketch` package aims at providing various functionalities for profiling data, *i.e.* conveying a specific *"status"* (model) that characterises the data that will be later collected/analysed/visualised.

A "status" is defined as a set of typical (not necessarly unique) features regarding: a location, a date/time, a quote and/or a specific user. In addition it enables to retrieve... weather conditions!

Main features:

- relies on existing libraries for geolocation and place identification, time conversions and scheduling, text extraction and translation,
- flexible in the definition of the profiles.

2.2.2 Modules

`sketch` Package

Package for sketching/profiling a query through the definition of associated location, timing, quotation...

Usage

```
>>> from kinki.sketch.location import Geographer, Location
>>> from kinki.sketch.timing import Chronometer, Timing
>>> from kinki.sketch.quotation import Interpreter, Quotation
>>> from kinki.sketch.weather import Meteorologist, Weather
>>> from kinki.sketch.user import User
>>> from kinki.sketch.status import Status
```

Contents

`location` Module

Module for geographic place/location identification and setting.

About

credits: grazzja

version: 0.1 – *since*: Fri Apr 04 14:40:34 2014

Description

Perform various types of 'conversions' in order to define equivalently and uniquely geographic locations.

Dependencies

call: `imports`, `yahoo`, `google`, `geoplanetapi`, `yahooGeoPlanet`, `geolocation`

require: `math`, `numpy`, `re`

optional `geopy`, `googlemaps`, `googleplaces`

Contents

class `sketch.location.`**`Geographer`**

Bases: `object`

Sketch Geographer.

Implement various tools for geolocation and location representation. Provide an interface to most of the methods from original geopy, geolocation, geoplanet, googlemaps and googleplaces APIs (those last two may require special credentials) through composition ('has-a' relationship, hence not real 'is-a' inheritance relationship).

As to avoid multiple inheritance, the collection of container attributes *CODER*, *LOCATOR*, *PLANET*, *GPLACES*, *GMAPS* are respectively used to embedd the APIs mentioned above. See detailed implementations and usages in of respective *_geoCoderAPI*, *_geoLocatorAPI*, *_geoPlanetAPI*, *_googleMapsAPI* and *_googlePlacesAPI* classes.

MIN_LATITUDE
dummy min latitude value in degree: -90.

MAX_LATITUDE
ibid for max latitude: 90.

MIN_LONGITUDE
dummy min longitude value in degree: -180.

MAX_LONGITUDE
ibid for max longitude: 180.

EARTH_RADIUS_EQUATOR
Equatorial radius: 6378.1370 km.

EARTH_RADIUS_POLAR
Polar radius: 6356.7523 km.

WGS84_SEMIAXIS_a
major semi-axis of WGS-84 geoidal reference equal to `EARTH_RADIUS_EQUATOR`.

WGS84_SEMIAXIS_b
ibid, minor semi-axis equal to `EARTH_RADIUS_POLAR`.

EARTH_RADIUS_MEAN
mean radius defined by IUGG, set to `(2*WGS84_SEMIAXIS_a + WGS84_SEMIAXIS_b)/3`, equal to 6371.0087 km.

EARTH_RADIUS_AVERAGE
average radius: 6372.7950 km.

DECIMAL_PRECISION
define the precision in location coordinates; default to `5`.

M_TO
dictionary of equivalent distances expressed in `['m', 'km', 'mi', 'ft']` units that are equivalent to **1 m**.

KM_TO,MI_TO,FT_TO
ibid for **1 km**, **1 mi** (mile) and **1 ft** (foot) respectively.

CODER = <sketch.location._geoCoderAPI object>
Geocoder defined as a (static) instance of *_geoCoderAPI* class derived from `geopy.geocoders`.

GMAPS = <sketch.location._googleMapsAPI object>
Geomapper defined as a (static) instance of *_googleMapsAPI* class derived from `googlemaps.GoogleMaps`.

GPLACES = <sketch.location._googlePlacesAPI object>
Geolocator defined as a (static) instance of *_googlePlacesAPI* class derived from `googleplaces.GooglePlaces`.

LOCATOR = <sketch.location._geoLocatorAPI object>
Geolocator defined as a (static) instance of *_geoLocatorAPI* class derived from `geolocation`.

PLANET = <sketch.location._geoPlanetAPI object>
Geocoder defined as a (static) instance of *_geoPlanetAPI* class, itself derived from `geoplanetapi` or `yahooGeoPlanet`.

classmethod bbox2latlon(*bbox*)
Convert an AOI bounding box into the corresponding (lat, Lon, rad) format.

```
>>> lat, Lon, rad = Geographer.bbox2latlon(bbox)
```

Parameters **bbox** (*list*) – a bounding box represented as a 4-lenght list with the (lat,Lon) coordinates of the South-West and North-East corners of the input polygon.

Returns **lat,Lon,rad** (*float*) – parameters defining the CIRCUMcircle of the input bounding box `bbox`.

Note: This method and *`Geographer.latlon2bbox()`* are not idempotent (say it otherwise `Geographer.latlon2bbox(Geographer.bbox2latlon(bbox)` does not return `bbox`): see comments on CIRCUMcircle and INcircle; however, the centre of the bounding box `box` (hence the tuple `(lat,Lon)`) is preserved.

Examples

```
>>> bbox = [2.216707433489782, 48.72843804413901, 2.477292566510218,␣
↪48.98924195586099]
>>> lLr = Geographer.bbox2latlon(bbox)
>>> print lLr
    (2.347, 48.85884, 14.50401801879798)
```

As mentioned, no idempotence, but the centre of the bounding box is still preserved:

```
>>> Geographer.latlon2bbox(*lLr) == bbox
    False
>>> lLr_ = Geographer.bbox2latlon(Geographer.latlon2bbox(*lLr))
>>> print lLr_
    (2.347, 48.85884, 29.007998346748554)
>>> lLr_[:2] == lLr[:2] # are the coordinates of the centre preserved?
    True
```

See also:

`Geographer.latlon2bbox()`

classmethod bbox2polygon(*bbox*, *order='lL'*)

Convert an AOI bounding box into the corresponding polygon of (lat, Lon) or (Lon, lat) coordinates (the latter case is used in GeoJSON format).

```
>>> polygon = Geographer.bbox2polygon(bbox, order='lL')
```

Parameters

- **bbox** (*list*) – a bounding box represented as a 4-lenght list with the (lat,Lon) coordinates (or (Lon,lat); see `order` below) of the South-West and North-East corners of the input polygon.
- **order** (*str*) – a string specifying the order of the coordinates inside the bounding box: it either 'lL' when latitudes come first (hence (lat,Lon), default), or 'Ll' when longitudes come first (hence (Lon,lat)).

Returns **polygon** (*list*) – a 4-lenght list of (lat,Lon) (or (Lon,lat) when `order=='Ll'`) coordinates representing the input bounding box `bbox`.

Example

```
>>> print Geographer.bbox2polygon([2.2241, 48.81554, 2.4699, 48.
↪90214])
    [[2.2241, 48.81554], [2.4699, 48.81554], [2.4699, 48.90214], [2.
↪2241, 48.90214]]
>>> print Geographer.bbox2polygon([2.2241, 48.81554, 2.4699, 48.
↪90214],order='Ll')
    [[48.81554, 2.2241], [48.81554, 2.4699], [48.90214, 2.4699], [48.
↪90214, 2.2241]]
```

See also:

Geographer.polygon2bbox()

classmethod bboxintersection(*bbox1*, *bbox2*)

Retrieve the intersection (None if null) of two AOI bounding boxes.

```
>>> bbox = Geographer.bboxintersection(bbox1, bbox2)
```

Returns **bbox** (*list*) – a bounding box representing the intersection of both `bbox1` and `bbox2` bounding boxes.

classmethod bboxintersects(*bbox1*, *bbox2*)

Determine if two AOI bounding boxes do intersect.

```
>>> resp = Geographer.bboxintersects(bbox1, bbox2)
```

Returns **resp** (*bool*) – `True` if `bbox1` and `bbox2` intersect, `False` otherwise.

classmethod bboxunion(*bbox2*)

Retrieve the union (largest encompassing) of two AOI bounding boxes.

```
>>> bbox = Geographer.bboxunion(bbox1, bbox2)
```

Returns **bbox** (*list*) – a bounding box representing the union of both `bbox1` and `bbox2` bounding boxes.

classmethod bboxwithin(*bbox1*, *bbox2*)

Determine if an AOI bounding box is contained in another one.

```
>>> resp = Geographer.bboxwithin(bbox1, bbox2)
```

Returns **resp** (*bool*) – `True` if `bbox1` is included within `bbox2`, `False` otherwise.

Example

```
>>> bbox = [2.2241, 48.81554, 2.4699, 48.90214]
>>> lLr = (2.347, 48.85884, 14.50401801879798)
>>> assert Geographer.bboxwithin(bbox, Geographer.latlon2bbox(*lLr))
```

classmethod code(**args*, ***kwargs*)

Convert place names to geographic coordinates (default) and reciprocally, depending on the type of input arguments passed.

```
>>> location = Geographer.code(*args, **kwargs)
```

Parameters **args** (*tuple, str*) – a tuple representing (lat,Lon) coordinates, or a string representing either a place name, or again (lat,Lon) coordinates.

Keyword Arguments **reverse** (*bool*) – set to `True` when *location* is passed as a tuple of (lat,Lon) so that a place name is reverse; default: `False`

Returns **location** (*tuple, str*) – a place name or a `(lat,Lon)` tuple.

Raises GeographerError: – when unable to recognize address/location.

Note: It may return no results if it is passed a non-existent address or a lat/lng in a remote location.

Examples

```
>>> print Geographer.code('Paris, France')
    (48.856614, 2.3522219)
>>> paris = Geographer.code('48.85693, 2.3412', reverse=True)
>>> print paris
    [u'76 Quai des Orfèvres, 75001 Paris, France', u"Saint-Germain-l
↪'Auxerrois, Paris, France",
     u'75001 Paris, France', u'1er Arrondissement, Paris, France', u
↪'Paris, France',
     u'Paris, France', u'Île-de-France, France', u'France']
>>> paris == Geographer.code(48.85693, 2.3412, reverse=True)
    True
```

See also:

`reverse()`

classmethod distance(**args, **kwargs*)

Class method used for computing pairwise distances between given locations, passed indifferently as places names or geographic coordinates.

```
>>> D = Geographer.distance(*args, **kwargs)
```

Parameters **args** (*tuple, str*) – a pair of locations represented either as tuple of (lat,Lon) coordinates or string of place name.

Keyword Arguments

- **dist** (*str*) – name of the geo-principle used to estimate the distance: it is any string in ['vincenty', 'great_circle']; see `geopy.distance()` method; default to vincenty.
- **unit** (*str*) – name of the unit used to return the result: any string in ['m', 'ft', 'mi', 'km'] list (obviously representing ['meters', 'feet', 'miles', 'kilometers'] units); default to km.

Returns **D** (`np.array`) – matrix of pairwise distances computed in *unit* unit.

Raises GeographerError: – when wrong unit/code for geodesic distance or when unable to find/recognize locations.

Examples

```
>>> print Geographer.distance((26.062951, -80.238853), (26.060484,-80.
↪207268),
...                            dist='vincenty', unit='m')
```

(continues on next page)

(continued from previous page)

```
    3172.3596179302895
>>> print Geographer.distance((26.062951, -80.238853), (26.060484,-80.
↪207268),
...                            dist='great_circle', unit='km')
    3.167782321855102
>>> print Geographer.distance((26.062951, -80.238853), 'Paris, France
↪',
...                            dist='great_circle', unit='km')
    7338.5353364838438
```

classmethod latlon2bbox(*lat*, *Lon*, *rad*, ***kwargs*)

Convert an AOI in (lat, Lon, rad) format into the corresponding bounding box.

```
>>> bbox = Geographer.latlon2bbox(lat, Lon, rad, **kwargs)
```

Parameters **lat,Lon,rad** (*float*) – see *bbox2latlon()*.

Returns **bbox** (*list*) – bounding box (see *bbox2latlon()*) whose INcircle is the circle defined by the centre `(lat,Lon)` and a radius `rad`.

Example

```
>>> lLr = (2.347, 48.85884, 14.50401801879798)
>>> print Geographer.latlon2bbox(*lLr)
    [2.216707433489782, 48.72843804413901, 2.477292566510218, 48.
↪98924195586099]
```

See also:

Geographer.bbox2latlon()

classmethod polygon2bbox(*polygon*, *order='lL'*)

Convert a polygon of (lat, Lon) or (Lon, lat) coordinates into an AOI bounding box.

```
>>> bbox = Geographer.polygon2bbox(polygon, order='lL')
```

Parameters **polygon,order** – see *bbox2polygon()*.

Returns **bbox** (*list*) – see *bbox2polygon()*.

Example

```
>>> print Geographer.polygon2bbox([[2.2241, 48.81554], [2.4699, 48.
↪81554],
...                                [2.4699, 48.90214], [2.2241, 48.
↪90214]])
    [2.2241, 48.81554, 2.4699, 48.90214])
```

See also:

Geographer.bbox2polygon()

classmethod reverse(**args*)

Convert geographic location (passed as a tuple of coordinates or a string with those coordinates).

```
>>> place = Geographer.reverse(*args)
```

This is nothing else than a (dummy) shortcut to:

```
>>> place = Geographer.code(*args, reverse=True)
```

Parameters **args** (*tuple, str*) – a tuple or a string representing (lat,Lon) coordinates.

Returns **place** (*str*) – a place name.

Examples

```
>>> paris = Geographer.reverse('48.85693, 2.3412')
>>> print paris
    [u'76 Quai des Orfèvres, 75001 Paris, France', u"Saint-Germain-l
↪'Auxerrois, Paris, France",
     u'75001 Paris, France', u'1er Arrondissement, Paris, France', u
↪'Paris, France',
     u'Paris, France', u'Île-de-France, France', u'France']
>>> paris == Geographer.reverse(48.85693, 2.3412)
    True
```

See also:

code()

classmethod round(*coordinates*)

Round coordinates up to an (internal, fixed) precision of 5 digits.

```
>>> coordinates = Geographer.round(coordinates)
```

Examples

```
>>> print location.Geographer.round(2.216707433489782)
    2.21671
>>> print location.Geographer.round([2.216707433489782, 48.
↪72843804413901, 2.477292566510218, 48.98924195586099])
    [2.21671, 48.72844, 2.47729, 48.98924]
```

class sketch.location.**Location**(**kwargs*)

Bases: sketch.base.Base

Class of geographic/location attributes and methods used to define/describe the geospatial status of an object.

The term location in geography is used to identify a point or an area on the Earth's surface or elsewhere.

Initialisation of a *Location* instance:

```
>>> x = Location(**kwargs)
```

_status [dict] a dictionary defining the status of a *Location* instance in a unique way. Its keys are a subset of ['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw',

'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id']. They can be set through keyword arguments with same name.

accuracy
Accuracy attribute (getter/setter) of a *Location* instance.

address
Address attribute (getter/setter) of a *Location* instance.

When not manually set, It is (if possible) derived from the *street*, *locality* and *country* attributes of the instance.

altitude
Altitude attribute (getter/setter) of a *Location* instance.

The returned value is rounded to 5 digits precision (getter). The real internal value is stored in the alt field of the *status* attribute (setter).

ang_unit
Angle unit attribute (getter/setter) of a *Location* instance.

It can be any string in ['rad', 'dps', 'deg']. It is set to deg by default.

bbox
Bounding box attribute (getter/setter) of a *Location* instance.

The returned value is rounded to 5 digits precision (getter). The real internal value is stored in the *bbox* field of the *status* attribute (setter).

clean()
Clean the _status attribute of a *Location* instance: pop all items that have a value in [(), None, [], ''].

```
>>> x.clean()
```

clear()
Clear the status of a *Location* instance, i.e. reset its _status attribute to an empty dictionary {}.

```
>>> x.clear()
```

clearBBox()
Clear the *bbox* bounding box attribute of the current instance.

clearLatLonRad()
Clear the (lat, Lon, rad) attributes of the current instance.

clearPlace()
Clear the (locality,country,woe_id) attributes of the current instance.

clearSWNE()
Clear the (sw,ne) South-West and North-East attributes of the current instance.

coordinates
Coordinates (latitude,longitude) property of a *Location* instance.

copy()
Copy of a *Location* instance.

```
>>> y = x.copy()
```

Returns **y** (*Location*) – a new instance initialised with a copy of its `_status` dictionary attribute, i.e. so that `x.status==y.status`.

country
Country attribute (`getter/setter`) of a *Location* instance.

country_id
Country identifier (`getter/setter`) of a *Location* instance.

dst_unit
Distance unit attribute (`getter/setter`) of a *Location* instance.

It can be any string in `['m', 'ft', 'mi', 'km']`. It is set to km by default.

filter(*keys*)
Filter the status items of a *Location* instance.

```
>>> status = x.filter(*keys)
```

Parameters **keys** (*optional, list of str, str*) – a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id']`; when `()` or `None`, it is set by default to the set `['bbox', 'lat', 'Lon', 'woeid', 'locality', 'country']`.

Returns **status** (*dict*) – a dictionary representing some status whose keys are *keys* and values are those found in the `_status` attribute of the class, say it otherwise `status = dict([(k,x._status.get(k)) for k in keys])`.

geocode(***kwargs*)
Geocode this instance geographic attributes: implements either *Geographer.code()* or *Geographer.reverse()*, depending on the type of output desired.

```
>>> location = x.geocode(**kwargs)
```

get()
Get an object status, i.e. return its `_status` attribute.

```
>>> status = x.get()
```

getBBox(***kwargs*)
Get/retrieve the bounding box *bbox* attribute already stored in the current instance, or estimate it from the parameters passed as keyword arguments.

```
>>> bbox = x.getBBox(**kwargs)
```

Keyword Arguments

- **bbox,bboxLl** (*list, tuple*) – bounding box represented as a 4-lenght `list`/:class‘tuple‘ with either the (lat,Lon) (if `bbox` is set) or (Lon,lat) (if `bboxLl` is set) coordinates of the South-West and North-East corners of the AOI; both parameters are incompatible.
- **lat,Lon,radius** – see *getLatLonRad()*.
- **sw,ne** – see *getSWNE()*.
- **place,woeid,address** – see *getPlace()*.
- **unit** – distance measurement unit; it can be any string in `['m', 'ft', 'mi', 'km']`; default is the *dst_unit* already stored in the geolocation instance; see also *bounding_locations_from()*.

- **ang_unit** – angle measurement unit; it can be any string in `['rad', 'dps', 'deg']`; default is the *ang_unit* already stored in the geolocation instance; see also *bounding_locations_from()*.

Returns **bbox** (*list*) – bounding box of the AOI associated with this geolocation.

getLatLonRad(***kwargs*)

Get/retrieve the `(lat, Lon, rad)` attributes already set in the current instance, or estimate them from other geolocation parameters passed as keyword arguments.

```
>>> lat, Lon, rad = x.getLatLonRad(**kwargs)
```

Keyword Arguments

- **lat,Lon** (*list,tuple*) – (lat,Lon) coordinates of the centre (centroid) of the AOI.
- **radius** (*float*) – radius of the AOI; if not provided while both `lat` and `Lon` are passed, it is set to `0.001 km`.
- **bbox,bboxLl** – see *getBBox()*.
- **sw,ne** – see *getSWNE()*.
- **place,woeid,address** – see *getPlace()*.
- **unit,ang_unit** – see *getBBox()*.

Returns **lat,Lon,rad** (*float*) – variables defining respectively the `(lat,Lon)` coordinates of the centroid and the radius of the AOI associated with this geolocation.

getPlace(***kwargs*)

Get/retrieve a place name from `(locality,country,woe_id)` attributes already stored in the current instance, or estimate it from other geolocation parameters passed as keyword arguments.

```
>>> place, woeid = x.getPlace(**kwargs)
```

Keyword Arguments

- **place** (*str*) – a string defining a location name, e.g. of the form `locatity, country`, for instance `Paris, France`.
- **woeid** (*int*) – a Where On Earth identifier.
- **address** – full address, consistent with place; both options are incompatible.
- **sw,ne** – see *getSWNE()*.
- **lat,Lon,radius** – see *getLatLonRad()*.
- **bbox,bboxLl** – see *getBBox()*.

Returns

- **place** (*str, list of str*) – list of strings or unique string found that match(es) the AOI associated with this geolocation.
- **woeid** (*int*) – corresponding unique identified WOE: the first WOE identifier matching any of the locations in `places` is returned, `None` otherwise.

getSWNE(***kwargs*)

Get/retrieve both SouthWest *sw* and NorthEast *ne* attributes already stored in the current instance, or estimate them from the keywords passed as arguments.

```
>>> sw, ne = x.getSWNE(**kwargs)
```

Keyword Arguments

- **sw,ne** (*list, tuple*) – South-West and North-East (lat,Lon) coordinates (in this order) of the South-West and North-East corners of the AOI represented as a 2-lenght `list/tuple`.
- **bbox,bboxLl** – see *getBBox()*.
- **lat,Lon,radius** – see *getLatLonRad()*.

- **place,woeid,address** – see *getPlace()*.
- **unit,ang_unit** – see *getBBox()*.

Returns **sw,ne** (*tuple*) – tuples of lenght 2, defining the `(lat,Lon)` coordinates of the South-West and North-East corners of the AOI associated with this geolocation.

grain

Grain (granularity) property (`getter/setter`) of a *Location* instance.

id

Property (`getter`) expressing the unique Id of an instance. It is based on the hash represention (SHA1) of the items of the `_status` attribute dictionary that uniquely defines the instance.

items()

Return the *items* of the `_status` attribute (see also `status` property).

Note: As we want to be able to use this method like the standard *items()* one it refers to, e.g. want to be able to call `items()` instead of just `items`, we define it as regular method, not a property.

See also:

keys(), *values()*

keys()

Return the *keys* of the `_status` attribute, e.g. nothing else than a list composed of a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id']`.

See also:

items(), *values()*

latitude

Latitude attribute (`getter/setter`) of a *Location* instance.

The returned value is rounded to 5 digits precision (`getter`). The real internal value is stored in the `lat` field of the *status* attribute (`setter`).

locality

Locality attribute (`getter/setter`) of a *Location* instance.

longitude

Longitude attribute (`getter/setter`) of a *Location* instance.

The returned value is rounded to 5 digits precision (`getter`). The real internal value is stored in the `Lon` field of the *status* attribute (`setter`).

ne

North-East (lat,Lon) coordinates (`getter/setter`) of a *Location* instance.

The returned coordinates are rounded to 5 digits precision (`getter`). The real internal coordinates are stored in the *ne* field of the *status* attribute (`setter`).

place

Place attribute (`getter/setter`) of a *Location* instance.

When not manually set, It is (if possible) derived from the *locality* and *country* attributes of the instance.

postal_id
Postal identifier (`getter/setter`) of a *`Location`* instance.

When not manually set, it is (if possible) derived from (the found ZIP code) of the *`locality`* attribute of the instance.

radius
Radius attribute (`getter/setter`) of a *`Location`* instance.

set(***kwargs*)
Set/update the status of a *`Location`* instance by setting all the items of its `_status` attribute that have not been set already consistently with other items that have been already inserted (e.g. in `__init__()`) or with the items passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments **kwargs** (`dict`) – dictionary of arguments used to set the current status; its keys should be a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id']`.

Note:
The *`set()`* method is essentialy running different settings methods, namely `['setLatLonRad', 'setBBox', 'setSWNE', 'setPlace']`, that are consistent with each other.

- The items in `kwargs` have 'priority' over those already stored in the `_status` attribute of the class, i.e. the `_status` dictionary will be updated with the `kwargs` dictionary.
- The completion of missing items consistent with those provided is ensured through the method `__call__()` which is ran prior to updating the status.
- Another control is performed which consists in checking that at least one priority argument (i.e. an item with key in `['bbox', 'lat', 'Lon', 'woeid', 'locality', 'country']`) has been passed.

---

setBBox(**bbox*, ***kwargs*)
Set/update the bounding box attribute of the current instance consistently with *`bbox`* argument and/or other geolocation parameters passed as keyword arguments.

```
>>> x.setBBox(*bbox, **kwargs)
```

Parameters **bbox** (*`list, tuple`*) – bounding box represented as a 4-lenght `list`/:class'tuple' with the (lat,Lon) coordinates of the South-West and North-East corners of the AOI; whenever it is not `()` or `(None,)` all keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments
- **bbox,bboxLl** – see *`getBBox()`*.
- **lat,Lon,radius** – see *`getLatLonRad()`*.
- **sw,ne** – see *`getSWNE()`*.
- **place,woeid,address** – see *`getPlace()`*.
- **unit,ang_unit** – see *`getBBox()`*.

setLatLonRad(**lLr*, ***kwargs*)
Set/update the `(lat, Lon, rad)` attributes of the current instance consistently with `lLr` arguments and/or other geolocation parameters passed as keyword arguments.

```
>>> x.setLatLonRad(*lLr, **kwargs)
```

Parameters **lLr** (*tuple*) – a tuple of lenght 2 or 3, defining `(lat,Lon)` coordinates and possibly a *radius*; when `len(lLr)==2`, *radius* is set to `0.001 km`; possibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored.

Keyword Arguments

- **lat,Lon,radius** – see *getLatLonRad()*.
- **bbox,bboxLl** – see *getBBox()*.
- **sw,ne** – see *getSWNE()*.
- **place,woeid,address** – see *getPlace()*.
- **unit,ang_unit** – see *getBBox()*.

setPlace(**place, **kwargs*)

Set/update the `(locality,country,woe_id)` attributes of the current instance consistently with *place* argument and/or other geolocation parameters passed as keyword arguments.

```
>>> x.setPlace(*place, **kwargs)
```

Parameters **place** (*tuple of tuple*) – a string defining a location name, e.g. of the form `locatity, country`, for instance `Paris, France`; possibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored.

Keyword Arguments

- **place,woeid,address** – see *getPlace()*.
- **sw,ne** – see *getSWNE()*.
- **lat,Lon,radius** – see *getLatLonRad()*.
- **bbox,bboxLl** – see *getBBox()*.

setSWNE(**swne, **kwargs*)

Set/update the SouthWest *sw* and NorthEast *ne* attributes of the current instance consistently with `swne` argument and/or other geolocation parameters passed as keyword arguments.

```
>>> x.setSWNE(*swne, **kwargs)
```

Parameters **swne** (*tuple of tuple*) – a tuple of tuples of lenght 2, defining `(lat,Lon)` coordinates of the South-West and North-East corners of the AOI; possibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored.

Keyword Arguments

- **sw,ne** (*list, tuple*) – see *getSWNE()*.
- **bbox,bboxLl** – see *getBBox()*.
- **lat,Lon,radius** – see *getLatLonRad()*.
- **place,woeid,address** – see *getPlace()*.
- **unit,ang_unit** – see *getBBox()*.

status

Status property (`getter/setter`) – return the `_status` attribute dictionary.

street

Street attribute (`getter/setter`) of a *Location* instance.

sw

South-West (lat,Lon) coordinates (`getter/setter`) of a *Location* instance.

The returned coordinates are rounded to 5 digits precision (`getter`). The real internal coordinates are stored in the *sw* field of the *status* attribute (`setter`).

topology(*operator*, **arg*, ***kwargs*)

Perform some simple topological operations over the geographic attributes of this instance as provided by its bounding box.

```
>>> location = x.topology(operator, *arg, **kwargs)
```

Note: Implement *Geographer.bboxwithin()*, *Geographer.bboxintersects()*, *Geographer.bboxintersection()* and *Geographer.bboxunion()* depending on the desired operator.

Parameters

- **operator** (*str*) – name of the chosen topological operator; it can be any string in `['within','intersects','intersection','union']`.
- **arg** – if not `()` or `(None,)`, it should be set as a *Location* to operate with.
- **kwargs** (if `arg` is not passed, the arguments passed in `kwargs` will) – be used to defined a location to operate with.

update(*reset=True*, ***kwargs*)

Update an item of the `_status` attribute of a *Location* instance without updating other: may create inconsistent status.

```
>>> x.update(reset=True, **kwargs)
```

Parameters **reset** (*bool*) – boolean variable defining the 'degree' of update; set to `True` when all status items are cleared and replaced by `kwargs` (default), `False` when only those fields present in `kwargs` are updated.

Keyword Arguments **kwargs** (*dict*) – dictionary of keyword arguments used to update the current status; its keys should be a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id']`.

values()

Return the *values* of the `_status` attribute.

See also:

items(), *keys()*

woe_id

WOE (Where on Earth) identifier (`getter`/`setter`) of a *Location* instance.

class `sketch.location.`**_googleMapsAPI**(*client_key=None*)

Class emulating `googlemaps.Client` API.

Inherit in particular the methods geocode and georeverse from the original API.

Most of the `googlemaps.Client` methods will be inherited by the *Geographer* class through composition and embedding in container attributes.

Initialize a geocoder based on `googlemaps.Client`.

```
>>> x = _googleMapsAPI(client_key=None)
```

Parameters **client_key** (*str*) – Google Maps client key.

distance_matrix(**args*, ***kwargs*)

Gets travel distance and time for a matrix of origins and destinations.

Parameters

- **origins** (*a single location, or a list of locations, where a location is a string, dict, list, or tuple*) – One or more locations and/or latitude/longitude values, from which to calculate distance and time. If you pass an address as a string, the service will geocode the string and convert it to a latitude/longitude coordinate to calculate directions.
- **destinations** (*a single location, or a list of locations, where a location is a string, dict, list, or tuple*) – One or more addresses and/or lat/lng values, to which to calculate distance and time. If you pass an address as a string, the service will geocode the string and convert it to a latitude/longitude coordinate to calculate directions.
- **mode** (*string*) – Specifies the mode of transport to use when calculating directions. Valid values are "driving", "walking", "transit" or "bicycling".
- **language** (*string*) – The language in which to return results.
- **avoid** (*string*) – Indicates that the calculated route(s) should avoid the indicated features. Valid values are "tolls", "highways" or "ferries".
- **units** (*string*) – Specifies the unit system to use when displaying results. Valid values are "metric" or "imperial".
- **departure_time** (*int or datetime.datetime*) – Specifies the desired time of departure.
- **arrival_time** (*int or datetime.datetime*) – Specifies the desired time of arrival for transit directions. Note: you can't specify both departure_time and arrival_time.
- **transit_mode** (*string or list of strings*) – Specifies one or more preferred modes of transit. This parameter may only be specified for requests where the mode is transit. Valid values are "bus", "subway", "train", "tram", "rail". "rail" is equivalent to ["train", "tram", "subway"].
- **transit_routing_preference** (*string*) – Specifies preferences for transit requests. Valid values are "less_walking" or "fewer_transfers".
- **traffic_model** – Specifies the predictive travel time model to use. Valid values are "best_guess" or "optimistic" or "pessimistic". The traffic_model parameter may only be specified for requests where the travel mode is driving, and where the request includes a departure_time.
- **region** (*string*) – Specifies the prefered region the geocoder should search first, but it will not restrict the results to only this region. Valid values are a ccTLD code.

Return type matrix of distances. Results are returned in rows, each row containing one origin paired with each destination.

geocode(*arg*, ***kwargs*)

Geocoding is the process of converting addresses (like `"1600 Amphitheatre Parkway, Mountain View, CA"`) into geographic coordinates (like latitude 37.423021 and longitude -122.083739), which you can use to place markers or position the map.

Parameters

- **address** (*string*) – The address to geocode.
- **components** (*dict*) – A component filter for which you wish to obtain a geocode, for example: `{'administrative_area': 'TX', 'country': 'US'}`
- **bounds** (*string or dict with northeast and southwest keys.*) – The bounding box of the viewport within which to bias geocode results more prominently.
- **region** (*string*) – The region code, specified as a ccTLD ("top-level domain")

two-character value.
- **language** – The language in which to return results.

Return type list of geocoding results.

class sketch.location.**_googlePlacesAPI**(*client_key=None*)

Class emulating `googleplaces.GooglePlaces` API.

Inherit in particular the methods nearby_search, text_search and get_place from the original API; instead, method geocode_location is replaced by method geocode.

Most of the `googleplaces.GooglePlaces` methods will be inherited by the *Geographer* class through composition and embedding in container attributes.

Initialize a geocoder based on `googlemaps.GooglePlaces`.

```
>>> x = _googlePlacesAPI(client_key=None)
```

Parameters **client_key** (*str*) – Google Places client key.

classmethod geocode(*sensor=False*, *api_key=None*)

Original Google Places geocoding method.

class sketch.location.**_geoPlanetAPI**(*appid=None*, ***kwargs*)

Class emulating `geoplanet` API and enabling various location definitions, extraction and conversion.

Most of the *_geoPlanetAPI* methods will be inherited by the *Geographer* class through composition and embedding in container attributes. Following, usages are presented using *Geographer*.

Initialize a geocoder based on Yahoo GeoPlanet API.

```
>>> x = _geoPlanetAPI(appid=None, **kwargs)
```

Parameters **appid** (*str*) – Yahoo GeoPlanet application ID.

Keyword Arguments **kwargs** – Parameters accepted by original `geoplanet` API.

locate(*arg*)

Convert a place name (represented by a string) to its WOE (Where On Earth) IDentifier, or the contrary, depending on the type/instance of the input argument.

```
>>> res = x.locate(arg)
```

Parameters **arg** (*str, int*) – string representing a place name or integer representing a WOE identifier.

Returns **res** (*int, str*) – depending on the type/instance of the input *arg* argument, represent the equivalent WOE identifier or place name.

Note: Use methods `get_place_by_woeid()` and `get_woeid_by_place()` from original GeoPlanet API.

---

place2locality(*place*)

Convert a place name (represented by a string) to the locality.

```
>>> locality = Geographer.place2locality(place)
```

Parameters **place** (*str*) – string representing a place name.
Returns **locality** (*str, list of str*) – locality names defined as the string concatenating both city and country names.

place2woeid(*place*)
Convert a WOE (Where On Earth) IDentifier to the corresponding place name (represented by a string).

```
>>> woeid = Geographer.place2woeid(place)
```

Parameters **place** (*str*) – string representing a place name.
Returns **woeid** (*int*) – equivalent WOE identifier.

See also:

locate(), *woeid2place()*.

woeid2bbox(*woeid*)
Convert a WOE (Where On Earth) IDentifier to the generic bounding box (represented as a list of SW and NE coordinates) defined over that WOEID.

```
>>> bbox = Geographer.woeid2bbox(woeid)
```

Parameters **woeid** (*int*) – integer representing a WOE identifier.
Returns **bbox** (*list of float*) – bounding box representing the place.

woeid2centroid(*woeid*)
Convert a WOE (Where On Earth) IDentifier to the centroid of the generic bounding box defined over that WOEID.

```
>>> centroid = Geographer.woeid2centroid(woeid)
```

Parameters **woeid** (*int*) – integer representing a WOE identifier.
Returns **centroid** (*tuple of float*) – barycentre (lat,Lon) of the generic bounding box defined by *woeid*.

woeid2place(*woeid*)
Convert a place name (represented by a string) to its WOE (Where On Earth) IDentifier.

```
>>> place = Geographer.woeid2place(woeid)
```

Parameters **woeid** (*int*) – integer representing a WOE identifier.
Returns **place** (*str*) – equivalent place name.

See also:

locate(), *place2woeid()*.

world
Property returning the geographical entity represented by a WOE (Where On Earth) IDentifier equal to 1.

class sketch.location.**_geoCoderAPI**(***kwargs*)
Class emulating `geopy` API.

Inherit all methods from original API.

Most of the *_geoCoderAPI* methods will be inherited by the *Geographer* class through composition and embedding in container attributes. Following, usages are presented using *Geographer*.

Initialize a geocoder derived from :mod:'geopy'.

```
>>> x = _geoCoderAPI(**kwargs)
```

Keyword Arguments

- **geocoder** (*str*) – geocoder identifier: this can be any accepted geocoder from the list `['Nominatim', 'GeoNames', 'Bing', 'GoogleV3', 'GeocoderDotUS', 'OpenMapQuest', 'YahooPlaceFinder', 'MapQuest', 'ArcGIS', 'LiveAddress', 'GeocodeFarm']`; default value is `'GoogleV3'`.
- **package,module** (*str*) – for some of the accepted geocoders, an API key may be required as well: it will then be retrieved from the module with name `module` in `kinki` package with name `package`: check that it has been correctly set; default values are :data:'media' for `package`, and the (lowercase) name of the geocoder `geocoder` for `module`.

update(*coder*, ***kwargs*)

Call method used for updating the default geocoder.

```
>>> x.update(coder, **kwargs)
```

Keyword Arguments kwargs – keyword arguments accepted by `geopy_` geocoders.

class `sketch.location.`**_geoLocatorAPI**(**args*, ***kwargs*)

Class emulating `GeoLocation` API.

Inherit for instance methods `_check_bounds()` from original API to check for (*lat*, *Long*) coordinates consistency; instead, methods *`distance_to()`* (computation of great circle distance between geolocations) and *`bounding_locations()`* (computation of the bounding coordinates of all points) are overriden.

Most of the *`_geoLocatorAPI`* methods will be inherited by the *`Geographer`* class through composition and embedding in container attributes. Following, usages are presented using *`Geographer`*.

Initialization overriding `geolocation` super initialization method.

```
>>> x = _geoLocatorAPI(*args, **kwargs)
```

classmethod ang_units_to(*from_*, *to*, *ang=0.0*)

Perform simple angular units conversion.

```
>>> u = Geographer.ang_units_to(from, to, ang=0.)
```

Parameters

- **from,to** (*str*) – 'origin' and 'destination' units: any strings in `['rad', 'dps', 'deg']`.
- **ang** (*float*) – angle value to convert; default to 0.

bounding_locations(*distance*, ***kwargs*)

Method overriding super method from `geolocation` API for computing bounding coordinates of all points on the surface of a sphere that have a great circle distance to the point represented by this GeoLocation instance that is less or equal to the distance argument.

```
>>> bbox = x.bounding_locations(distance, **kwargs)
```

Parameters **distance** (*float*) – distance to the location; it must be set in the unit defined by `unit` (see below).

Keyword Arguments

- **unit** (*str*) – distance measurement unit, i.e. distance unit of the input `distance` parameter; it can be any string in `LIST_DIST_UNIT`; default is `km`.
- **'earth_radius'** (*float*) – the radius of the sphere; defaults to '6378.137 km'.

Returns **bbox** (*list*) – a bounding box whose INcircle is the set of all points that have a great circle distance to the point represented by the input GeoLocation that is less or equal to the `distance` argument.

classmethod bounding_locations_from(*loc*, *distance*, ***kwargs*)

Compute bounding coordinates of all points on the surface of a sphere that have a great circle distance to a given point that is less or equal to the distance argument.

```
>>> bbox = Geographer.bounding_locations_from(loc, distance, **kwargs)
```

Parameters

- **loc** (*list, tuple*) – a tuple of lenght 2 defining the `(lat,Lon)` coordinates of a location; it must be set in the unit defined by `ang_unit` (see below).
- **distance** (*float*) – in between-locations distance, see *bounding_locations()*.

Keyword Arguments

- **ang_unit** (*str*) – angle measurement unit, i.e. unit of the input `loc` parameter; it can be any string in `['rad', 'dps', 'deg']`; default is `deg`.
- **unit,'earth_radius'** – see *bounding_locations()*.

Note: Generalise the `bounding_locations()` method.

---

classmethod centroid(**args*)

Retrieve the approximate centroid of a polygon (bounding box). Accuracy is not a major aspect here.

```
>>> lat, Lon = Geographer.centroid(*args)
```

Parameters **args** (*list of list/tuple*) – a list of (lat,Lon) coordinates representing the vertices of a polygon.

Returns **lat, Lon** (*tuple*) – (lat,Lon) coordinates of the centroid point, in the same unit as the parameters in `args`.

Note: Convert the polygon to a rectangle by selecting the points with:

- lowest/highest latitude,
- lowest/highest longitude,

among all (lat,Lon) vertex coordinates passed as arguments, then get the center of this rectangle as the centroid point.

---

classmethod convert_angle_units(*to='deg'*, ***kwargs*)

Convert composed angular units to a single one.

```
>>> u = Geographer.convert_angle_units(to='deg', **kwargs)
```

Parameters **to** (*str*) – desired 'final' unit: any string in `['rad', 'dps', 'deg']`; default to `deg`.

Keyword Arguments **kwargs** (`dict`) – dictionary of composed angles indexed by their unit, which can be, again, any string in `['rad', 'dps', 'deg']`.

classmethod convert_distance_units(*to='km'*, ***kwargs*)

Convert composed distance units to a single one.

```
>>> d = Geographer.convert_distance_units(to='km', **kwargs)
```

Parameters **to** (*str*) – desired 'destination' unit: any string in `['m', 'ft', 'mi', 'km']`; default to `km`.

Keyword Arguments **kwargs** (`dict`) – dictionary of composed distances indexed by their unit, which can be any string in `['m', 'ft', 'mi', 'km']`.

Example

Use indifferently dictionaries or positional arguments to pass the quantities to convert:

```
>>> assert Geographer.convert_distance_units('m', **{'km':1, 'm':10})␣
↪== 1010
>>> print Geographer.convert_distance_units(to='m', mi=2,  ft=10,␣
↪km=0.5)
    3721.7279999999996
>>> print  2*Geographer.MI_TO['m']
...         + 10.*Geographer.FT_TO['m']
...         + 0.5*Geographer.KM_TO['m']
    3721.7279999999996
```

classmethod deg2dps(*degrees*)

Convert degrees format to (degrees, primes, seconds).

```
>>> dps = Geographer.deg2dps(degrees)
```

classmethod deg2rad(*degrees*)

Convert degrees format to radians.

```
>>> radians = Geographer.deg2rad(degrees)
```

distance_to(*other*, ***kwargs*)

Method overriding super method from `geolocation` API for computing the great circle distance between this GeoLocation instance and another (where measurement unit is passed as an argument).

```
>>> R = x.distance_to(other, **kwargs)
```

Parameters **other** (`_geoLocatorAPI`) – a *`_geoLocatorAPI`* instance to which compute a distance to.

Keyword Arguments **unit,'earth_radius'** – see *`bounding_locations()`*.

classmethod distance_to_from(*loc1*, *loc2*, ***kwargs*)

Compute the (approximate) great circle distance between two points on the Earth (specified in decimal degrees). Accuracy is not a major aspect here.

```
>>> lat, Lon = Geographer.distance_to_from(loc1, loc2, **kwargs)
```

Parameters **loc1,loc2** (*tuple, list*) – (lat,Lon) coordinates of the two location points, in the same unit as the parameters in `args`.

Keyword Arguments **ang_unit,unit** – see *bounding_locations_from()*.

Note: Generalise the *distance_to()* method. Inspired by the code in: http://stackoverflow.com/a/4913653/983244

---

classmethod dps2deg(*dps*)

Convert (degrees, primes, seconds) format to degrees.

```
>>> degrees = Geographer.dps2deg(dps)
```

classmethod dps2rad(*dps*)

Convert (degrees, primes, seconds) format to radians.

```
>>> radians = Geographer.dps2rad(dps)
```

classmethod estimate_radius_WGS84(*lat*, ***kwargs*)

Calculate the Earth radius at a given latitude, according to the WGS-84 ellipsoid [m].

```
>>> R = Geographer.estimate_radius_WGS84(lat, **kwargs)
```

classmethod from_degrees(*deg_lat*, *deg_lon*)

Return a geolocation instance from (lat, Lon) coordinates expressed in degrees.

```
>>> x = Geographer.from_degrees(deg_lat, deg_lon)
```

Parameters **deg_lat,deg_lon** (*tuple*) – latitude and longitude (respectively) expressed in degrees.

Returns **x** (`_geoLocatorAPI`) – a *_geoLocatorAPI* instance from expressed (lat, Lon) coordinates.

See also:

from_dps(), *from_radians()*

classmethod from_dps(*dps_lat*, *dps_lon*)

Return a geolocation instance from (lat, Lon) coordinates expressed in DPS format.

```
>>> x = Geographer.from_dps(dps_lat, dps_lon)
```

Parameters **dps_lat,dps_lon** (*tuple*) – latitude and longitude (respectively) expressed in DPS format: (degrees, primes, seconds).

Returns **x** (`_geoLocatorAPI`) – a *_geoLocatorAPI* instance from expressed (lat, Lon) coordinates.

See also:

from_degrees(), *from_radians()*

classmethod from_radians(*rad_lat*, *rad_lon*)

Return a geolocation instance from (lat, Lon) coordinates expressed in degrees.

```
>>> x = Geographer.from_radians(rad_lat, rad_lon)
```

Parameters **rad_lat,rad_lon** (*tuple*) – latitude and longitude (respectively) expressed in radians.

Returns **x** (_geoLocatorAPI) – a *_geoLocatorAPI* instance from expressed (lat, Lon) coordinates.

See also:

from_dps(), *from_degrees()*

classmethod latdeg2m(*dlat*, *alat*)

Convert latitude difference in degrees into meters at given average latitude.

```
>>> dy = Geographer.latdeg2m(dlat, alat)
```

Parameters

- **dlat** (*float*) – latitude difference in degrees.
- **alat** (*float*) – average latitude at which the distance is calculated (between the two fixes).

Returns **dy** (*float*) – latitude difference in meters.

classmethod latm2deg(*dy*, *alat*)

Convert latitude difference in meters into degrees at given average latitude.

```
>>> dlat = Geographer.latm2deg(dy, alat)
```

Parameters

- **dy** (*float*) – latitude difference in meters.
- **alat** (*float*) – average latitude at which the distance is calculated (between the two fixes).

Returns **dlat** (*float*) – latitude difference in degrees.

classmethod lngdeg2m(*dlng*, *alat*)

Convert longitude difference in degrees into meters at given average latitude.

```
>>> dx = Geographer.lngdeg2m(dlng, alat)
```

Parameters

- **dlng** (*float*) – longitude difference in degrees.
- **alat** (*float*) – average latitude at which the distance is calculated (between the two fixes).

Returns **dx** (*float*) – longitude difference in meters.

classmethod lngm2deg(*dx*, *alat*)

Convert longitude difference in degrees into meters at given average latitude.

```
>>> dlng = Geographer.lngdeg2m(dx, alat)
```

Parameters

- **dx** (*float*) – longitude difference in meters.
- **alat** (*float*) – average latitude at which the distance is calculated (between the two fixes).

Returns **dlng** (*float*) – longitude difference in degrees.

classmethod rad2deg(*radians*)

Convert radians format to degrees.

```
>>> degrees = Geographer.rad2deg(radians)
```

classmethod rad2dps(*rad*)
Convert radians format to (degrees, primes, seconds).

```
>>> dps = Geographer.rad2dps(radians)
```

classmethod units_to(*from_*, *to*, *dist=1.0*)
Perform simple distance units conversion.

```
>>> d = Geographer.units_to(from, to, dist)
```

Parameters
- **from,to** (*str*) – 'origin' and 'destination' units: any strings in `['m', 'ft', 'mi', 'km']`.
- **dist** (*float*) – distance value to convert; default to 1.

Example

```
>>> assert Geographer.units_to('mi', 'ft',  10.) == 52800.0
```

`timing` Module

Module for time conversion and timing calculation.

About

credits: grazzja

version: 0.1 – *since*: Fri Apr 04 18:22:13 2014

Description

Perform various date/time format 'conversions' in order to define equivalently and uniquely timestamps.

Dependencies

call: `base`, `imports`, `decorators`

require: `datetime`, `calendar`, `re`

optional `dateutil`, `pyown`, `pytz`

Contents

class `sketch.timing.`**Chronometer**
Bases: `object`

Sketch Chronometer.

Class of methods providing various tools for timing definition and duration calculations.

SEC_TO
dictionary of timing quivalent to **1 sec** when expressed in seconds (`'sec'`), minutes (`'mn'`), hour (`'hr'`), day (`'d'`), month (`'m'`) or year (`'y'`) unit.

Example

```
>>> Chronometer.SEC_TO['mn']==1./60
    True
>>> Chronometer.SEC_TO['hr']==1./3600
    True
>>> print Chronometer.SEC_TO['y']
    3.1688750556929794e-08
```

MN_TO, HR_TO, DAY_TO, MTH_TO, YR_TO
ibid for **1 mn**, **1 hr**, **1 d**, **1 m** and **1 y** respectively.

DATETIME_UNITS = ['y', 'm', 'd', 'hr', 'mn', 'sec', 'day', 'hour', 'minute', 'month
list of timing units accepted for date/time representation.

TIMEDELTA_UNITS = ['y', 'm', 'd', 'hr', 'mn', 'sec', 'days', 'hours', 'minutes', 'm
list of timing units accepted for duration representation.

TIMING_UNITS = ['y', 'm', 'd', 'hr', 'mn', 'sec']
list of short names for timing units.

classmethod convert_time_units(*to='mn', **kwargs*)
Convert composed timing units to a single one.

```
>>> t = Chronometer.convert_time_units(to, **kwargs)
```

Parameters to (*str*) – desired 'final' unit: any string in `['y', 'm', 'd', 'hr', 'mn', 'sec']`; default to `'mn'`.
Keyword Arguments kwargs (`dict`) – dictionary of composed times indexed by their unit, which can be any string in `['y', 'm', 'd', 'hr', 'mn', 'sec']`.

Note: Quantities to convert are passed either as a dictionary or as positional arguments:

Example

```
>>> Chronometer.convert_time_units('mn', **{'hr':1, 'sec':420}) == 67
    True
>>> Chronometer.convert_time_units('mn', hr=1,  sec=420) == 67
    True
```

classmethod datetime(**arg, **kwargs*)
Perform time conversions in between various (not so) standard timing formats: unix timestamp, isoformat, ctime, . . .

```
>>> dt = Chronometer.datetime(*dtime, **kwargs)
```

Parameters dtime (*datetime.datetime, str, float, dict*) – an object specifying a time, e.g. it can be:
- a `datetime.datetime` object: `datetime.datetime(2014, 6, 19, 17, 58, 5)`,
- an iso-formated (ISO 8601) date: `"2014-06-19T17:58:05"` or `"2014-06-19 17:58:05"`,
- a string date: `"Thu Jun 19 17:58:05 2014"`,
- a float representing a unix timestamp date: `1403193485`,

- an explicit string date: `"45d 34hr 2900mn 4m 500sec 2y"` (where order does not matter),
- a dictionary-like date: `{'y':2, 'm':4, 'd':45, 'hr':34, 'mn':2900, 'sec':500}`.

When the date is expressed as an 'explicit string' or a dictionary, a time unit is any string in `['y', 'm', 'd', 'hr', 'mn', 'sec']`.

'now', 'utcnow' and 'today' are also accepted instead of an explicit date.

Keyword Arguments

- **fmt** (*str*) – variable specifying the desired output timing format: it can be any string among:
  - 'datetime' to output a `datetime.datetime` object,
  - 'dict' to output a dictionary indexed by the timing units, where a key is any string in `['y', 'm', 'd', 'hr', 'mn', 'sec']` (see also `datetime.timetuple()` method)),
  - 'timestamp' to output a unix timestamp (see also `calendar.timegm()` method)),
  - 'iso' to output an iso-formated string date (see also `datetime.isoformat()` method),
  - 'calendar' to output a calendar (ISO year, week number, and weekday) date (see also `datetime.isocalendar()` method),

  as well as:
  - a date formating string (e.g. `'%Y-%m-%dT%H:%M:%S'`, see `datetime.strftime()`),
  - `True` to force a string naming (see `time.ctime()` method)).

  When nothing is passed, a default output format is considered, depending on the input format of *dtime* (e.g. a *datetime* object is output for a `dict` *dtime*, ...).
- **no_micro_secs** (*bool*) – set to `False` to display microseconds as part of the output datetime when the chosen output format is the isoformat; default: `True`.

Returns **dt** (*datetime.datetime, str, float, dict, calendar (tuple)*) – a well-formatted (according to *fmt* specification) object representing a datetime equivalent to the input *dtime*.

Note: As for the `dict` format of the input `dtime`, long time unit names (keys in `['days', 'hours', 'minutes', 'months', 'seconds', 'years']`) are also accepted.

As for the *timestamp* format, a default timezone (`tzlocal()`) is automatically attached to the datetime.

Example

```
>>> import datetime
```

Let's construct some datetime objects:

```
>>> dt = datetime.datetime(2014, 6, 19, 17, 58, 5)
>>> dt_dict = {'y':2014, 'm':6, 'd':19, 'hr':17, 'mn':58,  'sec':5}
```

Again, we can use dictionary of positional arguments to pass the proper format:

```
>>> dt == Chronometer.datetime(dt_dict, **{'fmt':'datetime'})
    True
```

(continues on next page)

(continued from previous page)

```
>>> dt == Chronometer.datetime("Thu Jun 19 17:58:05 2014", fmt=
↪'datetime')
    True
```

By default, it behaves 'well' in many circumstances:

```
>>> dt == Chronometer.datetime(dt_dict)
    True
>>> Chronometer.datetime(dt_dict) == Chronometer.datetime(**dt_dict)
    True
```

However:

```
>>> Chronometer.datetime(dt, fmt='dict')
    {'second': 5, 'hour': 17, 'year': 2014, 'day': 19, 'minute': 58,
↪'month': 6}
```

Many more conversions are possible though:

```
>>> Chronometer.datetime(dt, **{'fmt':'%Y-%m-%dT%H:%M:%S'})
    '2014-06-19T17:58:05'
>>> Chronometer.datetime(dt_dict, **{'fmt':'%Y-%m-%dT%H:%M:%S'})
    '2014-06-19T17:58:05'
>>> Chronometer.datetime(dt_dict, **{'fmt':'%Y-%m-%d %H:%M:%S+00'})
    '2014-06-19 17:58:05+00'
>>> Chronometer.datetime(dt, fmt='%Y-%m-%d %H:%M:%S')
    '2014-06-19 17:58:05'
>>> Chronometer.datetime(dt_dict, fmt='%Y-%m-%d')
    '2014-06-19'
>>> Chronometer.datetime(dt, **{'fmt':True}) # use ctime()
    "Thu Jun 19 17:58:05 2014"
>>> Chronometer.datetime(dt_dict, fmt='calendar')
    (2014, 25, 4)
>>> Chronometer.datetime(dt_dict, fmt='iso')
    '2014-06-19T17:58:05'
>>> Chronometer.datetime(dt, **{'fmt':'iso'})
    '2014-06-19T17:58:05'
>>> Chronometer.datetime("Thu Jun 19 17:58:05 2014")
    {'second': 5, 'hour': 17, 'year': 2014, 'day': 19, 'minute': 58,
↪'month': 6}
```

Mind the UTC:

```
>>> import dateutil
>>> utc_tz = dateutil.tz.tzutc() # note: same as timing.UTC_TZ
>>> dt_utc = dt.replace(tzinfo=utc_tz)
>>> Chronometer.datetime(dt_utc, **{'fmt': 'timestamp'})
    1403200685
```

As desired, the operation is idempotent (when the right parameters are passed!):

```
>>> f_dt = Chronometer.datetime(dt, fmt='%Y-%m-%d %H:%M:%S')
>>> dt == Chronometer.datetime(f_dt, **{'fmt': 'datetime'})
    True
>>> f_dt_utc = Chronometer.datetime(dt_utc, fmt=timestamp)
>>> dt_utc == Chronometer.datetime(f_dt_utc, fmt=datetime)
    True
```

As for the *no_micro_secs* keyword argument, it is useful to avoid some 'unpleasant' notation:

```
>>> dt = datetime.datetime.now()
>>> dt
    datetime.datetime(2014, 8, 18, 14, 56, 17, 821000)
>>> print dt
    2014-08-18 14:56:17.821000
>>> Chronometer.datetime(dt, fmt='iso')
    '2014-08-18T14:56:17'
>>> Chronometer.datetime(dt, fmt='iso', no_micro_secs=False)
    '2014-08-18T14:56:17.821000'
```

Though it does not affect most representations:

```
>>> Chronometer.datetime(dt, fmt=True, no_micro_secs=False)
'Mon Aug 18 14:56:17 2014'
```

classmethod gt(*time1*, *time2*)

Compare two date/time instances: check that the first entered time instance is posterior (after) to the second one.

```
>>> resp = Chronometer.gt(time1, time2)
```

Parameters **time1,time2** (*datetime.datetime, str, float, dict*) – time instances whose format are any accepted by *Chronometer.datetime()*.

Returns **resp** (*bool*) – `True` if *time1*>'time2', i.e. *time1* represents a time posterior to *time2*; `False` otherwise.

Example

```
>>> dt = datetime.datetime(2014, 6, 19, 17, 58, 5)
>>> one_day = {'y':2014, 'm':6, 'd':19, 'hr':17, 'mn':58,  'sec':5}
>>> the_day_after = one_day.copy()
>>> the_day_after.update({'d': the_day_after['d']+1})
>>> Chronometer.gt(the_day_after, one_day)
    True
```

See also:

lt(), *gte()*

classmethod gte(*time1*, *time2*)

Compare two date/time instances: check that the first entered time instance is posterior to or simultaneous with the second one the second one.

```
>>> resp = Chronometer.gte(time1, time2)
```

Example

```
>>> Chronometer.gte(one_day_iso,one_day) and not Chronometer.gt(one_
↪day_iso,one_day)
    True
```

See also:

gt(), *lte()*

classmethod lt(*time1*, *time2*)

Compare two date/time instances: check that the first entered time instance is prior (before) to the second one.

```
>>> resp = Chronometer.lt(time1, time2)
```

Parameters **time1,time2** (*datetime.datetime, str, float, dict*) – see *gt()*.

Returns **resp** (*bool*) – `True` if *time1*'<'*time2*, i.e. *time1* represents a time prior to *time2*; `False` otherwise.

Example

Following *gt()* example:

```
>>> Chronometer.lt(one_day,the_day_after)
    True
>>> one_day_iso = Chronometer.datetime(one_day, **{'fmt': 'iso'})
>>> one_day_iso
    '2014-06-19T17:58:05'
>>> Chronometer.lt(one_day_iso,the_day_after)
    True
```

See also:

gt(), *lte()*

classmethod lte(*time1*, *time2*)

Compare two date/time instances: check that the first entered time instance is prior to or simultaneous with the second one.

```
>>> resp = Chronometer.lte(time1, time2)
```

Example

```
>>> Chronometer.lte(one_day_iso,one_day) and not Chronometer.lt(one_
↪day_iso,one_day)
    True
```

See also:

lt(), *gte()*

classmethod since(***kwargs*)

Calculate a beginning date(time) given a duration and an ending date(time).

```
>>> s = Chronometer.since(**kwargs))
```

Keyword Arguments

- **until** (*datetime.datetime, str, float, dict*) – ending date instance whose format is any of those accepted by *Chronometer.datetime()*.
- **span** (*datetime.timedelta, str, float, dict*) – duration expressed in any of the formats accepted by *Chronometer.timedelta()*.
- **fmt** (*str*) – additional parameter for formatting the output result: see *Chronometer.datetime()*.

Returns **s** (*datetime.datetime, str, float, dict*) – beginning date estimated from `until` and `span` timing arguments, expressed in any format as in `:literal:`fmt` and accepted by *Chronometer.datetime()*.

Example

```
>>> since = datetime.datetime.now() # this is: datetime.
↪datetime(2014, 8, 19, 16, 27, 41, 629000)
>>> print since
    2014-08-19 16:27:41.629000
>>> span = '1d'
>>> until = since + datetime.timedelta(1)
>>> print until
    2014-08-20 16:27:41.629000
>>> print Chronometer.since(until=until, span=span)
    {'second': 41, 'hour': 16, 'year': 2014, 'day': 20, 'minute': 27,
↪ 'month': 8}
>>> since_c = Chronometer.since(until=until, span=span, **{'fmt':
↪'datetime'})
>>> print since_c # this is: datetime.datetime(2014, 8, 19, 16, 23,␣
↪27)
    2014-08-20 16:27:41
>>> since_c == since
    False
>>> since_c - since # we missed the microseconds...
    datetime.timedelta(0, 86399, 371000)
```

Again, we can ensure precision by including the microseconds in the calculation:

```
>>> since_c = Chronometer.since(until=until, span=span,
...                             **{'fmt':'datetime',
...                                'no_micro_secs': False})
>>> print since_c # this is datetime.datetime(2014, 8, 20, 16, 27,␣
↪41, 629000)
    2014-08-19 16:27:41.629000
>>> since_c == since
    True
```

Other results are as expected, independently of the format of the input arguments:

```
>>> until_iso = Chronometer.datetime(until, **{'fmt':'iso'})
>>> print until_iso
    2014-08-20T16:27:41
>>> Chronometer.since(**{'until':until_iso, 'span':span, 'fmt':'dict
↪')
    {'second': 41, 'hour': 16, 'year': 2014, 'day': 19, 'minute': 27,
↪ 'month': 8}
```

(continues on next page)

(continued from previous page)

classmethod span(***kwargs*)

Calculate the timing span (duration) in between two given beginning and ending date(time)s.

```
>>> d = Chronometer.span(**kwargs)
```

Keyword Arguments

- **since,until** (*datetime.datetime, str, float, dict*) – beginning and ending (respectively) time instances whose formats are any of those accepted by `Chronometer.datetime()` (not necessarly identical for both instances).
- **fmt** (*str*) – additional parameter for formatting the output result: see `timedelta()`; default is a :class'datetime.datetime' format.

Returns **d** (*datetime.timedelta, str, float, dict*) – duration between the *since* and *until* *timing instances, expressed in any the format passed in 'fmt* and accepted by `Chronometer.timedelta()`.

Example

```
>>> since = datetime.datetime.now()
>>> print since
    2014-08-18 19:30:40.970000
>>> until = since + datetime.timedelta(10) # 10 days
>>> until # this is a datetime
    datetime.datetime(2014, 8, 28, 19, 30, 40, 970000)
>>> print until # which displays microseconds
    2014-08-28 19:30:40.970000
>>> print until - since
    10 days, 0:00:00
>>> Chronometer.span(since=since, until=until)
    datetime.timedelta(10)
>>> Chronometer.span(since=since, until=until, fmt='str')
    '864000.0sec'
```

The fact that microseconds are not taken into account ensures the consistency of the results when converting in between different formats:

```
>>> until_iso = Chronometer.datetime(until, **{'fmt':'iso'})
>>> until_iso
    '2014-08-28T19:30:40'
>>> print until_iso # microseconds have been dumped...
    2014-08-28T19:30:40
>>> Chronometer.span(since=since, until=until_iso)
    datetime.timedelta(10) # 10 days as set
```

However, if we were precisely taking into account the microseconds:

```
>>> Chronometer.span(since=since, until=until_iso, no_micro_
↪secs=False)
    datetime.timedelta(9, 86399, 30000)
```

while this obviously does not affect the precise calculation:

```
>>> Chronometer.span(since=since, until=until, no_micro_secs=False)
    datetime.timedelta(10)
```

classmethod timedelta(**span, **kwargs*)

Perform some duration calculations and conversions.

```
>>> dt = Chronometer.timedelta(*span, **kwargs)
>>>
```

Parameters **span** (*datetime.timedelta, str, float, dict*) – an object specifying a duration, e.g. it can be any form likewise:

- a `datetime.timedelta` object: `datetime.timedelta(2014, 6, 19, 17, 58, 5)`,
- an explicit duration string: `'2900mn 45d 500sec 34hr 4m 2y'`, (where order does not matter),
- an equivalent dictionary-like date: `{'years':2, 'months':4, 'days':45, 'hours':34, 'minutes':2900, 'seconds':500}`.

When the date is expressed as an 'explicit string' or a dictionary, a time duration unit can be any string in `['days', 'hours', 'minutes', 'months', 'seconds', 'years']`.

'zero' and 'hour' are also accepted instead of an explicit duration, and used to represent both null and 1-hour durations.

Keyword Arguments

- **fmt** (*str*) – variable specifying the desired output duration format: it can be any string among:
  - 'timedelta' to output a *datetime* object,
  - 'dict' to output a dictionary indexed by the timing units, where a key is any string in `['days', 'hours', 'minutes', 'months', 'seconds', 'years']`,
  - 'str' to output an explicit duration string (not used).
- **timing** (*str*) – if instead, some conversions is expected, this is used to pass the desired output format: it can be any string in `['y', 'm', 'd', 'hr', 'mn', 'sec']`.

Example

```
>>> import datetime
>>> Chronometer.timedelta('hour')
    '3600.0sec'
```

Let's construct some timedelta objects that are equivalent:

```
>>> td = datetime.timedelta(900, 57738, 80000)
>>> td_str = '45d 34hr 2900mn 4m 500sec 2y'
>>> td_dict = {'y':2, 'm':4, 'd':45, 'hr': 34, 'mn': 2900, 'sec':␣
↪500}
```

as to check the consistency:

```
>>> Chronometer.timedelta(td_str, fmt='dict') == td_dict
    True
>>> td == Chronometer.timedelta(td_str, **{fmt: timedelta}),
    True
>>> Chronometer.timedelta(td_str, dict=True)
    {'d': 45, 'hr': 34, 'mn': 2900, 'm': 4, 'sec': 500, 'y': 2}
>>> Chronometer.timedelta(td_dict) == Chronometer.timedelta(td_dict_
↪bis)
    True
>>> Chronometer.timedelta(td_str) == Chronometer.timedelta('4m 34hr␣
↪500sec 2y 2900mn 45d ')
    True
>>> Chronometer.timedelta(td_str, **{timing: True}) == td_dict
    True
>>> Chronometer.timedelta(td_dict, timing=True) == td_str
```

Note that these one work too:

```
>>> Chronometer.timedelta(**td_dict) == td_str
    True
>>> Chronometer.convert_time_units(d, **td_dict) > td.days
    True
```

classmethod units_to (*from_, to, time=1.0*)

Perform simple timing units conversion.

```
>>> t = Chronometer.units_to(from, to, time=1.)
```

Parameters

- **from,to** (*str*) – 'origin' and 'final' units: any strings in `['y', 'm', 'd', 'hr', 'mn', 'sec']`.
- **time** (*float*) – timing value to convert.

Examples

```
>>> Chronometer.units_to('mn', 'hr',  time=60) == 1
    True
>>> Chronometer.units_to('sec', 'd',  10) == 10*Chronometer.SEC_TO['d
↪']
    True
>>> Chronometer.units_to('hr', 'sec',  5) == 5*Chronometer.HR_TO['sec
↪']
    True
```

classmethod until (***kwargs*)

Calculate a beginning date(time) given a duration and an ending date(time).

```
>>> u = Chronometer.until(**kwargs))
```

Keyword Arguments

- **since** (*datetime.datetime, str, float,* dict) – beginning date instance whose format is any of those accepted by *Chronometer.datetime()*.
- **span, fmt** – see *Chronometer.since()*.

Returns

- **u** (*datetime.datetime, str, float, dict*) – ending date estimated from since and span timing arguments, expressed in any format as in :literal:`fmt and accepted by *Chronometer.datetime()*.
- **'second'** (*41, 'hour': 16, 'year': 2014, 'day': 20, 'minute': 27, 'month': 8*)

Example

```
>>> since = datetime.datetime.now() # this is: datetime.
↪datetime(2014, 8, 19, 16, 45, 5, 94000)
>>> print since
    2014-08-19 16:45:05.094000
>>> span = datetime.timedelta(2) # 2 days
>>> print span
    2 days, 0:00:00
>>> until = Chronometer.until(since=since, span=span, fmt=iso)
>>> print until
    '2014-08-21T16:45:05'
>>> Chronometer.since(until=until, span=span, fmt=datetime)
    datetime.datetime(2014, 8, 19, 16, 45, 5)
```

class sketch.timing.**Timing**(***kwargs*)

Bases: kinki.sketch.base.Base

Timing is the time when something happens or the spacing of events in time.

Initialisation of a *Timing* instance:

```
>>> x = Timing(**kwargs)
```

_status [dict] a dictionary defining the status of a *Timing* instance in a unique way. Its keys are a subset of ['timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']. They can be set through keyword arguments with same name.

clean()

Clean the _status attribute of a *Timing* instance: pop all items that have a value in [(), None, [], ''].

```
>>> x.clean()
```

clear()

Clear the status of a *Timing* instance, i.e. reset its _status attribute to an empty dictionary {}.

```
>>> x.clear()
```

clearSince(***kwargs*)

Clear the *since* attribute (only) of this instance.

clearSpan()
Clear the *span* attribute (only) of this instance.

clearUntil()
Clear the *until* attribute (only) of this instance.

copy()
Copy of a *Timing* instance.

```
>>> y = x.copy()
```

Returns **y** (*Timing*) – a new instance initialised with a copy of its `_status` dictionary attribute, i.e. so that `x.status==y.status`.

filter(*keys*)
Filter the status items of a *Timing* instance.

```
>>> status = x.filter(*keys)
```

Parameters **keys** (*optional, list of str, str*) – a subset of `['timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`; when `()` or `None`, it is set by default to the set `['since', 'until', 'span']`.

Returns **status** (*dict*) – a dictionary representing some status whose keys are *keys* and values are those found in the `_status` attribute of the class, say it otherwise `status = dict([(k,x._status.get(k)) for k in keys])`.

get()
Get an object status, i.e. return its `_status` attribute.

```
>>> status = x.get()
```

getSince(***kwargs*)
Get/retrieve the *since* attribute already stored in this instance, or estimate it from the keywords passed as arguments and/or from the already stored parameters in *status* attribute.

```
>>> s = x.getSince(**kwargs)
```

Keyword Arguments

- **until** (*datetime.datetime, str, float, dict*) – ending date instance whose format is any of those accepted by *Chronometer.datetime()*.
- **span** (*datetime.timedelta, str, float, dict*) – duration expressed in any of the formats accepted by *Chronometer.timedelta()*.
- **fmt** (*str*) – additional parameter for formatting the output result: see *Chronometer.datetime()*.

Returns **s** (*datetime.datetime, str, float, dict*) – estimated beginning date expressed in desired format.

See also:

since()

getSpan(***kwargs*)

Get/retrieve the *until* attribute already stored in this instance, or estimate it from the keywords passed as arguments and/or from the already stored parameters in *status* attribute.

```
>>> d = x.getSpan(**kwargs)
```

Keyword Arguments

- **since,until** (*datetime.datetime, str, float, dict*) – beginning and ending (respectively) time instances whose formats are any of those accepted by *Chronometer.datetime()* (not necessarly identical for both instances).
- **fmt** (*str*) – additional parameter for formatting the output result: see *Chronometer.datetime()*.

Returns **d** (*datetime.timedelta, str, float, dict*) – estimated duration estimated between the *since* and :data:'until timing instances and expressed in desired format.

See also:

span()

getUntil(***kwargs*)

Get/retrieve the *until* attribute already stored in this instance, or estimate it from the keywords passed as arguments and/or from the already stored parameters in *status* attribute.

```
>>> u = x.getUntil(**kwargs)
```

Keyword Arguments

- **since** (*datetime.datetime, str, float, dict*) – beginning date instance whose format is any of those accepted by *Chronometer.datetime()*.
- **span** (*datetime.timedelta, str, float, dict*) – duration expressed in any of the formats accepted by *Chronometer.timedelta()*.
- **fmt** (*str*) – additional parameter for formatting the output result: see *Chronometer.datetime()*.

Returns **u** (*datetime.datetime, str, float, dict*) – estimated ending date expressed in desired format.

See also:

until()

id

Property (`getter`) expressing the unique Id of an instance. It is based on the hash represention (SHA1) of the items of the `_status` attribute dictionary that uniquely defines the instance.

items()

Return the *items* of the `_status` attribute (see also `status` property).

Note: As we want to be able to use this method like the standard *items()* one it refers to, e.g. want to be able to call `items()` instead of just `items`, we define it as regular method, not a property.

See also:

keys(), *values()*

keys()

Return the *keys* of the `_status` attribute, e.g. nothing else than a list composed of a subset of `['timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`.

See also:

items(), *values()*

set(***kwargs*)

Set/update the status of a *Timing* instance by setting all the items of its `_status` attribute that have not been set already consistently with other items that have been already inserted (e.g. in `__init__()`) or with the items passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments **kwargs** (dict) – dictionary of arguments used to set the current status; its keys should be a subset of `['timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`.

Note:

The *set()* method is essentialy running different settings methods, namely `['setSince', 'setUntil', 'setSpan']`, that are consistent with each other.

- The items in `kwargs` have 'priority' over those already stored in the `_status` attribute of the class, i.e. the `_status` dictionary will be updated with the `kwargs` dictionary.
- The completion of missing items consistent with those provided is ensured through the method `__call__()` which is ran prior to updating the status.
- Another control is performed which consists in checking that at least one priority argument (i.e. an item with key in `['since', 'until', 'span']`) has been passed.

setSince(**since*, ***kwargs*)

Set/update the *since* attribute of this instance consistently with parameters in *since* and/or keywords passed as arguments, but also possibly with the already stored parameters in *status* itself.

```
>>> x.setSince(*since, **kwargs)
```

Parameters **since** (*datetime.datetime, str, float, dict*) – a date defining a starting date and expressed in any the format passed in `fmt` and accepted by *Chronometer.datetime()*.

Keyword Arguments **until, span, fmt** – see *getSince()*.

setSinceNow (***kwargs*)

Set the *since* to the instant UTC time(*'now'*), and update all other already stored parameters (*until* and *span*) in *status* consistently with it and possible additional arguments passed as keywords.

```
>>> x.setSinceNow(**kwargs)
```

Keyword Arguments **until, span, fmt** – see *getSince()*: it is either of those two arguments that can be passed, not both.

setSpan (**span, **kwargs*)

Set/update the *span* attribute of this instance consistently with parameters in *span* and/or keywords passed as arguments, but also possibly with the already stored parameters in *status* itself.

```
>>> x.setSpan(*span, **kwargs)
```

Parameters **span** (*datetime.timedelta, str, float, dict*) – a duration expressed in any of the formats accepted by *Chronometer.timedelta()*.

Keyword Arguments **since, until, fmt** – see *getSpan()*.

setUntil (**until, **kwargs*)

Set/update the *until* attribute of this instance consistently with parameters in *until* and/or keywords passed as arguments, but also possibly with the already stored parameters in *status* itself.

```
>>> x.setUntil(*until, **kwargs)
```

Parameters **until** (*datetime.datetime, str, float, dict*) – a date defining an ending date and expressed in any the format passed in `fmt` and accepted by *Chronometer.datetime()*.

Keyword Arguments **since, span, fmt** – see *getUntil()*.

setUntilNow (***kwargs*)

Set the *until* to the instant UTC time(*'now'*), and update all other already stored parameters (*since* and *span*) in *status* consistently.

```
>>> x.setUntilNow(**kwargs)
```

Keyword Arguments **since, span, fmt** – see *getUntil()*: it is either of those two arguments that can be passed, not both..

since

Datetime *since* property (`getter/setter`) of a *Timing* instance. Since date/time attribute:

- can be any *datetime* format accepteb by *Chronometer.datetime()* in input.

- is expressed as a `datetime.datetime` object in output (i.e. fmt='datetime').

Example

```
>>> t = Timing()
>>> t.unix_since = 'Fri Apr 04 18:22:13 2014' # see method below
>>> print t.unix_since # we are really confused, but things got␣
↪fixed
    1396628533
>>> t.since # well we also got this one right
    datetime.datetime(2014, 4, 4, 18, 22, 13, tzinfo=tzlocal())
>>> print t.since
    2014-04-04 18:22:13+02:00
```

span

Duration property (`getter/setter`) of a *Timing* instance. A time spanning entry:

- can be any *timedelta* format accepted by *Chronometer.timedelta()* in input.
- is expressed in `{DEF_TIMEDELTA_UNIT}` units in output.

status

Status property (`getter/setter`) – return the `_status` attribute dictionary.

timing

Timing property of a *Timing* instance – return a dictionary with *since*, *until* and *span* attributes.

Example

```
>>> t = Timing()
>>> t.since = 'Fri Apr 04 18:22:13 2014'
>>> t.until = 1396628833 # default local timezone will be added to␣
↪unix timestamp
>>> t.set(span=True)
>>> print t.timing
    {'since': datetime.datetime(2014, 4, 4, 18, 22, 13), 'span': 5.0,
    'until': datetime.datetime(2014, 4, 4, 18, 27, 13,␣
↪tzinfo=tzlocal())}
```

unix_since

Unix datetime *since* equivalent property (`getter/setter`) of a *Timing* instance. Unix since date/time attribute:

- can be any *datetime*-like format accepteb by *Chronometer.datetime()* in input.
- is expressed as a classical unix timestamp in output (i.e. fmt='timestamp').

Example

```
>>> t = Timing()
>>> t.unix_since = 'Fri Apr 04 18:22:13 2014'
>>> print t.unix_since
    1396628533
```

unix_until
Unix datetime *until* property (getter/setter) of a *Timing* instance. Ibid, see *unix_since()*.

until
Datetime *until* property (getter/setter) of a *Timing* instance. Ibid, see *Timing.since()*.

update(*reset=True*, ***kwargs*)
Update an item of the _status attribute of a *Timing* instance without updating other: may create inconsistent status.

```
>>> x.update(reset=True, **kwargs)
```

Parameters **reset** (*bool*) – boolean variable defining the 'degree' of update; set to True when all status items are cleared and replaced by kwargs (default), False when only those fields present in kwargs are updated.

Keyword Arguments **kwargs** (dict) – dictionary of keyword arguments used to update the current status; its keys should be a subset of ['timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time'].

values()
Return the *values* of the _status attribute.

See also:

items(), *keys()*

quotation Module

Module for text definition, quote extraction and translation.

About

credits: grazzja

version: 0.1 – *since*: Thu Apr 03 13:25:56 2014

Description

Perform basic linguistic operations, that is:

- language enquiry and detection (e.g. external libraries goslate or pycountry),
- text translation through external online APIs (e.g. Google Translator) also accessed using libraries goslate and Google Translate,
- text entities extraction using ttpand/or twitter-text.

Define a class to enable quoting and running such operations.

Dependencies

call: base, browser, extractor, imports, decorators

require: warnings, re, urllib

optional goslate, pycountry, twitter_text, GoogleTranslator

Contents

class sketch.quotation.**Interpreter**

Bases: object

Sketch Interpreter.

Class of methods providing various tools for text interpretation, extraction and translation.

Implicitely inherit most of the methods from original goslate and/or Google Translate, as well as ttp and/or twitter-text libraries through internal composition.

EXTRACTOR = <sketch.quotation._extractor object>

Extractor defined as a (static) *_extractor* instance.

GOSLATE = <goslate.Goslate object>

Translator defined as a (static) goslate.Goslate instance.

GOTRANS = <sketch.quotation._googleTranslator object>

Translator defined as a (static) *_googleTranslator*.

LANGS = {u'af': u'Afrikaans', u'am': u'Amharic', u'ar': u'Arabic', u'az': u'Az

List of ISO6391-coded languages recognised.

See also:

goslate.Goslate.get_languages, pycountry.languages.get

LANGUAGES = [u'Gujarati', u'Chinese (Traditional)', u'Scots Gaelic', u'Irish', u'G

Corresponding language names.

classmethod check(*text*, *lang=None*, *language=None*)

Check language definition for internal consistency.

```
>>> language, lang = Interpreter.check(text, lang=None,␣
↪language=None)
```

See also:

detect()

classmethod detect(*text*)

Class method for language detection.

Detect the language of a given text.

```
>>> lang = Interpreter.detect(text)
```

Parameters **text** (*str*) – original text.

Returns **lang** (*str*) – ISO6391 formatted language description.

Raises InterpreterError: – when detection failed.

Note: The detector provided by goslate is used in priority when the corresponding module is loaded.

Examples

The language of a text can be detected automatically:

```
>>> german_text = 'Einen schonen Tag allerseits'
>>> Interpreter.detect(german_text) == 'de'
    True
>>> english_text = 'what a beautiful day'
>>> Interpreter.detect(english_text) == 'en'
    True
>>> esperanto_text = 'Saluton! Mia nomo estas Frank. Kaj via?'
>>> Interpreter.detect(esperanto_text) == 'eo'
    True
```

classmethod extract (*text*, **attr*, ***kwargs*)

Class method for text entities extraction.

Extract all entities: usernames, lists, hashtags, cashtags and URLs from a text.

```
>>> entity = Interpreter.extract(text, *attr, **kwargs)
```

Parameters

- **text** (*str*) – original text to analyse.
- **attr** (*str, tuple*) – entity(ies) to extract, any string or list of strings of arguments in `['entities', 'reply', 'cashtag', 'tag', 'hashtag', 'mention', 'url', 'user']`.

Keyword Arguments **kwargs** – arguments used in `ExtExtractor` to define the transform associated to the extracted entities extractor; default is `{}`.

Returns **entity** (*dict*) – a dictionary matching the field of entity defined in `attr` with those found in `text`.

Raises InterpreterError: – when the entities to extract defined in `attr` is not recognised.

Examples

Entities can be extracted from a text:

```
>>> text = "@flyingpig So many $bank pigs seem to #pig die @swimming_
↪pig while eating an #cool apple."
>>> hashtag = Interpreter.extract(text,'hashtag')
>>> hashtag['hashtag'] == ['pig', 'cool']
```

True

```
>>> cashtag = Interpreter.extract(text,'cashtag')
>>> cashtag['cashtag'] == ['bank']
    True
>>> tag = Interpreter.extract(text,'tag')
>>> tag['tag'] == 'So many pigs seem to die while eating an apple.'
    True
>>> mention = Interpreter.extract(text,'mention')
```

(continues on next page)

(continued from previous page)

```
>>> [x.get('screen_name') for x in mention['mention']] == ['flyingpig
↪', 'swimming_pig']
    True
```

Entities can be extracted alltogether as well as their position in the text:

```
>>> print Interpreter.extract(text,'entities')
    {'entities': [{'indices': [0, 10], 'list_slug': '', 'screen_name
↪': u'flyingpig'},
                  {'indices': [19, 24], 'cashtag': u'bank'},
                  {'indices': [38, 42], 'hashtag': u'pig'},
                  {'indices': [47, 60], 'list_slug': '', 'screen_name
↪': u'swimming_pig'},
                  {'indices': [77, 82], 'hashtag': u'cool'}]}
```

Same for URLs:

```
>>> text = "So many pigs seem to die while http://flyingpig.com/
↪apple eating an apple."
>>> url = Interpreter.extract(text,'url')
>>> url['url'] == ['http://flyingpig.com/apple']
    True
```

static iso6391tolanguage(*lang*)

Define the full lenght name of a language associated to a given ISO6391 code representation.

Example

```
>>> Interpreter.iso6391tolanguage('en') == u'English'
    True
```

static languagetoiso6391(*language*)

Define the ISO6391 code representation of a given language name.

Example

```
>>> Interpreter.languagetoiso6391('Italian') == u'it'
    True
```

classmethod translate(*text*, ***kwargs*)

Class method for text translation.

Translate a given text from one language to another. If the original language is not defined, auto-detects it from the text.

```
>>> text = Interpreter.translate(text, **kwargs)
```

Parameters **text** (*unicode, str*) – basestring storing the text to translate.

Keyword Arguments

- **lang_from** (*str*) – ISO6391-coded origin language.
- **lang_to** (*str*) – ISO6391-coded destination language.

Returns **text** (*unicode, str*) – basestring storing the translated text.

Raises InterpreterError: – when language is not supported or translation failed.

Note: The translator provided by `goslate` is used in priority when the corresponding module is loaded.

Examples

Text can be translated 'from'/'to' languages:

```
>>> italian_text = 'Sono un berlinese'
>>> Interpreter.translate(italian_text, lang_from='it', lang_to='de
↪') == "Ich bin ein Berliner"
    True
```

In the absence of 'from' language, it is detected:

```
>>> Interpreter.translate(italian_text, lang_to='de') = "Ich bin ein␣
↪Berliner"
    True
>>> Interpreter.translate(italian_text, lang_to='el') == u
↪'\u0395\u03af\u03bc\u03b1\u03b9 Berliner'
    True
```

In the absence of 'to' language, default is set to English:

```
>>> Interpreter.translate(italian_text) == "I am a Berliner"
    True
```

class `sketch.quotation.`**`Quotation`**(***kwargs*)

Bases: `sketch.base.Base`

Class of text and languages attributes and methods used to define/describe the contextual status of an object.

The term quotation refers to a group of words taken from a text or speech and repeated by someone other than the original author or speaker.

Initialisation of a *`Quotation`* instance:

```
>>> x = Quotation(**kwargs)
```

_status [dict] a dictionary defining the status of a *`Quotation`* instance in a unique way. Its keys are a subset of `['text', 'language', 'lang', 'locale']`. They can be set through keyword arguments with same name.

`clean`()

Clean the `_status` attribute of a *`Quotation`* instance: pop all items that have a value in `[(), None, [], '']`.

```
>>> x.clean()
```

clear()
Clear the quotation *status* fields.

clearLanguage()
Clear both (*language*, *lang*) attributes.

clearLocale()
Clear the *locale* attribute.

clearQuote()
Clear the *quote* attribute.

clearTranslation(**lang_to*)
Clear all translated texts or only that in `lang_to` language(s), depending on the given parameter.

```
>>> quote.clearTranslation(*lang_to)
```

Parameters **lang_to** (*list, str, optional*) – ISO6391-coded language to clear. If `()` or `None`, all translations are cleared.

content
Property expressing the content of a *Quotation* instance – quote epurated from all entities other than text (i.e. `tag` only).

See also:

Interpreter.extract()

copy()
Copy of a *Quotation* instance.

```
>>> y = x.copy()
```

Returns **y** (*Quotation*) – a new instance initialised with a copy of its `_status` dictionary attribute, i.e. so that `x.status==y.status`.

detect()
Detect the language of the text stored in this instance if any.

```
>>> lang = quote.detect()
```

Returns **lang** (*str*) – language of the quote in ISO6391 format; `None` if there the quote has no *content*.

Examples

The language of the content (i.e. text without entities) is detected:

```
>>> quote = Quotation("@volante So many $banca pigs seem to #pig die␣
↪@maiale_gallegiante while eating an #simpatico apple.")
>>> quote.detect() == 'en'
    True
```

Entities (quote without *content*) have no defined language:

```
>>> Quotation("@flyingpig").detect() == None
    True
```

See also:

Interpreter.detect()

extract (**attr*)

Extract entities from the text stored in this instance if any.

```
>>> entity = x.extract(*attr)
```

Parameters **attr** (*str, list of str*) – entity to extract, any string argument in `{ENTITY_KEYS}`.

Returns **entity** (*dict, list of dict*) – dictionary or list of dictionary storing for the desired configured input 'entities' the patterns found in the quote of the instance.

Examples

```
>>> text = "@flyingpig So many $bank pigs seem to #pig die @swimming_
↪pig while eating an #cool apple."
>>> Q = Quotation(text)
>>> Q.extract('hashtag') == [u'pig', u'cool'] # order found in the␣
↪text
    True
>>> Q.extract('reply') == u'flyingpig'
>>> Q.extract('cashtag', 'user') == {'cashtag': [u'bank'], 'user': [u
↪'flyingpig', u'swimming_pig']}
    True
```

If we want to extract all of them at once, with their position in the text:

```
>>> print Q.extract()
    [{'indices': [0, 10], 'list_slug': '', 'screen_name': u'flyingpig
↪'},
    {'indices': [19, 24], 'cashtag': u'bank'}, {{'indices': [38, 42],
↪ 'hashtag': u'pig'},
    {'indices': [47, 60], 'list_slug': '', 'screen_name': u'swimming_
↪pig'},
    {'indices': [77, 82], 'hashtag': u'cool'}]
```

However:

```
>>> Q=Quotation("So many pigs seem to die while eating an apple.")
>>> Q.extract() == None
    True
```

See also:

Interpreter.extract()

filter (*keys*)

Filter the status items of a *Quotation* instance.

```
>>> status = x.filter(*keys)
```

Parameters keys (*optional, list of str, str*) – a subset of `['text', 'language', 'lang', 'locale']`; when `()` or `None`, it is set by default to the set `['text', 'language', 'lang']`.

Returns status (*dict*) – a dictionary representing some status whose keys are *keys* and values are those found in the `_status` attribute of the class, say it otherwise `status = dict([(k,x._status.get(k)) for k in keys])`.

get()

Get an object status, i.e. return its `_status` attribute.

```
>>> status = x.get()
```

getLanguage(***kwargs*)

Get/retrieve the (*language*, *lang*) attributes already stored in this instance, or estimate them from the parameters passed as keyword arguments, as well as from the *quote* attribute if already set.

```
>>> language, lang = quote.getLanguage(**kwargs)
```

Keyword Arguments

- **language** (*str, optional*) – keyword specifying the language (full name) of the text stored in the quote.
- **lang** (*str, optional*) – ibid with ISO6391 code, incompatible with *language*; if *lang* is set to `True`, the language will be automatically detected later on when the *set()* method is invoked.
- **text** (*str, optional*) – when the quote stored in the instance is `None`, `()` or an empty string `''`, this keyword can be used to pass the text of which the language is to be determined.

Returns language, lang (*str*) – both full name and ISO6391 code of the language associated to the considered quote; possibly use *detect()* when the language is not already set.

Examples

```
>>> Q = Quotation('Saluton! Mia nomo estas Frank. Kaj via?')
>>> Q.getLanguage() == ('Esperanto', 'eo')
    True
```

How much to rely on it though?

```
>>> Q = Quotation('Saluton! beaucoup allerseits tomorrow fanciulla')
>>> print Q.getLanguage()
    (u'English', u'en')
```

getLocale(***kwargs*)

Get/retrieve the *locale* attribute already stored in this instance, or estimate it from other

parameters passed as keyword arguments and/or the attributes (`language`, `lang`) of the instance if already set.

```
>>> locale = quote.getLocale(**kwargs)
```

Keyword Arguments

- **locale** (*str, optional*) – *locale* language specification in ISO6391 code.
- **lang** (*str, optional*) – language specification in ISO6391 code: if `locale` is not set, it will used instead.

Returns **locale** (*str*) – the `locale` language associated to the quote in ISO6391 code; by default, the language detected as the language of the text is returned.

getQuote(***kwargs*)

Get/retrieve either the `quote` attribute or a translation of it, when already stored in this instance, or estimate it from the parameters passed as keywords arguments.

```
>>> text = quote.getQuote(**kwargs)
```

Keyword Arguments **lang_to** (*list, str*) – when passed (as an ISO6391-coded language), a translated text is returned (calling `getTranslation()`); by default, the quote itself is returned.

Returns **text** (*str*) – the quote stored in the instance, or a translation of it when *lang_to* is passed as a keyword argument.

Example

```
>>> Q = Quotation('Einen schonen Tag allerseits')
>>> Q.getQuote() == Q.quote
    True
>>> Q.getQuote(lang_to='it') == u'Una bella giornata a tutti'
    True
```

getTranslation(**lang_to*)

Get/retrieve the translations of the text stored in this instance, if any, in all `lang_to` language(s).

```
>>> translation = quote.getTranslation(*lang_to)
```

Parameters **lang_to** (*list, str*) – string or list of strings defining the ISO6391-coded languages to translate the text to.

Returns **translation** (*dict, str*) – dictionary of translated texts, indexed by the languages ISO6391-codes, or single translation when `len(lang_to)==1`.

Note: Not already translated text is translated using `Quotation.translate()`.

Example

```
>>> text = "So many pigs seem to die while eating an apple."
>>> Q = Quotation(text)
>>> print Q.getTranslation('it', 'de', 'fr')
    {'fr': u'Tant de porcs semblent mourir en mangeant une pomme.',
    'de': u'So viele Schweine scheinen zu sterben, während isst␣
↪einen Apfel.',
    'it': u'Tanti i maiali sembrano morire mentre si mangia una mela.
↪'}
```

id

Property (`getter`) expressing the unique Id of an instance. It is based on the hash representation (SHA1) of the items of the `_status` attribute dictionary that uniquely defines the instance.

items()

Return the *items* of the `_status` attribute (see also `status` property).

Note: As we want to be able to use this method like the standard *items()* one it refers to, e.g. want to be able to call `items()` instead of just `items`, we define it as regular method, not a property.

See also:

keys(), *values()*

keys()

Return the *keys* of the `_status` attribute, e.g. nothing else than a list composed of a subset of `['text', 'language', 'lang', 'locale']`.

See also:

items(), *values()*

lang

ISO6391-coded language property (`getter/setter`) of a *Quotation* instance.

language

Language property (`getter/setter`) of a *Quotation* instance.

locale

ISO6391-coded locale property (`getter/setter`) of a *Quotation* instance.

quote

Quote/text property (`getter/setter`) of a *Quotation* instance.

set(***kwargs*)

Set/update the status of a *Quotation* instance by setting all the items of its `_status` attribute that have not been set already consistently with other items that have been already inserted (e.g. in `__init__()`) or with the items passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments **kwargs** (`dict`) – dictionary of arguments used to set the current status; its keys should be a subset of `['text', 'language', 'lang', 'locale']`.

Note:

The `set()` method is essentialy running different settings methods, namely `['setQuote', 'setLanguage', 'setLocale']`, that are consistent with each other.

- The items in `kwargs` have 'priority' over those already stored in the `_status` attribute of the class, i.e. the `_status` dictionary will be updated with the `kwargs` dictionary.
- The completion of missing items consistent with those provided is ensured through the method `__call__()` which is ran prior to updating the status.
- Another control is performed which consists in checking that at least one priority argument (i.e. an item with key in `['text', 'language', 'lang']`) has been passed.

setLanguage(**language*, ***kwargs*)

Set/update the (`language`, `lang`) attributes of this instance consistently with parameters in `language` and/or other parameters passed as keyword arguments, as well as also the `quote` itself.

```
>>> quote.setLanguage(*language, **kwargs)
```

Parameters **language** (*str*) – full name to store as the language of the quote in the `language` field of the instance; `lang` is also automatically set.

Keyword Arguments

- **language,lang** (*str, optional*) – see `getLanguage()`; ignored when *language* is passed.
- **'confidence'** (*float*) – parameter quantifying the confidence in the language accuracy.
- **'is_reliable'** (*bool*) – relative confidence.

Example

```
>>> Q = Quotation('Saluton! Mia nomo estas Frank. Kaj via?')
>>> Q.setLanguage()
>>> Q.language == 'Esperanto'
    True
```

setLocale(**locale*, ***kwargs*)

Set/update the `locale` attribute of this instance consistently with parameters in `locale` and/or other parameters passed as keyword arguments, as well as the language itself.

```
>>> quote.setLocale(*locale, **kwargs)
```

Parameters **locale** (*str*) – ISO6391 code to store as the `locale` language associated to the quote.

Keyword Arguments **lang** (`locale,`) – see `getLocale()`.

setQuote(**text, **kwargs*)

Set/update the quote attribute of this instance consistently with parameters in `text` and/or parameters passed as keyword arguments.

```
>>> quote.setQuote(*text, **kwargs)
```

Parameters **text** (*str, list of str*) – string or list of strings to store (possibly merge) in the instance.

Keyword Arguments **text** (*str, optional*) – when *text* is `None`, `()` or an empty string `''`, this keyword can also be used to pass it.

Example

```
>>> Q1, Q2 = Quotation('a la queue leuleu')
>>> Q2 = Quotation(); Q2.setQuote('a la queue leuleu')
>>> Q1.quote == Q2.quote
    True
```

setTranslation(**lang_to*)

Set/update the translations of the text stored in this instance, if any, in all `lang_to` language(s).

```
>>> quote.setTranslation(*lang_to)
```

Parameters **lang_to** (*str, list of str*) – string or list of strings defining the ISO6391-coded languages to translate the text to.

Note: Results are stored in the `translations` field of the `Quotation` instance and can be retrieved later on using `getTranslation()` method.

Example

```
>>> text = "So many pigs seem to die while eating an apple."
>>> Q = Quotation(text)
>>> Q.setTranslation('it')
>>> 'it' in Q.translations
    True
>>> Q.translations['it'] == 'Tanti i maiali sembrano morire mentre␣
↪si mangia una mela.' # [sic]
    True
```

status

Status property (`getter`/`setter`) – return the `_status` attribute dictionary.

text

Property (`getter` only) emulating *quote*.

translate(*lang_to, **kwargs*)

Translate the text stored in this instance to 'lang_to' language, if any.

```
>>> text = quote.translate(lang_to, **kwargs)
```

Parameters **lang_to** (*str*) – ISO6391-coded origin language.

Keyword Arguments **content** (*bool*) – flag set to `True` when only the *content* of the quote is to be translated, i.e. the quote with text only and no special entities; default: `False`.

See also:

Interpreter.translate()

Examples

```
>>> Q = Quotation('Einen schonen Tag allerseits', lang='de')
>>> Q.translate('it') = u'Una bella giornata a tutti'
    True
```

Even when the language is not set, it behaves 'nicely':

```
>>> Q = Quotation('Einen schonen Tag allerseits')
>>> Q.translate('it') = u'Una bella giornata a tutti'
    True
```

update(*reset=True*, ***kwargs*)

Update an item of the `_status` attribute of a *Quotation* instance without updating other: may create inconsistent status.

```
>>> x.update(reset=True, **kwargs)
```

Parameters **reset** (*bool*) – boolean variable defining the 'degree' of update; set to `True` when all status items are cleared and replaced by `kwargs` (default), `False` when only those fields present in `kwargs` are updated.

Keyword Arguments **kwargs** (`dict`) – dictionary of keyword arguments used to update the current status; its keys should be a subset of `['text', 'language', 'lang', 'locale']`.

values()

Return the *values* of the `_status` attribute.

See also:

items(), *keys()*

class sketch.quotation.**_googleTranslator**

Bases: `object`

class sketch.quotation.**_extractor**

Bases: `object`

Class emulating extractor method from twitter_text.

The original API provides function for the extraction and processing of usernames, lists, URLs and hashtags.

Most of the `_extractor` methods are inherited by `Interpreter` through appropriate composition. As such, when appropriate, the examples provided are ran using `Interpreter`.

cashtag (*text*, ***kwargs*)

Extract cashtags from a text, i.e. strings associated to a *$* flag.

```
>>> cashtag = Interpreter.cashtag(text, **kwargs)
```

Returns **cashtag** (*list*) – list of extracted cashtags.

Raises InterpreterError: – when *ExtExtractor* class is not defined.

Note: The list returned will not include the leading $ character. If the text is None or contains no cashtags an empty list will be returned. If a block is given then it will be called for each cashtag.

---

entities (*text*, ***kwargs*)

Extract entities from a text: usernames, lists, hashtags and URLs along with the indices for where the entity ocurred.

```
>>> entities = Interpreter.entities(text, **kwargs)
```

Parameters **text** (*str*) – string from which to extract entities.

Keyword Arguments **kwargs** – see transform keyword arguments of `extractor.extract_entities_with_indices()`

Returns **entities** (*dict*) – dict with single key 'entities' providing a list of extracted entities (any in `['entities', 'reply', 'cashtag', 'tag', 'hashtag', 'mention', 'url', 'user']`), plus the position of occurrence in the original text.

Raises InterpreterError: – when *ExtExtractor* class is not defined.

Note: If the text is None or contains no entity an empty list will be returned. If a transform is given then it will be called for each entity.

See also:

`extractor.extract_entities_with_indices()`

hashtag (*text*, ***kwargs*)

Extract hashtags from a text, i.e. strings associated to a *#* flag.

```
>>> hashtag = Interpreter.hashtag(text, **kwargs)
```

Returns **hashtag** (*list*) – list of extracted hashtags.

Raises InterpreterError: – when neither *ExtExtractor* nor *ttpParser* classes are defined.

Note: The list returned will not include the leading # character. If the text is None or contains no hashtags an empty list will be returned. If a block is given then it will be called for each hashtag.

mention (*text*, ***kwargs*)

Extract mentions from a text: list of all usernames or lists mentioned along with the indices for where the mention ocurred.

```
>>> mention = Interpreter.mention(text, **kwargs)
```

Returns **mention** (*list*) – list of extracted mention, plus the position of occurrence in the original text.

Raises InterpreterError: – when `ExtExtractor` class is not defined.

Note: If the text is None or contains no username or list mentions, an empty list will be returned. If a transform is given, then it will be called with each username, list slug, the start index, and the end index in the text. The list_slug will be an empty string if this is a username mention.

reply (*text*, ***kwargs*)

Extract replies from a text, i.e. all usernames associated to a @ at the beginning of the text.

```
>>> reply = Interpreter.reply(text, **kwargs)
```

Returns **reply** (*list*) – list of extracted replied-to usernames.

Raises InterpreterError: – when neither *ExtExtractor* nor *ttpParser* classes are defined.

Note: If the text is None or contains no username mentions an empty list will be returned. If a transform is given then it will be called with the username replied to (if any).

tag (*text*, ***kwargs*)

Extract tags from a text, i.e. text other than usernames, hashtags and URLs (everything except what is returned by entities method).

```
>>> tag = Interpreter.tag(text, **kwargs)
```

Returns **tag** (*str*) – special text extracted.

Note: If a transform is given then it will be called for each entity.

url (*text*, ***kwargs*)

Extract URLs (e.g. starting with *http*) inserted in a text.

```
>>> url = Interpreter.url(text, **kwargs)
```

Returns **url** (*list*) – list of extracted URLs.

Raises InterpreterError: – when neither *ExtExtractor* nor *ttpParser* classes are defined.

Note: If the text is None or contains no URLs an empty list will be returned. If a transform is given then it will be called for each URL.

---

user (*text*, ***kwargs*)

Extract usernames from a text, i.e.names associated to a @ flag.

```
>>> user = Interpreter.user(text, **kwargs)
```

Returns **user** (*list*) – list of extracted usernames.

Raises InterpreterError: – when neither *ExtExtractor* nor *ttpParser* classes are defined.

Note: If the text is None or contains no username mentions an empty list will be returned. If a transform is given then it will be called for each username.

---

weather Module

Module for weather reporting and forecasting.

About

credits: grazzja

version: 0.1 – *since*: Tue May 13 11:32:33 2014

Description

Dependencies

***call*: base, location, timing,** timeformatutils, imports, decorators

optional pyowm, yweather, pywapi

Contents

class sketch.weather.**Meteorologist**

Bases: object

Sketch Meteorologist.

Implement various tools for weather conditions estimations and forecast. Provide an interface to most of the methods from original pyowm, yweather, and pywapi APIs (those may require special credentials) through composition ('has-a' relationship, hence not real 'is-a' inheritance relationship).

As to avoid multiple inheritance, the collection of container attributes *WOWM*, *YCLIENT* and *WAPI* are respectively used to embedd the APIs mentioned above. See detailed implementations and usages in of _pyowmAPI and _yweather classes.

WAPI = None
Weather API defined as a (static) instance of `pywapi` class, itself derived from `geoplanetapi` or `yahooGeoPlanet`.

WOWM = <sketch.weather._pyowmAPI object>
Open Weather Map API defined as a (static) instance of `_pyowmAPI` class, itself emulating from `pyowm.OWM`.

YCLIENT = <sketch.weather._yweather object>
Yahoo Weather API defined as a (static) instance of `_yweather` class, itself derived from `yweather.Client`.

classmethod noaa(*station_id*)
Retrieve weather conditions using NOAA stations.

```
>>> data = Meteorologist.noaa(station_id)
```

Parameters **station_id** (*int*) – the ID of the weather station near the necessary location.

Returns **data** (*dict*) – a dictionary of weather data that exists in XML feed.

Note: See module `python-weather` at https://code.google.com/p/python-weather and method `Weather.location2station()` to get the station IDs.

See also:

`pywapi.get_weather_from_noaa()`

classmethod openweather(**location*, ***kwargs*)
Retrieve weather conditions using Open Weather Map API.

```
>>> data = Meteorologist.openweather(*location, **kwargs)
```

Parameters **location** – parameter conveying the location for which to publish a weather report; see *weather_at()*.

Keyword Arguments

- **lat,Lon** (*float*) – parameters conveying the `(lat,Lon)` coordinates of a location.
- **place,woeid,lid** – arguments refining the location definition; see *weather_at()* and *weather_location()* methods.

Returns **data** (*dict*) – a dictionary of weather conditions that exists in XML feed.

Note: Check the original API at: http://openweathermap.org/history, as well as the documentation of the pyowm interface at: https://pyowm.readthedocs.org/en/latest/pyowm.webapi25.html.

Examples

Retrieve Open Weather Map reports in London:

```
>>> w = Meteorologist.openweather('London,uk')
```

```
>>> assert w is not None
>>> assert w.get_wind() is not None
>>> assert w.get_temperature('celsius') is not None
>>> assert w.get_humidity() is not None
```

Different location formats are accepted:

```
>>> Meteorologist.openweather(*Geographer.code('London,uk')) is not␣
↪None
```

See also:

`find_weather_by_coords()`, `weather_at()`, *`weather_location()`*, `openforecast()`

classmethod weather_at(**location, **kwargs*)

Retrieve weather condictions at specific location(s) using either of the Yahoo Weather, Weather_dot_com, or Open Weather Map APIs.

```
>>> data = Meteorologist.weather_at(*location, **kwargs)
```

Parameters **location** – parameter conveying the location for which to publish a weather report; see *`weather_at()`*.

Keyword Arguments

- **wunit** (*str*) – type of units used in weather reports: `'metric'` for metric and `'imperial'` or `''` for non-metric; defaults to `'metric'`.
- **place,woeid,lid** – arguments refining the location definition; see *`weather_at()`* and *`weather_location()`* methods.

Returns **data** (*dict*) – a dictionary of weather data (conditions) that exists in XML feed.

Note: Choosing metric units changes all the weather units to metric, for example, wind speed will be reported as kilometers per hour and barometric pressure as millibars.

Examples

See also:

`openweather()`, *`yahooweather()`*, *`weathercom()`*, *`weather_location()`*

classmethod weather_location(**location, **kwargs*)

Estimate the place name, Yahoo! Weather's location identifier, and Weather Channel's location identifier (in this order) of a given location.

```
>>> place, woe_id, location_id = Meteorologist.weather_
↪location(*location, **kwargs)
```

Parameters **location** (list, str, `Location`, tuple) – parameter conveying the location for which to publish a weather report: it can be a string defining the

location name, a `Location` instance setting the location sketch or a tuple providing the location `(lat,Lon)` coordinates; posibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored; a list of several different locations can also be passed.

Keyword Arguments

- **`place`** – a string defining a location name, e.g. of the form `locatity, country`, for instance `Paris, France`, when `location` is not passed.
- **`woeid`** (*`int`*) – a integer value representing the WOE identifier of a location.
- **`lid`** (*`int`*) – a five digit zip code or location ID.
- **`kwargs`** – any other keyword parameters accepted by `kinki.sketch.location.Location.set()`.

Returns

- **place** (*str*) – place name representing the location conveyed by `location`.
- **woe_id** (*int*) – a WOE identifier representing the location conveyed by `location`.
- **location_id** (*int*) – a NOAA station ID near the location conveyed by `location` if any.

Note: A class representing a location in the world. A location is defined through a toponym, a couple of geographic coordinates such as longitude and latitude and a numeric identifier assigned by the OWM web API that uniquely spots the location in the world. Optionally, the country specification may be provided.

See also https://pyowm.readthedocs.org/en/latest/pyowm.webapi25.html.

Example

```
>>> print Meteorologist.weather_location(place='Paris, France')
    ('Paris, France', 615702, 'FRIF1165')
```

See also:

`pywapi.get_location_ids()`, `pyowm.webapi25.location.Location()`

`classmethod weather_timing`(**timing*, ***kwargs*)

Class method used for retrieving an ISO formatted timing.

```
>>> timing = Meteorologist.weather_timing(*timing, **kwargs)
```

Parameters **`timing`** (datetime.datetime, str, float, dict, `Timing`, list) – an object conveying some time value; posibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored; a list of several different time schedules can also be passed.

Keyword Arguments

- **`datetime`** (datetime.datetime, str, float, dict, `kinki.sketch.timing.Timing`, list) – an object defining the time schedule when `timing` is not passed.

- **kwargs** – any other keyword parameters accepted by `kinki.sketch.timing.Timing.set()`.

Returns timing (*str*) – ISO8601-formatted string (with pattern `'%Y-%m-%d %H:%M:%S+00'`) corresponding to the time value conveyed by the specified object.

Example

```
>>> t = datetime.datetime(2014, 6, 19, 17, 58, 5)
>>> print t
    2014-06-19T17:58:05
>>> t_iso = timing.Chronometer.datetime(t, **{ 'fmt': 'iso' })
>>> t_iso
    '2014-06-19T17:58:05'
>>> print Meteorologist.weather_timing(t)
    2014-06-19 17:58:05+00
>>> print Meteorologist.weather_timing(t_iso)
    2014-06-19 17:58:05+00
```

See also:

`pyowm.utils.timeformatutils.to_ISO8601()`

classmethod weathercom(**location, **kwargs*)

Rretrieve weather conditions using Weather_dot_com API.

```
>>> data = Meteorologist.weathercom(*location, **kwargs)
```

Parameters location – parameter conveying the location for which to publish a weather report; see *weather_at()*.

Keyword Arguments

- **wunit** (*str*) – type of units used in weather reports; see *weather_at()*.
- **place,woeid,lid** – arguments refining the location definition; see *weather_at()* and *weather_location()* methods.

Returns data (*dict*) – a dictionary of weather data that exists in XML feed.

Note: The location ID (e.g. in `lid`) is used by the APIs to establish a report on weather conditions.

Units create issues with metric requests for non-metric countries e.g.:

```
>>> assert pywapi.get_weather_from_weather_com('GRXX0036',
↪wunit='metric')
```

while :

```
>>> assert pywapi.get_weather_from_weather_com('USNY0996',
↪wunit='metric'))
    AssertionError: ...
```

Examples

Retrieve weather conditions in Paris:

```
>>> assert Meteorologist.weathercom('Paris, France') is not None
>>> assert Meteorologist.weathercom(woeid=615702) is not None
```

See also:

`pywapi.get_weather_from_weather_com()`

classmethod yahooweather(**location, **kwargs*)

Retrieve weather conditions using Yahoo Weather API.

```
>>> data = Meteorologist.yahooweather(*location, **kwargs)
```

Parameters **location** – parameter conveying the location for which to publish a weather report; see `weather_at()`.

Keyword Arguments

- **wunit** (*str*) – type of units used in weather reports; see `weather_at()`.
- **place,woeid,lid** – arguments refining the location definition; see `weather_at()` and `weather_location()` methods.

Returns **data** (*dict*) – a dictionary of weather data that exists in XML feed.

Note: The location ID (e.g. in `lid`) is used by the APIs to establish a weather report.

Examples

See also:

`pywapi.get_weather_from_yahoo()`, `yweather.Client.fetch_weather()`

class sketch.weather.**Weather**(***kwargs*)

Bases: `sketch.base.Base`

Class of attributes and methods used to describe the weather conditions associated to a given object.

Weather is the set of weather conditions at a given place or weather forecast at that same place at a given time.

Initialisation of a `Weather` instance:

```
>>> x = Weather(**kwargs)
```

_status [dict] a dictionary defining the status of a `Weather` instance in a unique way. Its keys are a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`. They can be set through keyword arguments with same name.

datetime [str] attribute conveying the time value of the weather reports, when requested.

location [dict] attribute conveying the location of the weather reports, through the unambiguous definition of `[place, woeid, lid]` items.

clean()

Clean the `_status` attribute of a *Weather* instance: pop all items that have a value in `[(), None, [], '']`.

```
>>> x.clean()
```

clear()

Clear the status of a *Weather* instance, i.e. reset its `_status` attribute to an empty dictionary `{}`.

```
>>> x.clear()
```

clearCondition()

Clear the *condition* attribute of the current instance.

clearForecast()

Clear the *forecast* attribute of the current instance.

clearLocation()

Clear the `location` attributes of the current instance.

clearTiming()

Clear the `datetime` attribute of the current instance.

condition

Weather *condition* property (`getter/setter`) of a *Weather* instance. This attribute is a dictionary with various key information on weather conditions.

copy()

Copy of a *Weather* instance.

```
>>> y = x.copy()
```

Returns **y** (*Weather*) – a new instance initialised with a copy of its `_status` dictionary attribute, i.e. so that `x.status==y.status`.

filter(*keys*)

Filter the status items of a *Weather* instance.

```
>>> status = x.filter(*keys)
```

Parameters **keys** (*optional, list of str, str*) – a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`; when `()` or `None`, it is set by default to the set `['forecast', 'condition']`.

Returns **status** (*dict*) – a dictionary representing some status whose keys are *keys* and values are those found in the `_status` attribute of the class, say

it otherwise `status = dict([(k,x._status.get(k)) for k in keys])`.

forecast

Weather *forecast* property (`getter/setter`) of a *Weather* instance. This attribute is a dictionary with various key information on weather forecasts.

get()

Get an object status, i.e. return its `_status` attribute.

```
>>> status = x.get()
```

getCondition(**location*, ***kwargs*)

Get/retrieve the 'weather conditions' attribute already stored in the current instance, or estimate it from other parameters passed as keyword arguments. .

```
>>> conditions = x.getCondition(*location, **kwargs)
```

Parameters **location** – see *setLocation()*.

Keyword Arguments **kwargs** – see *getLocation()*.

getForecast(**location*, ***kwargs*)

Get/retrieve the 'weather forecast' attribute already stored in the current instance, or estimate it from other geolocation and timing parameters passed as keyword arguments.

```
>>> forecast = x.getForecast(*location, **kwargs)
```

Parameters **location** – see *setLocation()*.

Keyword Arguments **kwargs** – see *getTiming()* and *getLocation()*.

getLocation(***kwargs*)

Get/retrieve the (place, WOE_id, location_id) attributes already stored in this instance, or estimate them from the keywords passed as arguments.

```
>>> location = x.getLocation(**kwargs)
```

Keyword Arguments **kwargs** – any other keyword parameters accepted by `kinki.sketch.location.Location.get()`.

Returns **location** (*dict*) – a dictionary unambiguously representing a location through `[place,woeid,lid]` items storing respectively its name, its WOE identifier, the ID of the nearest NOAA station if any.

See also:

weather_location()

getTiming(***kwargs*)

Get/retrieve the `datetime` attribute already stored in this instance, or estimate it from the keywords passed as arguments.

```
>>> timing = x.getTiming(**kwargs)
```

Keyword Arguments kwargs – any other keyword parameters accepted by `kinki.sketch.timing.Timing.get()`.

Returns datetime (*str*) – ISO8601-formatted string (with pattern `'%Y-%m-%d %H:%M:%S+00'`) corresponding to the time value conveyed by the specified object; this format is the one specifically used in `pyowm` module.

See also:

weather_timing()

id

Property (`getter`) expressing the unique Id of an instance. It is based on the hash represention (SHA1) of the items of the `_status` attribute dictionary that uniquely defines the instance.

items()

Return the *items* of the `_status` attribute (see also `status` property).

Note: As we want to be able to use this method like the standard *items()* one it refers to, e.g. want to be able to call `items()` instead of just `items`, we define it as regular method, not a property.

See also:

keys(), *values()*

keys()

Return the *keys* of the `_status` attribute, e.g. nothing else than a list composed of a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`.

See also:

items(), *values()*

set(***kwargs*)

Set/update an object's weather status by first setting/updating its time and location fields that have not been set already.

```
>>> x.set(**kwargs)
```

Keyword Arguments kwargs (`dict`) – dictionary of arguments used to set the current status; its keys should be a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`.

Examples

Note: All geographic and temporal attributes, namely methods `setLocation()` and `setTiming()` are ran, then check already when possible if weather conditions and forecast are available, namely `setCondition()` and `setForecast()` are (optionally) ran.

- The items in `kwargs` have 'priority' over those already stored in the `_status` attribute of the class, namely: `datetime` and `location` attributes. This means that the `_status` dictionary will be updated with the `kwargs` dictionary.
- The completion of missing items consistent with those provided is ensured through the method `__call__()` which is ran prior to updating the status.
- Another control is performed which consists in checking that at least one priority argument (i.e. an item with key in `['forecast', 'condition']`) has been passed.

setCondition(**location*, ***kwargs*)

Set/update the 'weather conditions' attribute of the current instance consistently with `location` argument and/or geolocation parameters passed as keyword arguments.

```
>>> x.setCondition(*location, **kwargs)
```

Parameters **location** – see `setLocation()`.

Keyword Arguments **kwargs** – see `getLocation()`.

setForecast(**location*, ***kwargs*)

Set/update the 'weather forecast' attribute of the current instance through the settings in `location` argument and/or other geolocation and timing parameters passed as keyword arguments.

```
>>> x.setForecast(*location, **kwargs)
```

Parameters **location** – see `setLocation()`.

Keyword Arguments **kwargs** – see `getTiming()` and `getLocation()`.

setLocation(**location*, ***kwargs*)

Set/update the `[place, woeid, lid]` of thitems of this instance consistently with parameters location and/or keywords passed as arguments.

```
>>> x.setLocation(*location, **kwargs)
```

Parameters **location** (list, str, `Location`, tuple) – parameter conveying the location for which to publish a weather report: it can be a string defining the location name, a `Location` instance setting the location sketch or a tuple providing the location `(lat,Lon)` coordinates; posibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored; a list of several different locations can also be passed.

Keyword Arguments **kwargs** – see `getLocation()`.

Examples

```
>>> weather = Weather('Paris, France')
>>> weather.set()
>>> assert weather.getCondition() is not None
>>> assert weather.getForecast(**{ 'datetime': tomorrow}) is not None
```

setTiming(**timing*, ***kwargs*)

Set/update the `datetime` attribute of this instance consistently with parameters `timing` and/or parameters passed as keyword arguments.

```
>>> x.setTiming(*timing, **kwargs)
```

Parameters **timing** (datetime.datetime, str, float, dict, `Timing`, list) – parameter conveying some time value; posibly `()` or `(None,)` so as to consider the keyword arguments in `kwargs` (see below), otherwise all keyword arguments are ignored; a list of several different time schedules can also be passed.

Keyword Arguments **kwargs** – see *`getTiming()`*.

status

Status property (`getter`/`setter`) – return the `_status` attribute dictionary.

As for the `getter` operation, return information related to the actual weather report only, hence the entries for `['forecast', 'condition']` keys; other keys (e.g. thos in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time']`) can be retrieved only by calling explicitely `_status` attribute.

today

Datetime object representing 'today' (the day it is called) – wrapper to `pyowm.timeutils.today()`.

tomorrow

Datetime object representing 'tomorrow' (the day after it is called) – wrapper to `pyowm.timeutils.tomorrow()`.

update(*reset=True*, ***kwargs*)

Update an item of the `_status` attribute of a *`Weather`* instance without updating other: may create inconsistent status.

```
>>> x.update(reset=True, **kwargs)
```

Parameters **reset** (*`bool`*) – boolean variable defining the 'degree' of update; set to `True` when all status items are cleared and replaced by `kwargs` (default), `False` when only those fields present in `kwargs` are updated.

Keyword Arguments **kwargs** (`dict`) – dictionary of keyword arguments used to update the current status; its keys should be a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since',`

'until', 'unix_since', 'unix_until', 'span', 'date', 'time'].

values()
Return the *values* of the _status attribute.

See also:

items(), *keys()*

user Module

Module for user definition and description.

About

credits: grazzja

version: 0.1 – *since*: Fri Jun 13 01:11:32 2014

Description

Dependencies

call: base, imports, decorators

Contents

class sketch.user.**User**(***kwargs*)
Bases: sketch.base.Base

Class of attributes and methods used to describe a user producing a media object.

Initialisation of a *User* instance:

```
>>> x = User(**kwargs)
```

_status [dict] a dictionary defining the status of a *User* instance in a unique way. Its keys are a subset of ['name', 'email', 'user', 'user_id', 'first_name', 'last_name']. They can be set through keyword arguments with same name.

clean()
Clean the _status attribute of a *User* instance: pop all items that have a value in [(), None, [], ''].

```
>>> x.clean()
```

clear()
Clear the status of a *User* instance, i.e. reset its _status attribute to an empty dictionary {}.

```
>>> x.clear()
```

clearName()
Clear all name related attributes (*name*, as well as *firstname* and *lastname*) of the instance.

copy()
Copy of a *User* instance.

```
>>> y = x.copy()
```

Returns **y** (*User*) – a new instance initialised with a copy of its `_status` dictionary attribute, i.e. so that `x.status==y.status`.

email

User email attribute (`getter/setter`).

filter(*keys*)

Filter the status items of a *User* instance.

```
>>> status = x.filter(*keys)
```

Parameters **keys** (*optional, list of str, str*) – a subset of `['name', 'email', 'user', 'user_id', 'first_name', 'last_name']`; when `()` or `None`, it is set by default to the set `['name']`.

Returns **status** (*dict*) – a dictionary representing some status whose keys are *keys* and values are those found in the `_status` attribute of the class, say it otherwise `status = dict([(k,x._status.get(k)) for k in keys])`.

firstname

User firstname attribute (`getter/setter`).

get()

Get an object status, i.e. return its `_status` attribute.

```
>>> status = x.get()
```

getName(***kwargs*)

Get/retrieve the (name, firstname, lastname) attributes already stored in this instance, or estimate them from the keywords passed as arguments.

```
>>> name, firstname, lastname = x.getName(**kwargs)
```

Keyword Arguments

- **name** (*str*) – see *setName()*.
- **first_name** (*str*) – string representing the first name, i.e. the first part of a *name*.
- **last_name** (*str*) – string representing the first name, i.e. the second part of a *name*.

Returns **name, firstname, lastname** (*str*) – string representing the full name, first and last names.

id

Property (`getter`) expressing the unique Id of an instance. It is based on the hash represention (SHA1) of the items of the `_status` attribute dictionary that uniquely defines the instance.

items()

Return the *items* of the `_status` attribute (see also `status` property).

Note: As we want to be able to use this method like the standard *items()* one it refers to, e.g. want to be able to call `items()` instead of just `items`, we define it as regular method, not a property.

See also:

keys(), *values()*

keys()

Return the *keys* of the `_status` attribute, e.g. nothing else than a list composed of a subset of `['name', 'email', 'user', 'user_id', 'first_name', 'last_name']`.

See also:

items(), *values()*

lastname

User lastname attribute (`getter`/`setter`).

name

User (full)name attribute (`getter`/`setter`), e.g. so that `name = '{first}{last}'.format(first=firstname,last=lastname)`.

set(***kwargs*)

Set/update the status of a *User* instance by setting all the items of its `_status` attribute that have not been set already consistently with other items that have been already inserted (e.g. in `__init__()`) or with the items passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments **kwargs** (dict) – dictionary of arguments used to set the current status; its keys should be a subset of `['name', 'email', 'user', 'user_id', 'first_name', 'last_name']`.

Note:

The *set()* method is essentialy running different settings methods, namely `['setName']`, that are consistent with each other.

- The items in `kwargs` have 'priority' over those already stored in the `_status` attribute of the class, i.e. the `_status` dictionary will be updated with the `kwargs` dictionary.
- The completion of missing items consistent with those provided is ensured through the method `__call__()` which is ran prior to updating the status.
- Another control is performed which consists in checking that at least one priority argument (i.e. an item with key in `['name']`) has been passed.

setName(**name*, ***kwargs*)

Set/update the (name, firstname, lastname) attributes of this instance consistently with parameters name and/or keywords passed as arguments.

```
>>>  x.setName(*name, **kwargs)
```

Parameters **name** (*str*) – string to store as the username in the instance; posibly '' when a keyword argument name is passed.

Keyword Arguments

- **name** (*str*) – when *name* is *None*, *()* or an empty string '', this keyword can also be used to pass it.
- **last_name** (*first_name,*) – see *getName()*.

Example

```
>>> U1 = User(); U1.setName('John Smith')
>>> U2 = User(); U2.setName(first_name='John', last_name='Smith')
>>> assert U1.name == U2.name
```

status

Status property (getter/setter) – return the _status attribute dictionary.

update (*reset=True*, ***kwargs*)

Update an item of the _status attribute of a *User* instance without updating other: may create inconsistent status.

```
>>> x.update(reset=True, **kwargs)
```

Parameters **reset** (*bool*) – boolean variable defining the 'degree' of update; set to True when all status items are cleared and replaced by kwargs (default), False when only those fields present in kwargs are updated.

Keyword Arguments **kwargs** (dict) – dictionary of keyword arguments used to update the current status; its keys should be a subset of ['name', 'email', 'user', 'user_id', 'first_name', 'last_name'].

user

User attribute (getter/setter), e.g. a login name.

user_id

User ID attribute (getter/setter), e.g. a login ID.

values ()

Return the *values* of the _status attribute.

See also:

items(), *keys()*

status Module

Module for general status definition.

About

credits: grazzja

version: 0.1 – *since*: Sun May 4 10:49:58 2014

Description

Create well-structured and universal (generic) status dictionaries to be used by the APIs searching and DBs quering modules.

Dependencies

call: `base`, `location`, `Timing`, `quotation`, `Weather`, `decorators`

require: `re`

Contents

class `sketch.status.`**`Status`**(**args*, ***kwargs*)

Bases: `sketch.base.Base`

Status class used to form queries based on location, quotation (text), timing.Timing and user.User profile.

This class is built as a composition of attributes defining different 'properties' of the status. Following, a *`Status`* instance inherits all properties and methods of its properties (see below: *`location`*, *`timing.Timing`*, *`user.User`*, and *`quotation`*).

`location`

attribute storing all geolocation related status fields as a `Location` instance.

`timing.`**`Timing`**

attribute storing all date/time related status fields as a `Timing` instance.

`quotation`

attribute storing all quote/text related status fields as a `Quotation` instance.

`user.`**`User`**

attribute storing all user.User related status fields as a `User` instance.

`_status`

A dictionary defining the complete status of a *`Status`* instance in a unique way. It is in fact a combined dictionary of different status properties, namely its keys are a subset of `['location', 'timing.Timing', 'quotation', 'user.User']`.

Initialisation of a *`Status`* instance:

```
>>> x = Status(*args, **kwargs)
```

Parameters

- **`args`** (*`dict`*) – a dictionary can be passed as to configure the instance; it should be filled with the same items as those accepted by `kwargs`.
- **`kwargs`** – keyword arguments used to set the fields of the instance with same name; its keys are a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`.

`clean`()

Clean the `_status` attribute of a *`Status`* instance: pop all items that have a value in `[(), None, [], '']`.

```
>>> x.clean()
```

clear(**prop*)

Implemented for convenience: see *clearStatus()*.

```
>>> x.clear(*prop)
```

clearStatus(**prop*)

Clear the status of some properties a *Status* instance.

```
>>> x.clearStatus(*prop)
```

Parameters **prop** (*list of str, str, None*) – define the internal properties to clear, see *setStatus()*; when set to `()` or `(None,)`, all properties are cleared, i.e. , the `_status` attribute is reset to an empty dictionary `{}`

See also:

clear()

copy()

Copy of a *Status* instance.

```
>>> y = x.copy()
```

Returns **y** (*Status*) – a new instance initialised with a copy of its `_status` dictionary attribute, i.e. so that `x.status==y.status`.

filter(*keys*)

Filter the status items of a *Status* instance.

```
>>> status = x.filter(*keys)
```

Parameters **keys** (*optional, list of str, str*) – a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`; when `()` or `None`, it is set by default to the set `['bbox', 'lat', 'Lon', 'woeid', 'locality', 'country', 'since', 'until', 'span', 'text', 'language', 'lang', 'name']`.

Returns **status** (*dict*) – a dictionary representing some status whose keys are *keys* and values are those found in the `_status` attribute of the class, say it otherwise `status = dict([(k,x._status.get(k)) for k in keys])`.

get()

Get an object status, i.e. return its `_status` attribute.

```
>>> status = x.get()
```

getStatus(**prop*)

Get/retrieve the status of a *Status* instance already stored in this instance.

```
>>> status = x.getStatus(*prop)
```

Parameters **prop** (*list of str, str, None*) – define the properties to retrieve; it can be (a list of) any string in `['location', 'timing', 'quotation', 'user']`; `'weather.Weather'` is also accepted; when set to `()` or `(None,)`, all properties are retrieved.

Returns **status** (*dict, dict of dict*) – a dictionary of status, possibly indexed by the property names in `prop` when `len(prop)!=1`, otherwise by the status keys of the unique property in `prop` when `len(prop)==1`.

id

Property (`getter`) expressing the unique Id of an instance. It is based on the hash representation (SHA1) of the items of the `_status` attribute dictionary that uniquely defines the instance.

items()

Return the *items* of the `_status` attribute (see also `status` property).

Note: As we want to be able to use this method like the standard *items()* one it refers to, e.g. want to be able to call `items()` instead of just `items`, we define it as regular method, not a property.

See also:

keys(), *values()*

keys()

Return the *keys* of the `_status` attribute, e.g. nothing else than a list composed of a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`.

See also:

items(), *values()*

set(***kwargs*)

Implemented for convenience: see *setStatus()*.

```
>>> x.set(**kwargs)
```

setStatus(**prop, **kwargs*)

Set/update the status of a *Status* instance by setting all the items of its `_status` attribute that have not been set already consistently with other items that have been already inserted (e.g. in `__init__()`) or with the items passed in the keyword arguments.

```
>>> x.setStatus(*prop, **kwargs)
```

Parameters **prop** (*list of str, str, None*) – define the properties to retrieve; it can be (a list of) any string in `['location', 'timing', 'quotation', 'user']`;

Keyword Arguments **kwargs** (`dict`) – dictionary of arguments used to set the current status; its keys should be a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`.

status

Status property – consistently `getter/setter` the fields of the `_status` attribute dictionary.

update(*reset=True*, ***kwargs*)

Update an item of the `_status` attribute of a *`Status`* instance without updating other: may create inconsistent status.

```
>>> x.update(reset=True, **kwargs)
```

Parameters **reset** (*bool*) – boolean variable defining the 'degree' of update; set to `True` when all status items are cleared and replaced by `kwargs` (default), `False` when only those fields present in `kwargs` are updated.

Keyword Arguments **kwargs** (`dict`) – dictionary of keyword arguments used to update the current status; its keys should be a subset of `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`.

values()

Return the *`values`* of the `_status` attribute.

See also:

`items()`, *`keys()`*

2.3 Online data access and fetching

2.3.1 Purpose

The `fetch` package aims at providing various functionalities for fetching data (*e.g.* data from online social media), *i.e.* collecting data with specific *"status profile"*.

Main features:

- relies on existing *Python* interfaces to the APIs,
- does not impose any model on fetched data.

2.3.2 Modules

`media` Package

Package for various configuration parameters of online social media.

Description

Global variables to store personal API keys ('hard' coded in the script) and other keywords arguments that are common to all media.

Usage

```
>>> from kinki.media import ALL_MEDIA, DEF_OWNER_NAME, DEF_OWNER_EMAIL,
...                         DEF_APP_NAME, DEF_APP_NAME, DEF_APP_DESC
```

Contents

`media.`**`ALL_MEDIA`**
List of considered media: `['twitter', 'flickr', 'linkedin', 'foursquare', 'tumblr', 'facebook', 'instagram', 'panoramio']`.

`media.`**`DEFAULT_MEDIA`**
List of default media for testing: `['twitter', 'flickr']`.

`media.`**`DEF_OWNER_NAME`**
Application default owner: `{'last': 'qool', 'first': 'deap'}`.

`media.`**`DEF_OWNER_EMAIL`**
Default owner email: `deapqool@gmail.com`.

`media.`**`DEF_APP_NAME`**
Default generic Application name: `DQTester`.

`media.`**`DEF_APP_DESC`**
Default description of application purpose: `DEaP QooL tester`.

`alchemy` Module

Global variables to store personal Alchemy API keys.

Usage

```
>>> from kinki.media.alchemy import API_KEY
```

Dependencies

call: `media`

Contents

`media.alchemy.`**`API_KEY = 'a753d0a52d6f1ef42e6750a96fb3cb64b6824134'`**
API key.

`media.alchemy.`**`API_KEY_2 = '3fa9cc13ed2fda2fde78587f8bd4a02e7dc0d947'`**
additional API key.

`facebook` Module

Global variables to store personal Facebook API credentials.

Usage

```
>>> from kinki.media.facebook import API_KEY, API_SECRET, API_TOKEN
>>> from kinki.media.facebook import KWARGS_API_SEARCH, KWARGS_API_FILTER
```

Dependencies

call: `media`

Contents

`media.facebook.`**`KWARGS_API_SEARCH`**
Specific search terms.

`media.facebook.`**`KWARGS_API_FILTER`**
Specifc filter terms.

`media.facebook.`**`API_KEY = '248061922044909'`**
API key corresponding to the application ID.

`media.facebook.`**`API_SECRET = 'd4688c9398d0ef3fa2a1f2596475cff9'`**
API secret key.

`media.facebook.`**`API_TOKEN = '248061922044909|2c_BNQz0pgnTYx6s9vMHnVWG8K4'`**
API token.

`flickr` Module

Global variables to store personal Flickr API credentials.

Usage

```
>>> from kinki.media.flickr import API_KEY, API_SECRET
>>> from kinki.media.flickr import KWARGS_API_SEARCH, KWARGS_API_FILTER
```

Dependencies

call: `media`, `location`, `timing`

Contents

`media.flickr.`**`KWARGS_API_SEARCH`**
dict – Specific search terms: see https://www.flickr.com/services/api/flickr.photos.search.html.

media/../_images/facebook_api_keys.png

media/../_images/facebook_api_keys_.png

Fig. 1: *Screen captures of DEaP QooL credentials.*

media/../_images/flickr_api_keys.png

Fig. 2: *Screen capture of DEaP QooL credentials.*

`media.flickr.`**`KWARGS_API_FILTER`**
dict – Specifc filter terms.

`media.flickr.`**`API_KEY = '4e609049d8f70bb3ea1ff0a43e5ae803'`**
API key.

`media.flickr.`**`API_SECRET = '65073669504a52ec'`**
API secret key.

`foursquare` Module

Global variables to store personal Foursquare API credentials.

Usage

```
>>> from kinki.media.foursquare import CLIENT_ID, API_KEY
>>> from kinki.media.foursquare import KWARGS_API_SEARCH, KWARGS_API_FILTER
```

Dependencies

call: `media`, `user`, `location`, `status`

Contents

`media.foursquare.`**`KWARGS_API_SEARCH`**
Specific search terms: see https://developer.foursquare.com/docs/, and also the various API requests:

- user: https://developer.foursquare.com/docs/users/search,
- venues: https://developer.foursquare.com/docs/venues/search,
- tips: https://developer.foursquare.com/docs/tips/search,
- special: https://developer.foursquare.com/docs/specials/search,
- events: https://developer.foursquare.com/docs/events/search.

`media.foursquare.`**`KWARGS_API_FILTER`**
Specifc filter terms.

`media.foursquare.`**`API_KEY = 'OFSHHK5PT4L3S5FEH5W0OHBYBQBQXJMYQWQTXXERLLW3NEOO'`**
API key.

`media.foursquare.`**`CLIENT_ID = '33XJMQEKEUJ51CTHQYYERVTHDMTJ0W0ZSWLI1V24QSVYLNB3'`**
client ID.

`geonames` Module

Global variables to store personal GeoNames username.

Usage

```
>>> from kinki.media import geonames
```

Dependencies

call: `media`

google Module

Global variables to store personal Google Translate API credentials.

Usage

```
>>> from media.google import CLIENT_ID, API_KEY
```

Dependencies

call: media

Contents

media/../_images/google_api_keys.png

media.google.**API_KEY = 'AIzaSyA_uYf_EI850pc_24lTXpN1CgI-w1Jhdjg'**
API key.

media/../_images/google_api_keys_.png

Fig. 3: *Screen captures of DEaP QooL credentials.*

`media.google.CLIENT_ID = '6351027822'`
API client ID.

`linkedin` Module

Global variables to store personal LinkedIn API credentials.

Usage

```
>>> from kinki.media.linkedin import API_KEY, API_SECRET, API_TOKEN, API_
↪TOKEN_SECRET
>>> from kinki.media.linkedin import KWARGS_API_SEARCH, KWARGS_API_FILTER
```

Dependencies

call: `media`, `location`, `quotation`

Contents

`media.linkedin.KWARGS_API_SEARCH`
Specific search terms: see https://developer.linkedin.com/documents/profile-fields, and see also the various API requests:

- **people: https://developer.linkedin.com/documents/people-search-api,** https://developer.linkedin.com/documents/profile-fields,
- connections: https://developer.linkedin.com/documents/connections-api,
- groups: https://developer.linkedin.com/documents/groups-api,
- companies: https://developer.linkedin.com/documents/company-lookup-api-and-fields,
- jobs: https://developer.linkedin.com/documents/job-search-api.

Note also the selectors field which defines the entities to be returned: https://developer.linkedin.com/documents/understanding-field-selectors

`media.linkedin.KWARGS_API_FILTER`
Specifc filter terms.

`media.linkedin.API_KEY = '773k3s7p0qd93x'`
API key.

`media.linkedin.API_SECRET = 'zK07rtZf7B186LeS'`
API secret key.

`media.linkedin.API_TOKEN = '77a20da9-2e4e-4c81-aad1-3f4518f76c3f'`
API token.

`media.linkedin.API_TOKEN_SECRET = 'c889f250-d725-4ae4-9f65-c9613970633f'`
API secret token.

`owm` Module

Global variables to store personal Open Weather Map API credentials.

Usage

```
>>> from kinki.media.owm import API_KEY
```

media/../_images/linkedin_api_keys.png

Fig. 4: *Screen capture of DEaP QooL credentials.*

Dependencies

call: `media`

Contents

media/../_images/owm_api_keys.png

Fig. 5: *Screen capture of DEaP QooL credentials.*

`media.owm.API_KEY = '05f413a3fcbc42d53523420311a997de'`
API key corresponding to the application ID.

`panoramio` Module

Global variables to store special terms to use the Panoramio API.

Usage

```
>>> from kinki.media.panoramio import KWARGS_API_SEARCH, KWARGS_API_FILTER
```

Dependencies

call: `media`, `status`

Contents

`media.panoramio.`**`KWARGS_API_SEARCH`**
Specific search terms: see http://www.panoramio.com/api/data/api.html.

`media.panoramio.`**`KWARGS_API_FILTER`**
Specifc filter terms.

`twitter` Module

Global variables to store personal Twitter API credentials.

Usage

```
>>> from kinki.media.twitter import API_KEY, API_SECRET, API_TOKEN, API_
↪TOKEN_SECRET
>>> from kinki.media.twitter import KWARGS_API_SEARCH, KWARGS_API_FILTER
```

Dependencies

call: `media`, `location`, `quotation`, `status`, `user`

Contents

`media.twitter.`**`KWARGS_API_SEARCH`**
Specific search terms: see https://dev.twitter.com/docs/api/1.1. See also the various API requests:

- users: https://dev.twitter.com/docs/api/1.1/get/users/search,
- tweets: https://dev.twitter.com/docs/api/1.1/get/search/tweets,
- geo: https://dev.twitter.com/docs/api/1.1/get/geo/search.

`media.twitter.`**`KWARGS_API_FILTER`**
Specifc filter terms.

`media.twitter.`**`API_KEY = 'Hi4prU7z2LgklagSF5f2QNk4M'`**
API key.

`media.twitter.`**`API_SECRET = 'FnJlCoswvDX056ppUB74pM6rSzAVeBcNjgsT3Nok1ZM14T38OU'`**
API secret key.

`media.twitter.`**`API_TOKEN = '2435142270-JDu2p9FNwwvnVeusaPQIsvvInNvGZ4HhCf0lQAi'`**
API token.

`media.twitter.`**`API_TOKEN_SECRET = 'LRdvfI2v22CuHRpvRY3q412L5ODQYb7rIEsym9k0pmBJy'`**
API secret token.

`yahoo` Module

Global variables to store personal Yahoo API credentials.

Usage

media/../_images/yelp_api_keys.png

Fig. 6: *Screen capture of DEaP QooL credentials.*

```
>>> from kinki.media.yahoo import API_KEY
```

Dependencies

call: `media`

Contents

media/../_images/yahoo_api_keys.png

Fig. 7: *Screen capture of DEaP QooL credentials.*

`media.yahoo.`**`API_KEY = 'PSXZnuvV34G3NKdiEdmh8Bqs.ywMKJ0AslsR4.lkhUWn7AvuYZaNIayvrS6oWUg`**

API key corresponding to the application ID.

fetch Package

Package for fetching data from online (social) media.

Usage

```
>>> from kinki.fetch.twitter import Twitter
>>> from kinki.fetch.foursquare import Foursquare
>>> from kinki.fetch.linkedin import Linkedin
>>> from kinki.fetch.facebook import Facebook
```

```
>>> from kinki.fetch.flickr import Flickr
>>> from kinki.fetch.instagram import Instagram
>>> from kinki.fetch.panoramio import Panoramio
```

Contents

facebook Module

About

credits: grazzja

version: 0.1 – *since*: Tue Apr 01 15:21:34 2014

Description

Dependencies

call: `media`, `base`, `facebook`, `imports`

require: `requests`, `urllib`, `urllib2`, `json/simplejson`

optional `facebook`, `pyfacebook`, `fbconsole`

Contents

class `fetch.facebook.`**`Facebook`**(***kwargs*)

Bases: `kinki.fetch.base.Base`

A client for the Facebook Graph API. See https://developers.facebook.com/docs/graph-api for complete documentation on the API.

Implement different APIs under a unique umbrella. Provide, through composition, an interface to most of the methods from classes in `['_Facebook']` which respectively reimplement the original wrappers `['facebook']` of the Facebook API.

Note: The Graph API is made up of the objects in Facebook (e.g., people, pages, events, photos) and the connections between them (e.g., friends, photo tags, and event RSVPs). This client provides access to those primitive types in a generic way.

See also https://developers.facebook.com/docs/graph-api/reference for a list of all of the objects and connections supported by the API. See http://developers.facebook.com/docs/authentication/ for details on obtaining an access token.

Initialisation of a *`Facebook`* instance: set all credentials and access rules related attributes.

```
>>> x = Facebook(**kwargs)
```

Keyword Arguments

- **key,secret,token,token_secret** (*str*) – keys used to set various credentials, e.g. key, secret, token, and token secret, usually required for connecting to an API.
- **api,request,acces,authorize,authenticate,redirect** (*str*) – URLs used for accessing the API.
- **read,write** (*str*) – access methods, e.g. `'GET'` and `'POST'` (default), for connection settings.

access_methods
Access mehods attribute – return (`getter`) the set of `read`/`write` methods of a *Facebook* instance for connecting to the media.

access_token
Access token attribute (`getter`/`setter`) of a *Facebook* instance used for connection.

access_token_secret
Secret access token attribute (`getter`/`setter`) of a *Facebook* instance used for connection.

access_urls
Access URLs attribute – return (`getter`) the dictionary of URLs of the media API.

api_application_defs
Return (`getter`) the main description attributes of the connecting application associated to a *Facebook* instance.

api_query_keys
Return (`getter`) the set of keys used to formulate a query to a *Facebook* instance of the media.

apis
APIs – return (`getter`) the list of connecting APIs for this media.

auth
Authorization key attribute (`getter`/`setter`) of a *Facebook* instance used for connection.

clear()
Clear the main API parameters of the current *Facebook* instance.

clearAPI()
Clear the media API clients of the current *Facebook* instance.

clearAuthentication()
Clear authentication of the current *Facebook* instance.

clearCredential()
Clear all credentials of the current *Facebook* instance.

clearKey()
Clear all keys of the current *Facebook* instance.

clearToken()
Clear all tokens of the current *Facebook* instance.

clearURL()
Clear media authenticating URLs of the current *Facebook* instance. Methods are left as it is.

client_id
Client ID attribute (getter/setter) of a *Facebook* instance used for connection.

client_key
Client key attribute (getter/setter) of a *Facebook* instance used for connection.

client_secret
Secret client key attribute (getter/setter) of a *Facebook* instance used for connection.

connections(*user_id*, *arg_conn*, ***kwargs*)
Fetch the connections for given object; override self.api.connections

```
>>> resp = x.connections(user_id, arg_conn, **kwargs)
```

credentials
Credentials attribute – return (getter) the list of credentials of a *Facebook* instance used for connecting to the media.

dance()
Do the oauth dance: get and set access token and token secret using internal keys.

```
>>> x.dance()
```

See also:

setToken(), *getToken()*

domain
Domain URL attribute – return (getter) the main URL domain of the media API.

friends(*_id*)
Return a cursored collection of user IDs for every friend of a given user.

```
>>> ids = x.friends(user_id, **kwargs)
```

get()
Get/retrieve the stored API parameters: credentials, authorization and apis (in this order).

```
>>> credentials, auth, apis = x.get()
```

See also:

credentials, *auth*, *apis*

getAPI(***kwargs*)
Get the API clients associated to available wrappers to the media using the credentials of the current *Facebook* instance for connecting.

```
>>> apis = x.getAPI(**kwargs)
```

Keyword Arguments

- **access_urls,access_methods** (*dict*) – access methods and URLs used to reset the access to the media; see *getToken()*.
- **all_apis** (*bool*) – optional flag set to True (default) when all APIs obtained by the different *Python* wrappers are returned in the output dictionary.

Returns **apis** (*dict*) – a dictionary of valid API wrapper(s) (indexed by the *Python* wrappers name in ['facebook']) for connecting to the media.

getAuthentication(***kwargs*)

Generic method returning the Authentication Authenticate using the credentials passed in args or those already stored in this instance.

```
>>> auth = x.getAuthentication(**kwargs)
```

Keyword Arguments

- **_id** (*str*) – optional client's ID.
- **access_urls** (dict) – dictionary of URLs access points; keys must be a subset of `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.

getCredential(***kwargs*)

Get/retrieve the media credentials of the current *Facebook* instance.

```
>>> credentials = x.getCredential()
```

See also:

getKey(), *getToken()*

getDefaultAPI(***kwargs*)

Get default API client using the credentials already stored in the current *Facebook* instance.

```
>>> api = x.getDefaultAPI(**kwargs)
```

Note: *DUMMY* copy/paste of *getAPI()* method.

getDefaultAuthentication(***kwargs*)

Get default authentication using the credentials already stored in the current *Facebook* instance.

```
>>> auth = x.getDefaultAuthentication(**kwargs)
```

Note: *DUMMY* copy/paste of *getAuthentication()* method.

getDefaultCredential(***kwargs*)

Load 'hard-stored' credentials from an external media configuration file.

```
>>> credentials = x.getDefaultCredential(**kwargs)
```

See also:

getDefaultKey(), *getDefaultToken()*

getDefaultKey(***kwargs*)

Load 'hard-stored' keys from the media specific file in kinki.media module.

```
>>> keys = x.getDefaultKey(**kwargs)
```

Keyword Arguments

- **package** (*str*) – string indicating a *subpackage* name (e.g. a string like *fetch*) or a *subpackage* path.
- **module** (*str*) – string indicating a python module name (e.g. a string like imports) or a json file basename where default parameter arguments are set.

See also:

kimports(), setDefault()

getDefaultToken(***kwargs*)

Load 'hard-stored' tokens from the media specific file in kinki.media module.

```
>>> tokens = x.getDefaultToken(**kwargs)
```

Keyword Arguments **module,package** – see *getDefaultKey()* method.

getDefaultURL(***kwargs*)

Get/retrieve media authenticating URLs and connecting (read/write) methods of the current *Facebook* instance.

```
>>> urls, methods = x.getDefaultURL(**kwargs)
```

Keyword Arguments

- **module,package** – see *getDefaultKey()* method.
- **oauth_version** (*int*) – oauth framework number; see *getToken()* method.
- **read,write** – access methods; see *getToken()* method.

Returns

- **urls** (*dict*) – a dictionary of URLs indexed by ['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api'].
- **methods** (*dict*) – a dictionary of access methods indexed by ['read', 'write'].

getKey(***kwargs*)

Get/retrieve the access keys stored in the current *Facebook* instance.

```
>>> keys = x.getKey(**kwargs)
```

getToken(***kwargs*)

Get/retrieve tokens and authorization using OAuth?Session request methods from requests_oauthlib/oauth2 modules.

```
>>> tokens = x.getToken(**kwargs)
```

Keyword Arguments

- **access_urls** (dict) – dictionary of URLs used when connecting the API for authorization.

- **access_methods** (dict) – access methods (e.g. 'GET' and 'POST') definition.
- **oauth_version** (*int*) – oauth framework number: 1 or 2; default is :literal:2‘.
- **_id** (*str*) – optional client ID.
- **permissions** (*list*) – optional set of permissions used for connecting specific endpoints.

Note:

- OAuth 1 is for user authenticated calls (e.g. posting, following people, sending DMs, etc.),
- OAuth 2 is for application authenticated calls (when you don't want to authenticate a user and make read-only calls).

---

getURL(***kwargs*)

Get/retrieve the access URLs and access methods of the current *Facebook* instance.

```
>>> URLS, methods = x.getURL()
```

keys

Keys attribute – return (getter) the dictionary of credentials used for connecting to the media. *keys* is indexed by the strings in ['key', 'secret', 'token', 'token_secret'].

loadToken(*token_file*)

Read a token file and set the access token and token secret.

```
>>> x.loadToken(token_file)
```

Parameters **token_file** (*str*) – byte string file with tokens stored (one per line).

See also:

setToken()

media

Media attribute – return (getter) the media name.

medianame

Attribute used for convenience – see *media*.

mutual_friends(*_id*)

Construct a graph of mutual friendships with a given user.

```
>>> ids = x.mutual_friends(user_id)
```

objects(*user_id*, ***args*)

Fetch all of the given object from the graph.

```
>>> resp = x.objects(user_id, **kwargs)
```

permissions
Permission attribute – return (`getter`) the set of permissions granted when connecting to the media.

query(**arg*, ***kwargs*)
Define a regular query specific to the media, i.e. depending on the syntax used by the API: the syntax refers to the set of key terms used to query the API (e.g. retrieved using *api_query_keys()* property).

```
>>> key, query = x.query(*arg, **kwargs)
```

Parameters **arg** (dict, *query.api.API*, *query.api.Query*) –

arguments used by the wrapper to query the media; it is either:

- a generic query with generic terms, i.e. any key in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']` used by a `Status` instance, which are then 'translated/converted' to the specific syntax of the media (cases :class'dict' or *query.api.API*), or
- an already well-formed' query arguments with appropriate syntax (case *query.api.Query*);

possibly `()` or `(None,)` when parameters are passed through keyword arguments in `kwargs` (see below).

Keyword Arguments

- **Search** (`dict`) – a dictionary of already 'well-formed' search arguments; in practice, overrides the input `arg` parameter.
- **kwargs** (`dict`) – additional generic query arguments.

See also:

`get()`, *api_query_keys()*.

request(*path*, ***kwargs*)
Fetch the given path in the Graph API; override api.request;

```
>>> resp = x.request(path, **kwargs)
```

Note: Also support FQL queries.

saveToken(*token_file*, **tokens*)
Write a token file to hold the oauth access token and token secret.

```
>>> x.saveToken(*tokens)
```

Parameters

- **token_file** (*str*) – byte string file where to store tokens.
- **tokens** (*list of str*) – Access token and secret access token.

search (*page*)

set (***kwargs*)

Set/update the API parameters of a {FETCH} instance by setting all the attributes used for connection to the media when they have not been set. This is done consistently with internal attributes that have been already set (e.g. in __init__() or through default setting) or with the other parameters passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments

- **kwargs** (dict) – dictionary of arguments used to set the API parameters; its keys should be a any of those already accepted by *getCredential()*, *getURL()*, *getAuthentication()* and *getAPI()*. In the case it is empty (i.e. {}, default settings are chosen).
- **package,module** (*str*) –

  in the case default settings are selected, strings indicating resp.:

  - a *subpackage* name (e.g. a string like *fetch*) or a

    subpackage path,

  - a python module name (e.g. a string like imports) or a

    json file basename where default parameter arguments are set.

Note: The *set()* method is essentialy running different setting methods, namely *setCredential()*, *setURL()*, *setAuthentication()* and *setAPI()* methods.

See also:

kimports()

setAPI (**apis, **kwargs*)

Set the media API clients.

```
>>> x.setAPI(*apis, **kwargs)
```

Parameters **apis** (*tuple of dict*) – dictionary of API wrapper(s) to store in this instance; possibly () or (None,), then the APIs are set to the clients associated to the tokens in the current instance as returned by the *Python* wrappers in ['facebook'].

Keyword Arguments **kwargs** (dict) – see *getAPI()* method.

setAuthentication (**auth, **kwargs*)

```
>>> x.setAuthentication(*auth, **kwargs)
```

Parameters **auth** (*str*) – authorization key represented as a string; posssibly `()` or `(None,)`, then the key is set through keyword argument auth in `kwargs` (see below), or is derived from the access tokens stored in this instance.

Keyword Arguments **auth** (*str*) – other way of passing the authorization key; ignored when *auth* is passed.

setCredential(**credentials*, ***kwargs*)

Set/update the media credentials (access keys and tokens) of the current *Facebook* instance.

```
>>> x.setCredential(*credentials, **kwargs)
```

Parameters **credentials** (*tuple of str*) – credentials; whenever it is not `()` or , `(None,)` most keyword arguments in `kwargs` (see below) may be ignored.

Keyword Arguments

- **key,secret** – access keys; see *setKey()*.
- **token,token_secret** – access tokens; see *setToken()*.
- **_id** (*str*) – optional client's ID.

setKey(**keys*, ***kwargs*)

Set/update the access keys of the current *Facebook* instance.

```
>>> x.setKey(*keys, **kwargs)
```

Parameters **keys** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **key,secret** (*str*) – client's access key and secret used for connection.

setToken(**tokens*, ***kwargs*)

Set/update the access tokens of the current *Facebook* instance.

```
>>> x.setToken(*tokens, **kwargs)
```

Parameters **tokens** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **token,token_secret** (*str*) – client's access token and token secret used for connection.

setURL(**urls*, ***kwargs*)

Set media authenticating URLs and connecting (`'GET'/'POST'`) methods.

```
>>> x.setURL(*urls, **kwargs)
```

Parameters **urls** (*dict*) – dict of access URLs (indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`) and access methods (indexed

by `['read', 'write']`); whenever it is not `()` or `(None,)` all keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments

- **`api`** (*str*) – Main URL domain of the API.
- **`request,acces,authorize,authenticate,redirect`** (*str*) – URLs with respective 'request', 'acces', 'authorize', 'authenticate', and 'redirect' endpoints to the media.
- **`access_methods`** (`dict`) – dictionary (indexed by `['read', 'write']`) of access methods, e.g. `'GET'` and `'POST'` (default).

`stream`(**args*, ***kwargs*)

`url_fql`

`wrappers`

APIs wrappers – return (`getter`) the list of python wrappers for the different connecting APIs.

class `fetch.facebook.`**`_Facebook`**

Facebook API using *Python* `facebook` module. See http://www.pythonforfacebook.com.

Note: See http://facebook-python-library.docs-library.appspot.com/facebook-python/examples/oauth.html

---

class `fetch.facebook.`**`_Pyfacebook`**

`flickr` Module

About

credits: grazzja

version: 0.1 – *since*: Tue Apr 01 15:09:34 2014

Description

Dependencies

call: `media`, `base`, `flickr`, `imports`

require: `sys`, `re`

optional `flickrapi`, `flickr_api`

Contents

class `fetch.flickr.`**`Flickr`**(**args*, ***kwargs*)

Bases: `kinki.fetch.base.Base`

`getLocationsFromPhotos`(*photos*)

`getPhotosByPage`(*userid*, *page_nr*)

`getUserInfo`(*userId*)

Retrieves a target's username, real name and location as provided by flickr API

Returns a target dictionary

returnLocations(*target*, *search_params*)

searchForTargets(*search_term*)

foursquare Module

About

credits: grazzja

version: 0.1 – *since*: Wed Apr 09 16:00:59 2014

Description

Dependencies

call: `media`, `base`, `imports`

require: `webbrowser`

optional `pyfoursquare`, `foursquare`

Contents

class `fetch.foursquare.`**Foursquare**(***kwargs*)

Bases: `kinki.fetch.base.Base`

A client for the Foursquare API. See https://developer.foursquare.com for complete documentation on the API.

Provide, through composition, an interface to most of the methods from classes `['_Foursquare', '_Pyfoursquare']` which respectively reimplement the original wrappers `['foursquare', 'pyfoursquare']` of the Foursquare API.

Note: See also API endpoints: https://developer.foursquare.com/docs/.

Initialisation of a *Foursquare* instance: set all credentials and access rules related attributes.

```
>>> x = Foursquare(**kwargs)
```

Keyword Arguments

- **key,secret,token,token_secret** (*str*) – keys used to set various credentials, e.g. key, secret, token, and token secret, usually required for connecting to an API.
- **api,request,acces,authorize,authenticate,redirect** (*str*) – URLs used for accessing the API.
- **read,write** (*str*) – access methods, e.g. `'GET'` and `'POST'` (default), for connection settings.

access_methods

Access mehods attribute – return (`getter`) the set of `read`/`write` methods of a *Foursquare* instance for connecting to the media.

access_token

Access token attribute (`getter`/`setter`) of a *Foursquare* instance used for connection.

access_token_secret
Secret access token attribute (getter/setter) of a *Foursquare* instance used for connection.

access_urls
Access URLs attribute – return (getter) the dictionary of URLs of the media API.

api_application_defs
Return (getter) the main description attributes of the connecting application associated to a *Foursquare* instance.

api_query_keys
Return (getter) the set of keys used to formulate a query to a *Foursquare* instance of the media.

apis
APIs – return (getter) the list of connecting APIs for this media.

auth
Authorization key attribute (getter/setter) of a *Foursquare* instance used for connection.

clear()
Clear the main API parameters of the current *Foursquare* instance.

clearAPI()
Clear the media API clients of the current *Foursquare* instance.

clearAuthentication()
Clear authentication of the current *Foursquare* instance.

clearCredential()
Clear all credentials of the current *Foursquare* instance.

clearKey()
Clear all keys of the current *Foursquare* instance.

clearToken()
Clear all tokens of the current *Foursquare* instance.

clearURL()
Clear media authenticating URLs of the current *Foursquare* instance. Methods are left as it is.

client_id
Client ID attribute (getter/setter) of a *Foursquare* instance used for connection.

client_key
Client key attribute (getter/setter) of a *Foursquare* instance used for connection.

client_secret
Secret client key attribute (getter/setter) of a *Foursquare* instance used for connection.

credentials
Credentials attribute – return (getter) the list of credentials of a *Foursquare* instance used for connecting to the media.

dance()
Do the oauth dance: get and set access token and token secret using internal keys.

```
>>> x.dance()
```

See also:

setToken(), *getToken()*

domain

Domain URL attribute – return (`getter`) the main URL domain of the media API.

get()

Get/retrieve the stored API parameters: credentials, authorization and apis (in this order).

```
>>> credentials, auth, apis = x.get()
```

See also:

credentials, *auth*, *apis*

getAPI(***kwargs*)

Get the API clients associated to available wrappers to the media using the credentials of the current *Foursquare* instance for connecting.

```
>>> apis = x.getAPI(**kwargs)
```

Keyword Arguments

- **access_urls,access_methods** (*dict*) – access methods and URLs used to reset the access to the media; see *getToken()*.
- **all_apis** (*bool*) – optional flag set to True (default) when all APIs obtained by the different *Python* wrappers are returned in the output dictionary.

Returns **apis** (*dict*) – a dictionary of valid API wrapper(s) (indexed by the *Python* wrappers name in `['foursquare', 'pyfoursquare']`) for connecting to the media.

getAuthentication(***kwargs*)

Generic method returning the Authentication Authenticate using the credentials passed in args or those already stored in this instance.

```
>>> auth = x.getAuthentication(**kwargs)
```

Keyword Arguments

- **_id** (*str*) – optional client's ID.
- **access_urls** (*dict*) – dictionary of URLs access points; keys must be a subset of `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.

getCredential(***kwargs*)

Get/retrieve the media credentials of the current *Foursquare* instance.

```
>>> credentials = x.getCredential()
```

See also:

getKey(), *getToken()*

getDefaultAPI(***kwargs*)

Get default API client using the credentials already stored in the current `Foursquare` instance.

```
>>> api = x.getDefaultAPI(**kwargs)
```

Note: *DUMMY* copy/paste of `getAPI()` method.

getDefaultAuthentication(***kwargs*)

Get default authentication using the credentials already stored in the current `Foursquare` instance.

```
>>> auth = x.getDefaultAuthentication(**kwargs)
```

Note: *DUMMY* copy/paste of `getAuthentication()` method.

getDefaultCredential(***kwargs*)

Load 'hard-stored' credentials from an external media configuration file.

```
>>> credentials = x.getDefaultCredential(**kwargs)
```

See also:

`getDefaultKey()`, `getDefaultToken()`

getDefaultKey(***kwargs*)

Load 'hard-stored' keys from the media specific file in kinki.media module.

```
>>> keys = x.getDefaultKey(**kwargs)
```

Keyword Arguments

- **package** (`str`) – string indicating a *subpackage* name (e.g. a string like `fetch`) or a *subpackage* path.
- **module** (`str`) – string indicating a python module name (e.g. a string like `imports`) or a json file basename where default parameter arguments are set.

See also:

`kimports()`, `setDefault()`

getDefaultToken(***kwargs*)

Load 'hard-stored' tokens from the media specific file in kinki.media module.

```
>>> tokens = x.getDefaultToken(**kwargs)
```

Keyword Arguments **module,package** – see `getDefaultKey()` method.

getDefaultURL(***kwargs*)

Get/retrieve media authenticating URLs and connecting (read/write) methods of the current `Foursquare` instance.

```
>>> urls, methods = x.getDefaultURL(**kwargs)
```

Keyword Arguments

- **module,package** – see *getDefaultKey()* method.
- **oauth_version** (*int*) – oauth framework number; see *getToken()* method.
- **read,write** – access methods; see *getToken()* method.

Returns

- **urls** (*dict*) – a dictionary of URLs indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.
- **methods** (*dict*) – a dictionary of access methods indexed by `['read', 'write']`.

getKey (***kwargs*)

Get/retrieve the access keys stored in the current *Foursquare* instance.

```
>>> keys = x.getKey(**kwargs)
```

getToken (***kwargs*)

Get/retrieve tokens and authorization using OAuth?Session request methods from `requests_oauthlib`/`oauth2` modules.

```
>>> tokens = x.getToken(**kwargs)
```

Keyword Arguments

- **access_urls** (`dict`) – dictionary of URLs used when connecting the API for authorization.
- **access_methods** (`dict`) – access methods (e.g. `'GET'` and `'POST'`) definition.
- **oauth_version** (*int*) – oauth framework number: 1 or 2; default is :literal:2‘.
- **_id** (*str*) – optional client ID.
- **permissions** (*list*) – optional set of permissions used for connecting specific endpoints.

Note:

- OAuth 1 is for user authenticated calls (e.g. posting, following people, sending DMs, etc.),
- OAuth 2 is for application authenticated calls (when you don't want to authenticate a user and make read-only calls).

---

getURL (***kwargs*)

Get/retrieve the access URLs and access methods of the current *Foursquare* instance.

```
>>> URLS, methods = x.getURL()
```

keys

Keys attribute – return (`getter`) the dictionary of credentials used for connecting to the media. `keys` is indexed by the strings in `['key', 'secret', 'token', 'token_secret']`.

loadToken(*token_file*)

Read a token file and set the access token and token secret.

```
>>> x.loadToken(token_file)
```

Parameters **token_file** (*str*) – byte string file with tokens stored (one per line).

See also:

`setToken()`

media

Media attribute – return (`getter`) the media name.

medianame

Attribute used for convenience – see `media`.

permissions

Permission attribute – return (`getter`) the set of permissions granted when connecting to the media.

query(**arg*, ***kwargs*)

Define a regular query specific to the media, i.e. depending on the syntax used by the API: the syntax refers to the set of key terms used to query the API (e.g. retrieved using `api_query_keys()` property).

```
>>> key, query = x.query(*arg, **kwargs)
```

Parameters **arg** (dict, `query.api.API`, `query.api.Query`) –

arguments used by the wrapper to query the media; it is either:

- a generic query with generic terms, i.e. any key in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']` used by a `Status` instance, which are then 'translated/converted' to the specific syntax of the media (cases :class'dict' or `query.api.API`), or
- an already well-formed' query arguments with appropriate syntax (case `query.api.Query`);

possibly `()` or `(None,)` when parameters are passed through keyword arguments in `kwargs` (see below).

Keyword Arguments

- **Search** (dict) – a dictionary of already 'well-formed' search arguments; in practice, overrides the input arg parameter.
- **kwargs** (dict) – additional generic query arguments.

See also:

get(), *api_query_keys()*.

saveToken(*token_file*, **tokens*)

Write a token file to hold the oauth access token and token secret.

```
>>> x.saveToken(*tokens)
```

Parameters

- **token_file** (*str*) – byte string file where to store tokens.
- **tokens** (*list of str*) – Access token and secret access token.

search(**args*, ***kwargs*)

Search using the media API for Foursquare.

```
>>> res = x.search(*args, **kwargs)
```

Parameters **args,kwargs** – arguments used by the API wrapper to search the media.

See also:

getAPI()

set(***kwargs*)

Set/update the API parameters of a {FETCH} instance by setting all the attributes used for connection to the media when they have not been set. This is done consistently with internal attributes that have been already set (e.g. in __init__() or through default setting) or with the other parameters passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments

- **kwargs** (dict) – dictionary of arguments used to set the API parameters; its keys should be a any of those already accepted by *getCredential()*, *getURL()*, *getAuthentication()* and *getAPI()*. In the case it is empty (i.e. {}, default settings are chosen).
- **package,module** (*str*) –

  in the case default settings are selected, strings indicating resp.:

  – a *subpackage* name (e.g. a string like *fetch*) or a

  subpackage path,

  – a python module name (e.g. a string like imports) or a

  json file basename where default parameter arguments are set.

Note: The `set()` method is essentialy running different setting methods, namely `setCredential()`, `setURL()`, `setAuthentication()` and `setAPI()` methods.

See also:

`kimports()`

setAPI(**apis, **kwargs*)

Set the media API clients.

```
>>> x.setAPI(*apis, **kwargs)
```

Parameters **apis** (*tuple of dict*) – dictionary of API wrapper(s) to store in this instance; possibly `()` or `(None,)`, then the APIs are set to the clients associated to the tokens in the current instance as returned by the *Python* wrappers in `['foursquare', 'pyfoursquare']`.

Keyword Arguments **kwargs** (`dict`) – see `getAPI()` method.

setAuthentication(**auth, **kwargs*)

```
>>> x.setAuthentication(*auth, **kwargs)
```

Parameters **auth** (*str*) – authorization key represented as a string; posssibly `()` or `(None,)`, then the key is set through keyword argument auth in `kwargs` (see below), or is derived from the access tokens stored in this instance.

Keyword Arguments **auth** (*str*) – other way of passing the authorization key; ignored when `auth` is passed.

setCredential(**credentials, **kwargs*)

Set/update the media credentials (access keys and tokens) of the current `Foursquare` instance.

```
>>> x.setCredential(*credentials, **kwargs)
```

Parameters **credentials** (*tuple of str*) – credentials; whenever it is not `()` or , `(None,)` most keyword arguments in `kwargs` (see below) may be ignored.

Keyword Arguments

- **key,secret** – access keys; see `setKey()`.
- **token,token_secret** – access tokens; see `setToken()`.
- **_id** (*str*) – optional client's ID.

setKey(**keys, **kwargs*)

Set/update the access keys of the current `Foursquare` instance.

```
>>> x.setKey(*keys, **kwargs)
```

Parameters **keys** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **key,secret** (*str*) – client's access key and secret used for connection.

setToken(**tokens*, ***kwargs*)

Set/update the access tokens of the current *Foursquare* instance.

```
>>> x.setToken(*tokens, **kwargs)
```

Parameters **tokens** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **token,token_secret** (*str*) – client's access token and token secret used for connection.

setURL(**urls*, ***kwargs*)

Set media authenticating URLs and connecting (`'GET'`/`'POST'`) methods.

```
>>> x.setURL(*urls, **kwargs)
```

Parameters **urls** (*dict*) – dict of access URLs (indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`) and access methods (indexed by `['read', 'write']`); whenever it is not `()` or `(None,)` all keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments

- **api** (*str*) – Main URL domain of the API.
- **request,acces,authorize,authenticate,redirect** (*str*) – URLs with respective 'request', 'acces', 'authorize', 'authenticate', and 'redirect' endpoints to the media.
- **access_methods** (dict) – dictionary (indexed by `['read', 'write']`) of access methods, e.g. `'GET'` and `'POST'` (default).

stream(**args*, ***kwargs*)

wrappers

APIs wrappers – return (`getter`) the list of python wrappers for the different connecting APIs.

class fetch.foursquare.**_Foursquare**

Foursquare API using *Python* `foursquare` module. See https://github.com/mLewisLogic/foursquare.

Note: See `foursquare` Readme for a full list of endpoint and methods, in particular `'venues'`, `'venues_tips'`, `'tips'`, `'users'`, `'user_friends'`, or `'user_checkins'`.

---

class fetch.foursquare.**_Pyfoursquare**

Linkedin API using *Python* `pyfoursquare` module. See https://github.com/marcelcaraciolo/foursquare.

Note: See `foursquare` Readme for a full list of endpoint and methods, in particular `'venues'`, `'venues_tips'`, `'tips'`, `'users'`, `'user_friends'`, or `'user_checkins'`.

`linkedin` Module

About

credits: grazzja

version: 0.1 – *since*: Tue Apr 01 15:09:34 2014

Description

Dependencies

call: `media`, `base`, `linkedin`, `base`, `status`, `location`, `imports`, `decorators`

require: `linkedin/pylinkedin`

Contents

class `fetch.linkedin.`**`Linkedin`**(***kwargs*)

Bases: `kinki.fetch.base.Base`

A client for the LinkedIn API. See http://developer.linkedin.com for complete documentation on the API.

Provide, through composition, an interface to most of the methods from classes `[]` which respectively reimplement the original wrappers `[]` of the Linkedin API.

Note: See https://developer.linkedin.com/apis

Initialisation of a *`Linkedin`* instance: set all credentials and access rules related attributes.

```
>>> x = Linkedin(**kwargs)
```

Keyword Arguments

- **`key,secret,token,token_secret`** (*`str`*) – keys used to set various credentials, e.g. key, secret, token, and token secret, usually required for connecting to an API.
- **`api,request,acces,authorize,authenticate,redirect`** (*`str`*) – URLs used for accessing the API.
- **`read,write`** (*`str`*) – access methods, e.g. `'GET'` and `'POST'` (default), for connection settings.

`access_methods`

Access mehods attribute – return (`getter`) the set of `read`/`write` methods of a *`Linkedin`* instance for connecting to the media.

`access_token`

Access token attribute (`getter`/`setter`) of a *`Linkedin`* instance used for connection.

access_token_secret
Secret access token attribute (`getter`/`setter`) of a `Linkedin` instance used for connection.

access_urls
Access URLs attribute – return (`getter`) the dictionary of URLs of the media API.

api_application_defs
Return (`getter`) the main description attributes of the connecting application associated to a `Linkedin` instance.

api_query_keys
Return (`getter`) the set of keys used to formulate a query to a `Linkedin` instance of the media.

apis
APIs – return (`getter`) the list of connecting APIs for this media.

auth
Authorization key attribute (`getter`/`setter`) of a `Linkedin` instance used for connection.

clear()
Clear the main API parameters of the current `Linkedin` instance.

clearAPI()
Clear the media API clients of the current `Linkedin` instance.

clearAuthentication()
Clear authentication of the current `Linkedin` instance.

clearCredential()
Clear all credentials of the current `Linkedin` instance.

clearKey()
Clear all keys of the current `Linkedin` instance.

clearToken()
Clear all tokens of the current `Linkedin` instance.

clearURL()
Clear media authenticating URLs of the current `Linkedin` instance. Methods are left as it is.

client_id
Client ID attribute (`getter`/`setter`) of a `Linkedin` instance used for connection.

client_key
Client key attribute (`getter`/`setter`) of a `Linkedin` instance used for connection.

client_secret
Secret client key attribute (`getter`/`setter`) of a `Linkedin` instance used for connection.

credentials
Credentials attribute – return (`getter`) the list of credentials of a `Linkedin` instance used for connecting to the media.

dance()
Do the oauth dance: get and set access token and token secret using internal keys.

```
>>> x.dance()
```

See also:

setToken(), *getToken()*

domain

Domain URL attribute – return (`getter`) the main URL domain of the media API.

get()

Get/retrieve the stored API parameters: credentials, authorization and apis (in this order).

```
>>> credentials, auth, apis = x.get()
```

See also:

credentials, *auth*, *apis*

getAPI(***kwargs*)

Get the API clients associated to available wrappers to the media using the credentials of the current *Linkedin* instance for connecting.

```
>>> apis = x.getAPI(**kwargs)
```

Keyword Arguments

- **access_urls,access_methods** (*dict*) – access methods and URLs used to reset the access to the media; see *getToken()*.
- **all_apis** (*bool*) – optional flag set to True (default) when all APIs obtained by the different *Python* wrappers are returned in the output dictionary.

Returns **apis** (*dict*) – a dictionary of valid API wrapper(s) (indexed by the *Python* wrappers name in `[]`) for connecting to the media.

getAuthentication(***kwargs*)

Generic method returning the Authentication Authenticate using the credentials passed in args or those already stored in this instance.

```
>>> auth = x.getAuthentication(**kwargs)
```

Keyword Arguments

- **_id** (*str*) – optional client's ID.
- **access_urls** (*dict*) – dictionary of URLs access points; keys must be a subset of `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.

getCredential(***kwargs*)

Get/retrieve the media credentials of the current *Linkedin* instance.

```
>>> credentials = x.getCredential()
```

See also:

getKey(), *getToken()*

getDefaultAPI(***kwargs*)

Get default API client using the credentials already stored in the current *Linkedin* instance.

```
>>> api = x.getDefaultAPI(**kwargs)
```

Note: *DUMMY* copy/paste of `getAPI()` method.

getDefaultAuthentication(***kwargs*)

Get default authentication using the credentials already stored in the current `Linkedin` instance.

```
>>> auth = x.getDefaultAuthentication(**kwargs)
```

Note: *DUMMY* copy/paste of `getAuthentication()` method.

getDefaultCredential(***kwargs*)

Load 'hard-stored' credentials from an external media configuration file.

```
>>> credentials = x.getDefaultCredential(**kwargs)
```

See also:

`getDefaultKey()`, `getDefaultToken()`

getDefaultKey(***kwargs*)

Load 'hard-stored' keys from the media specific file in kinki.media module.

```
>>> keys = x.getDefaultKey(**kwargs)
```

Keyword Arguments

- **package** (*str*) – string indicating a *subpackage* name (e.g. a string like `fetch`) or a *subpackage* path.
- **module** (*str*) – string indicating a python module name (e.g. a string like `imports`) or a json file basename where default parameter arguments are set.

See also:

`kimports()`, `setDefault()`

getDefaultToken(***kwargs*)

Load 'hard-stored' tokens from the media specific file in kinki.media module.

```
>>> tokens = x.getDefaultToken(**kwargs)
```

Keyword Arguments **module,package** – see `getDefaultKey()` method.

getDefaultURL(***kwargs*)

Get/retrieve media authenticating URLs and connecting (read/write) methods of the current `Linkedin` instance.

```
>>> urls, methods = x.getDefaultURL(**kwargs)
```

Keyword Arguments

- **module,package** – see `getDefaultKey()` method.
- **oauth_version** (*int*) – oauth framework number; see `getToken()` method.
- **read,write** – access methods; see `getToken()` method.

Returns

- **urls** (*dict*) – a dictionary of URLs indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.
- **methods** (*dict*) – a dictionary of access methods indexed by `['read', 'write']`.

getKey(***kwargs*)

Get/retrieve the access keys stored in the current `Linkedin` instance.

```
>>> keys = x.getKey(**kwargs)
```

getToken(***kwargs*)

Get/retrieve tokens and authorization using OAuth?Session request methods from `requests_oauthlib/oauth2` modules.

```
>>> tokens = x.getToken(**kwargs)
```

Keyword Arguments

- **access_urls** (`dict`) – dictionary of URLs used when connecting the API for authorization.
- **access_methods** (`dict`) – access methods (e.g. `'GET'` and `'POST'`) definition.
- **oauth_version** (*int*) – oauth framework number: 1 or 2; default is :literal:2‘.
- **_id** (*str*) – optional client ID.
- **permissions** (*list*) – optional set of permissions used for connecting specific endpoints.

Note:

- OAuth 1 is for user authenticated calls (e.g. posting, following people, sending DMs, etc.),
- OAuth 2 is for application authenticated calls (when you don’t want to authenticate a user and make read-only calls).

---

getURL(***kwargs*)

Get/retrieve the access URLs and access methods of the current `Linkedin` instance.

```
>>> URLS, methods = x.getURL()
```

keys

Keys attribute – return (`getter`) the dictionary of credentials used for connecting to the media. `keys` is indexed by the strings in `['key', 'secret', 'token', 'token_secret']`.

loadToken(*token_file*)

Read a token file and set the access token and token secret.

```
>>> x.loadToken(token_file)
```

Parameters **token_file** (*str*) – byte string file with tokens stored (one per line).

See also:

`setToken()`

media

Media attribute – return (`getter`) the media name.

medianame

Attribute used for convenience – see `media`.

permissions

Permission attribute – return (`getter`) the set of permissions granted when connecting to the media.

query(**arg*, ***kwargs*)

Define a regular query specific to the media, i.e. depending on the syntax used by the API: the syntax refers to the set of key terms used to query the API (e.g. retrieved using `api_query_keys()` property).

```
>>> key, query = x.query(*arg, **kwargs)
```

Parameters **arg** (dict, `query.api.API`, `query.api.Query`) –

arguments used by the wrapper to query the media; it is either:

- a generic query with generic terms, i.e. any key in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']` used by a `Status` instance, which are then 'translated/converted' to the specific syntax of the media (cases :class'dict' or `query.api.API`), or
- an already well-formed' query arguments with appropriate syntax (case `query.api.Query`);

possibly `()` or `(None,)` when parameters are passed through keyword arguments in `kwargs` (see below).

Keyword Arguments

- **Search** (dict) – a dictionary of already 'well-formed' search arguments; in practice, overrides the input arg parameter.
- **kwargs** (dict) – additional generic query arguments.

See also:

get(), *api_query_keys()*.

saveToken(*token_file*, **tokens*)

Write a token file to hold the oauth access token and token secret.

```
>>> x.saveToken(*tokens)
```

Parameters

- **token_file** (*str*) – byte string file where to store tokens.
- **tokens** (*list of str*) – Access token and secret access token.

search(**args*, ***kwargs*)

Search using the media API for Linkedin.

```
>>> res = x.search(*args, **kwargs)
```

Parameters **args,kwargs** – arguments used by the API wrapper to search the media.

See also:

getAPI()

set(***kwargs*)

Set/update the API parameters of a {FETCH} instance by setting all the attributes used for connection to the media when they have not been set. This is done consistently with internal attributes that have been already set (e.g. in __init__() or through default setting) or with the other parameters passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments

- **kwargs** (dict) – dictionary of arguments used to set the API parameters; its keys should be a any of those already accepted by *getCredential()*, *getURL()*, *getAuthentication()* and *getAPI()*. In the case it is empty (i.e. {}, default settings are chosen).
- **package,module** (*str*) –

  in the case default settings are selected, strings indicating resp.:

  – a *subpackage* name (e.g. a string like *fetch*) or a

  subpackage path,

  – a python module name (e.g. a string like imports) or a

  json file basename where default parameter arguments are set.

Note: The `set()` method is essentialy running different setting methods, namely `setCredential()`, `setURL()`, `setAuthentication()` and `setAPI()` methods.

See also:

`kimports()`

setAPI(**apis, **kwargs*)

Set the media API clients.

```
>>> x.setAPI(*apis, **kwargs)
```

Parameters **apis** (*tuple of dict*) – dictionary of API wrapper(s) to store in this instance; possibly `()` or `(None,)`, then the APIs are set to the clients associated to the tokens in the current instance as returned by the *Python* wrappers in `[]`.

Keyword Arguments **kwargs** (`dict`) – see `getAPI()` method.

setAuthentication(**auth, **kwargs*)

```
>>> x.setAuthentication(*auth, **kwargs)
```

Parameters **auth** (*str*) – authorization key represented as a string; posssibly `()` or `(None,)`, then the key is set through keyword argument auth in `kwargs` (see below), or is derived from the access tokens stored in this instance.

Keyword Arguments **auth** (*str*) – other way of passing the authorization key; ignored when `auth` is passed.

setCredential(**credentials, **kwargs*)

Set/update the media credentials (access keys and tokens) of the current `Linkedin` instance.

```
>>> x.setCredential(*credentials, **kwargs)
```

Parameters **credentials** (*tuple of str*) – credentials; whenever it is not `()` or , `(None,)` most keyword arguments in `kwargs` (see below) may be ignored.

Keyword Arguments

- **key,secret** – access keys; see `setKey()`.
- **token,token_secret** – access tokens; see `setToken()`.
- **_id** (*str*) – optional client's ID.

setKey(**keys, **kwargs*)

Set/update the access keys of the current `Linkedin` instance.

```
>>> x.setKey(*keys, **kwargs)
```

Parameters **keys** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **key,secret** (*str*) – client's access key and secret used for connection.

setToken(**tokens*, ***kwargs*)

Set/update the access tokens of the current `Linkedin` instance.

```
>>> x.setToken(*tokens, **kwargs)
```

Parameters **tokens** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **token,token_secret** (*str*) – client's access token and token secret used for connection.

setURL(**urls*, ***kwargs*)

Set media authenticating URLs and connecting (`'GET'/'POST'`) methods.

```
>>> x.setURL(*urls, **kwargs)
```

Parameters **urls** (*dict*) – dict of access URLs (indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`) and access methods (indexed by `['read', 'write']`); whenever it is not `()` or `(None,)` all keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments

- **api** (*str*) – Main URL domain of the API.
- **request,acces,authorize,authenticate,redirect** (*str*) – URLs with respective 'request', 'acces', 'authorize', 'authenticate', and 'redirect' endpoints to the media.
- **access_methods** (`dict`) – dictionary (indexed by `['read', 'write']`) of access methods, e.g. `'GET'` and `'POST'` (default).

static status(*link*)

stream(**args*, ***kwargs*)

wrappers

APIs wrappers – return (`getter`) the list of python wrappers for the different connecting APIs.

class `fetch.linkedin.`**_Linkedin**

class `fetch.linkedin.`**_Pylinkedin**

`panoramio` Module

About

credits: grazzja

version: 0.1 – *since*: Wed Apr 09 23:29:09s 2014

Description

Dependencies

call: `base`, `media`, `panoramio`, `panoramioapi`, `imports`

Contents

class `fetch.panoramio.`**`Panoramio`**`(`***dummy*`)`

Bases: `kinki.fetch.base.Base`

A client for the Panoramio API using standard API. See http://www.panoramio.com/api/widget/api.html.

Provide, through composition, an interface to most of the methods from classes `['_Panoramioapi']` which respectively reimplement the original wrappers `['PanoramioAPI']` of the Panoramio API.

Note: The Panoramio API does no require any token/key.

Initialisation of a `Panoramio` instance.

```
>>> x = Panoramio()
```

`api_application_defs`

Return (`getter`) the main description attributes of the connecting application associated to a `Panoramio` instance.

`api_query_keys`

Return (`getter`) the set of keys used to formulate a query to a `Panoramio` instance of the media.

`apis`

APIs – return (`getter`) the list of connecting APIs for this media.

`clearAPI``()`

Clear the media API clients of the current `Panoramio` instance.

`client_id`

Client ID attribute (`getter`/`setter`) of a `Panoramio` instance used for connection.

`domain`

Domain URL attribute – return (`getter`) the main URL domain of the media API.

`getAPI``(`***dummy*`)`

Get the API clients associated to available wrappers to the media using the credentials of the current `Panoramio` instance for connecting.

```
>>> apis = x.getAPI()
```

Returns **apis** (*dict*) – a dictionary of valid API wrapper(s) (indexed by the *Python* wrapper name) for connecting to the media.

`getDefaultAPI``(`***kwargs*`)`

Get default API client using the credentials already stored in the current `Panoramio` instance.

```
>>> api = x.getDefaultAPI(**kwargs)
```

Note: *DUMMY* copy/paste of `getAPI()` method.

media
Media attribute – return (`getter`) the media name.

medianame
Attribute used for convenience – see `media`.

permissions
Permission attribute – return (`getter`) the set of permissions granted when connecting to the media.

query(**arg*, ***kwargs*)
Define a regular query specific to the media, i.e. depending on the syntax used by the API: the syntax refers to the set of key terms used to query the API (e.g. retrieved using `api_query_keys()` property).

```
>>> key, query = x.query(*arg, **kwargs)
```

Parameters **arg** (dict, `query.api.API`, `query.api.Query`) –

arguments used by the wrapper to query the media; it is either:

- a generic query with generic terms, i.e. any key in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']` used by a `Status` instance, which are then 'translated/converted' to the specific syntax of the media (cases :class'dict' or `query.api.API`), or
- an already well-formed' query arguments with appropriate syntax (case `query.api.Query`);

possibly `()` or `(None,)` when parameters are passed through keyword arguments in `kwargs` (see below).

Keyword Arguments

- **Search** (`dict`) – a dictionary of already 'well-formed' search arguments; in practice, overrides the input `arg` parameter.
- **kwargs** (`dict`) – additional generic query arguments.

See also:

`get()`, `api_query_keys()`.

search(**args*, ***kwargs*)
Search using the media API for `Panoramio`.

```
>>> res = x.search(*args, **kwargs)
```

Parameters **args,kwargs** – arguments used by the API wrapper to search the media.

See also:

getAPI()

setAPI(**apis*, ***kwargs*)

Set the media API clients.

```
>>> x.setAPI(*apis, **kwargs)
```

Parameters **apis** (*tuple of dict*) – dictionary of API wrapper(s) to store in this instance; possibly `()` or `(None,)`, then the APIs are set to the clients associated to the tokens in the current instance as returned by the *Python* wrappers in `['PanoramioAPI']`.

Keyword Arguments **kwargs** (`dict`) – see *getAPI()* method.

wrappers

APIs wrappers – return (`getter`) the list of python wrappers for the different connecting APIs.

class `fetch.panoramio.`**_Panoramioapi**

Panoramio API using a *Python* standard library.

`twitter` Module

About

credits: grazzja, rogliel

version: 0.1 – *since*: Mon Mar 31 09:42:39 2014

Description

Dependencies

call: `media`, `base`, `twitter`, `base`, `status`, `location`, `timing`, `quotation`, `extractor`, `geolocation`, `geoplanetapi`, `alchemyapi`, `imports`, `decorators`

require: `twitter_text`, `twitter`, `birdy`, `TwitterSearch`, `tweepy`, `twython`, `python_twitter`, `sys`, `re`, `time`, `datetime`, `collections`, `urllib`, `urllib2`, `json/simplejson`

Contents

class `fetch.twitter.`**Twitter**(***kwargs*)

Bases: `fetch.base.Base`

A client for the Twitter API using standard twitter API. See https://dev.twitter.com for complete documentation on the API.

Implement different APIs under a unique umbrella. Provide, through composition, an interface to most of the methods from classes in `['_Twitter', '_Birdy', '_Twittersearch',`

`'_Twython', '_Tweepy', '_Twitterapi']` which respectively reimplement the original wrappers `['twitter', 'birdy.twitter', 'TwitterSearch', 'twython', 'tweepy', 'TwitterAPI']` of the Twitter API.

Note: See https://dev.twitter.com/docs/api/1.1/get/search/tweets for specific search request.

See also https://dev.twitter.com/docs/streaming-apis (for streaming) and https://dev.twitter.com/docs/api (REST)

Initialisation of a *Twitter* instance: set all credentials and access rules related attributes.

```
>>> x = Twitter(**kwargs)
```

Keyword Arguments

- **key,secret,token,token_secret** (*str*) – keys used to set various credentials, e.g. key, secret, token, and token secret, usually required for connecting to an API.
- **api,request,acces,authorize,authenticate,redirect** (*str*) – URLs used for accessing the API.
- **read,write** (*str*) – access methods, e.g. `'GET'` and `'POST'` (default), for connection settings.

RT_PATTERNS = <_sre.SRE_Pattern object>

Pattern for identifying retweeted tweets. This regex is adapted from Stack Overflow (http://bit.ly/1821y0J):

http://stackoverflow.com/questions/655903/python-regular-expression-for-retweets

access_methods

Access mehods attribute – return (`getter`) the set of `read`/`write` methods of a *Twitter* instance for connecting to the media.

access_token

Access token attribute (`getter`/`setter`) of a *Twitter* instance used for connection.

access_token_secret

Secret access token attribute (`getter`/`setter`) of a *Twitter* instance used for connection.

access_urls

Access URLs attribute – return (`getter`) the dictionary of URLs of the media API.

api_application_defs

Return (`getter`) the main description attributes of the connecting application associated to a *Twitter* instance.

api_query_keys

Return (`getter`) the set of keys used to formulate a query to a *Twitter* instance of the media.

apis

APIs – return (`getter`) the list of connecting APIs for this media.

auth

Authorization key attribute (`getter`/`setter`) of a *Twitter* instance used for connection.

clear()
Clear the main API parameters of the current *Twitter* instance.

clearAPI()
Clear the media API clients of the current *Twitter* instance.

clearAuthentication()
Clear authentication of the current *Twitter* instance.

clearCredential()
Clear all credentials of the current *Twitter* instance.

clearKey()
Clear all keys of the current *Twitter* instance.

clearToken()
Clear all tokens of the current *Twitter* instance.

clearURL()
Clear media authenticating URLs of the current *Twitter* instance. Methods are left as it is.

client_id
Client ID attribute (`getter`/`setter`) of a *Twitter* instance used for connection.

client_key
Client key attribute (`getter`/`setter`) of a *Twitter* instance used for connection.

client_secret
Secret client key attribute (`getter`/`setter`) of a *Twitter* instance used for connection.

credentials
Credentials attribute – return (`getter`) the list of credentials of a *Twitter* instance used for connecting to the media.

dance()
Do the oauth dance: get and set access token and token secret using internal keys.

```
>>> x.dance()
```

See also:

setToken(), *getToken()*

static display(*tweet*)

static diversity(*tweet*, *div=None*)

domain
Domain URL attribute – return (`getter`) the main URL domain of the media API.

static entities(*tweet*)

static entities_common(*tweets*, *entity_threshold=2*)
Retrieve most common entities in twets.

```
>>> resp = Twitter.entities_common(tweets, entity_threshold=2)
```

entities_count = <fetch.twitter.__entities_count object>

static geocode(*tweet*)

get()
Get/retrieve the stored API parameters: credentials, authorization and apis (in this order).

```
>>> credentials, auth, apis = x.get()
```

See also:

credentials, *auth*, *apis*

getAPI(***kwargs*)

Get the API clients associated to available wrappers to the media using the credentials of the current *Twitter* instance for connecting.

```
>>> apis = x.getAPI(**kwargs)
```

Keyword Arguments

- **access_urls,access_methods** (*dict*) – access methods and URLs used to reset the access to the media; see *getToken()*.
- **all_apis** (*bool*) – optional flag set to True (default) when all APIs obtained by the different *Python* wrappers are returned in the output dictionary.

Returns apis (*dict*) – a dictionary of valid API wrapper(s) (indexed by the *Python* wrappers name in `['twitter', 'birdy.twitter', 'TwitterSearch', 'twython', 'tweepy', 'TwitterAPI']`) for connecting to the media.

getAuthentication(***kwargs*)

Generic method returning the Authentication Authenticate using the credentials passed in args or those already stored in this instance.

```
>>> auth = x.getAuthentication(**kwargs)
```

Keyword Arguments

- **_id** (*str*) – optional client's ID.
- **access_urls** (*dict*) – dictionary of URLs access points; keys must be a subset of `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.

getCredential(***kwargs*)

Get/retrieve the media credentials of the current *Twitter* instance.

```
>>> credentials = x.getCredential()
```

See also:

getKey(), *getToken()*

getDefaultAPI(***kwargs*)

Get default API client using the credentials already stored in the current *Twitter* instance.

```
>>> api = x.getDefaultAPI(**kwargs)
```

Note: *DUMMY* copy/paste of *getAPI()* method.

getDefaultAuthentication(***kwargs*)

Get default authentication using the credentials already stored in the current `Twitter` instance.

```
>>> auth = x.getDefaultAuthentication(**kwargs)
```

Note: *DUMMY* copy/paste of `getAuthentication()` method.

getDefaultCredential(***kwargs*)

Load 'hard-stored' credentials from an external media configuration file.

```
>>> credentials = x.getDefaultCredential(**kwargs)
```

See also:

`getDefaultKey()`, `getDefaultToken()`

getDefaultKey(***kwargs*)

Load 'hard-stored' keys from the media specific file in kinki.media module.

```
>>> keys = x.getDefaultKey(**kwargs)
```

Keyword Arguments

- **package** (*str*) – string indicating a *subpackage* name (e.g. a string like `fetch`) or a *subpackage* path.
- **module** (*str*) – string indicating a python module name (e.g. a string like `imports`) or a json file basename where default parameter arguments are set.

See also:

`kimports()`, `setDefault()`

getDefaultToken(***kwargs*)

Load 'hard-stored' tokens from the media specific file in kinki.media module.

```
>>> tokens = x.getDefaultToken(**kwargs)
```

Keyword Arguments **module,package** – see `getDefaultKey()` method.

getDefaultURL(***kwargs*)

Get/retrieve media authenticating URLs and connecting (read/write) methods of the current `Twitter` instance.

```
>>> urls, methods = x.getDefaultURL(**kwargs)
```

Keyword Arguments

- **module,package** – see `getDefaultKey()` method.
- **oauth_version** (*int*) – oauth framework number; see `getToken()` method.
- **read,write** – access methods; see `getToken()` method.

Returns

- **urls** (*dict*) – a dictionary of URLs indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`.
- **methods** (*dict*) – a dictionary of access methods indexed by `['read', 'write']`.

getKey(***kwargs*)

Get/retrieve the access keys stored in the current `Twitter` instance.

```
>>> keys = x.getKey(**kwargs)
```

getToken(***kwargs*)

Get/retrieve tokens and authorization using OAuth?Session request methods from `requests_oauthlib`/`oauth2` modules.

```
>>> tokens = x.getToken(**kwargs)
```

Keyword Arguments

- **access_urls** (`dict`) – dictionary of URLs used when connecting the API for authorization.
- **access_methods** (`dict`) – access methods (e.g. `'GET'` and `'POST'`) definition.
- **oauth_version** (`int`) – oauth framework number: 1 or 2; default is :literal:2‘.
- **_id** (`str`) – optional client ID.
- **permissions** (`list`) – optional set of permissions used for connecting specific endpoints.

Note:

- OAuth 1 is for user authenticated calls (e.g. posting, following people, sending DMs, etc.),
- OAuth 2 is for application authenticated calls (when you don’t want to authenticate a user and make read-only calls).

---

getURL(***kwargs*)

Get/retrieve the access URLs and access methods of the current `Twitter` instance.

```
>>> URLS, methods = x.getURL()
```

keys

Keys attribute – return (`getter`) the dictionary of credentials used for connecting to the media. `keys` is indexed by the strings in `['key', 'secret', 'token', 'token_secret']`.

loadToken(*token_file*)

Read a token file and set the access token and token secret.

```
>>> x.loadToken(token_file)
```

Parameters **token_file** (*str*) – byte string file with tokens stored (one per line).

See also:

setToken()

static max_id_query(*tweet*, *max_id=None*)

media

Media attribute – return (`getter`) the media name.

medianame

Attribute used for convenience – see *media*.

permissions

Permission attribute – return (`getter`) the set of permissions granted when connecting to the media.

query(**arg*, ***kwargs*)

Define a regular query specific to the media, i.e. depending on the syntax used by the API: the syntax refers to the set of key terms used to query the API (e.g. retrieved using *api_query_keys()* property).

```
>>> key, query = x.query(*arg, **kwargs)
```

Parameters **arg** (dict, *query.api.API*, *query.api.Query*) –

arguments used by the wrapper to query the media; it is either:

- a generic query with generic terms, i.e. any key in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']` used by a `Status` instance, which are then 'translated/converted' to the specific syntax of the media (cases :class'dict' or *query.api.API*), or
- an already well-formed' query arguments with appropriate syntax (case *query.api.Query*);

possibly `()` or `(None,)` when parameters are passed through keyword arguments in `kwargs` (see below).

Keyword Arguments

- **Search** (`dict`) – a dictionary of already 'well-formed' search arguments; in practice, overrides the input `arg` parameter.
- **kwargs** (`dict`) – additional generic query arguments.

See also:

`get()`, *`api_query_keys()`*.

static retweet_attributions(*tweet*)

static retweet_filter(*tweet*, *fields=[]*)

saveToken(*token_file*, **tokens*)

Write a token file to hold the oauth access token and token secret.

```
>>> x.saveToken(*tokens)
```

Parameters

- **token_file** (*str*) – byte string file where to store tokens.
- **tokens** (*list of str*) – Access token and secret access token.

search(**args*, ***kwargs*)

Search using the media API for `Twitter`.

```
>>> res = x.search(*args, **kwargs)
```

Parameters **args,kwargs** – arguments used by the API wrapper to search the media.

See also:

`getAPI()`

set(***kwargs*)

Set/update the API parameters of a `{FETCH}` instance by setting all the attributes used for connection to the media when they have not been set. This is done consistently with internal attributes that have been already set (e.g. in `__init__()` or through default setting) or with the other parameters passed in the keyword arguments.

```
>>> x.set(**kwargs)
```

Keyword Arguments

- **kwargs** (`dict`) – dictionary of arguments used to set the API parameters; its keys should be a any of those already accepted by *`getCredential()`*, *`getURL()`*, *`getAuthentication()`* and *`getAPI()`*. In the case it is empty (i.e. `{}`, default settings are chosen).
- **package,module** (*str*) –

  in the case default settings are selected, strings indicating resp.:

  - a *subpackage* name (e.g. a string like *`fetch`*) or a

  subpackage path,

  - a python module name (e.g. a string like `imports`) or a

  json file basename where default parameter arguments are set.

Note: The `set()` method is essentialy running different setting methods, namely `setCredential()`, `setURL()`, `setAuthentication()` and `setAPI()` methods.

See also:

`kimports()`

setAPI(**apis*, ***kwargs*)

Set the media API clients.

```
>>> x.setAPI(*apis, **kwargs)
```

Parameters **apis** (*tuple of dict*) – dictionary of API wrapper(s) to store in this instance; possibly `()` or `(None,)`, then the APIs are set to the clients associated to the tokens in the current instance as returned by the *Python* wrappers in `['twitter', 'birdy.twitter', 'TwitterSearch', 'twython', 'tweepy', 'TwitterAPI']`.

Keyword Arguments **kwargs** (`dict`) – see `getAPI()` method.

setAuthentication(**auth*, ***kwargs*)

```
>>> x.setAuthentication(*auth, **kwargs)
```

Parameters **auth** (*str*) – authorization key represented as a string; posssibly `()` or `(None,)`, then the key is set through keyword argument auth in `kwargs` (see below), or is derived from the access tokens stored in this instance.

Keyword Arguments **auth** (*str*) – other way of passing the authorization key; ignored when `auth` is passed.

setCredential(**credentials*, ***kwargs*)

Set/update the media credentials (access keys and tokens) of the current `Twitter` instance.

```
>>> x.setCredential(*credentials, **kwargs)
```

Parameters **credentials** (*tuple of str*) – credentials; whenever it is not `()` or , `(None,)` most keyword arguments in `kwargs` (see below) may be ignored.

Keyword Arguments

- **key,secret** – access keys; see `setKey()`.
- **token,token_secret** – access tokens; see `setToken()`.
- **_id** (*str*) – optional client's ID.

setKey(**keys*, ***kwargs*)

Set/update the access keys of the current `Twitter` instance.

```
>>> x.setKey(*keys, **kwargs)
```

Parameters **keys** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **key,secret** (*str*) – client's access key and secret used for connection.

setToken(**tokens, **kwargs*)

Set/update the access tokens of the current *Twitter* instance.

```
>>> x.setToken(*tokens, **kwargs)
```

Parameters **tokens** (*tuple of str*) – access keys given as a 2-tuple of str; whenever it is not `()` or `(None,)`, keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments **token,token_secret** (*str*) – client's access token and token secret used for connection.

setURL(**urls, **kwargs*)

Set media authenticating URLs and connecting (`'GET'/'POST'`) methods.

```
>>> x.setURL(*urls, **kwargs)
```

Parameters **urls** (*dict*) – dict of access URLs (indexed by `['authorize', 'redirect', 'authenticate', 'request', 'acces', 'api']`) and access methods (indexed by `['read', 'write']`); whenever it is not `()` or `(None,)` all keyword arguments in `kwargs` (see below) are ignored.

Keyword Arguments

- **api** (*str*) – Main URL domain of the API.
- **request,acces,authorize,authenticate,redirect** (*str*) – URLs with respective 'request', 'acces', 'authorize', 'authenticate', and 'redirect' endpoints to the media.
- **access_methods** (dict) – dictionary (indexed by `['read', 'write']`) of access methods, e.g. `'GET'` and `'POST'` (default).

static status(*tweet*)

stream(**args, **kwargs*)

static togeojson(*tweet, attributes=False, ptjson=None*)

static user_info(*tweet, info={}, tweet_ids={}, location={}*)

wrappers

APIs wrappers – return (`getter`) the list of python wrappers for the different connecting APIs.

class fetch.twitter.**_Twitter**

Twitter API using standard *Python* `twitter` API module.

Note: See *"Recipes for Mining Twitter"* at https://github.com/ptwobrussell/Recipes-for-Mining-Twitter

---

classmethod connections(*api*, *arg_conn*, ***kwargs*)

Return a cursored collection of user IDs for every user who is somehow connected ('friend' or 'follower') to a given user.

```
>>> ids = x.connections(arg_conn, **kwargs)
```

Parameters **arg_conn** (*str*) – string indicating the type of connection: `'friends'` or `'followers'`.

Keyword Arguments

- **screen_name,user_id** (*str*) – define the identity of the user from which connections will be determine; both options are incompatible.
- **max_ids** (*int*) – Maximum number of ids to retrieve; defaults to 100.

Returns **ids** (*list*) – list of connections (`'friends'` or `'followers'`) of the specified user (:literal:'screen_name'' or :literal:''user_id').

Note: See `recipe__get_friends_followers.py()` in *"Recipes for Mining Twitter"* (https://github.com/ptwobrussell/Recipes-for-Mining-Twitter/)

---

classmethod followers(*api*, ***kwargs*)

Return a cursored collection of user IDs for every follower of a given user.

```
>>> ids = x.followers(**kwargs)
```

Keyword Arguments **kwargs** – see *connections()*.

Note: https://dev.twitter.com/docs/api/1.1/get/followers/ids

---

classmethod friends(*api*, ***kwargs*)

Return a cursored collection of user IDs for every friend of a given user.

```
>>> ids = x.friends(**kwargs)
```

Keyword Arguments **kwargs** – see *connections()*.

Note: https://dev.twitter.com/docs/api/1.1/get/friends/ids

---

classmethod info_by_id(*api*, ***kwargs*)

Retrieve information about given users.

```
>>> id_to_info = x.info_by_id(**kwargs)
```

Keyword Arguments

- **screen_name,user_id** (*list*) – define the ids of some user of which information will be retrieved; both options are incompatible.
- **kwargs** – keyword arguments accepted by *:meth:'API.users.lookup*.

Returns **id_to_info** (*dict*) – dictionary of information available about specified users.

Note: See `recipe__get_user_info.py()` in *"Recipes for Mining Twitter"* (https://github.com/ptwobrussell/Recipes-for-Mining-Twitter/)

classmethod status_by_id(*api*, *_id*)

Show status by id.

```
>>> ids = x.status_show(_id)
```

Note: Shortcut to `API.statuses.show(id=_id, include_entities=1)`.

classmethod status_show(*api*, ***kwargs*)

Show status.

```
>>> ids = x.status_show(**kwargs)
```

Note: Shortcut to `API.statuses.show(include_entities=1, **kwargs)`.

classmethod trends(*api*, ***kwargs*)

Retrieve Twitter trends.

```
>>> t = x.trends(**kwargs)
```

Keyword Arguments

- **lat,long** (*float*) – coordinates of a location of which closest trends are returned.
- **id** (*bool,int*) – either a WOEID or `False` when the set of locations with available trends is returned; defaults to `True` as to select `1`.
- **topic** (*bool*) – set to `True` when most treding topics are returned.

Note: Depending on the keyword arguments above, it returns either: - the overall trending topics, or - the trending topics in a specific WOEID, when information is available

(https://dev.twitter.com/docs/api/1.1/get/trends/place), or

- the trending topics closest to a specified location (https://dev.twitter.com/docs/api/1.1/get/trends/closest), or
- a set of locations that Twitter has trending topic information available and reachable (https://dev.twitter.com/docs/api/1.1/get/trends/available),

class `fetch.twitter.`**_Birdy**

Twitter API using *Python* `birdy` module for dynamically/automatically construct calls.

Note: See https://github.com/inueni/birdy for a list of available methods.

class `fetch.twitter.`**`_Twittersearch`**

Twitter API using *Python* `TwitterSearch` module.

Note: See https://twittersearch.readthedocs.org/en/latest/ for a list of available methods.

class `fetch.twitter.`**`_Twython`**

Twitter API using *Python* `Twython` module.

Note: See https://twython.readthedocs.org/en/latest/api.html for a list of available methods.

classmethod followers(*api*, ***kwargs*)

Return a cursored collection of user IDs for every follower of a given user.

```
>>> ids = x.followers(**kwargs)
```

Keyword Arguments kwargs – see `twython.Twython.get_followers_ids()`.

classmethod friends(*api*, ***kwargs*)

Return a cursored collection of user IDs for every friend of a given user.

```
>>> ids = x.friends(**kwargs)
```

Keyword Arguments kwargs – see `twython.Twython.get_friends_ids()`.

classmethod retweets(*api*, ***kwargs*)

Returns up to 100 of the first retweets of a given tweet.

```
>>> RT = x.retweets(**kwargs)
```

Keyword Arguments kwargs – see `twython.Twython.get_retweets()`.

classmethod trends(*api*, ***kwargs*)

Retrieve Twitter trends.

```
>>> t = x.trends(**kwargs)
```

Keyword Arguments lat,long,id – see `_twitter.trends()`.

Note: Depending on the keyword arguments above, it returns either: - the trending topics in a specific WOEID, when information is available, or - the trending topics closest to a specified location, or - a set of locations that Twitter has trending topic information available

and reachable,

class `fetch.twitter.`**`_Tweepy`**

Twitter API using *Python* `tweepy` module.

Note: See https://pythonhosted.org/tweepy/api.html for a list of available methods.

classmethod cursor(*api*, ***kwargs*)

Return a cursor on tweets flow.

```
>>> t = x.cursor(**kwargs)
```

Keyword Arguments **kwargs** – see `tweepy.Cursor()`.

classmethod followers(*api*, ***kwargs*)

Return a cursored collection of user IDs for every follower of a given user.

```
>>> ids = x.followers(**kwargs)
```

Keyword Arguments **kwargs** – see `tweepy.API.followers_ids()`.

classmethod friends(*api*, ***kwargs*)

Return a cursored collection of user IDs for every friend of a given user.

```
>>> ids = x.friends(**kwargs)
```

Keyword Arguments **kwargs** – see `tweepy.API.friends_ids()`.

classmethod stream(*auth*, ***kwargs*)

Streaming Twitter.

```
>>>  x.stream(auth, **kwargs)
```

Parameters **kwargs** (*dict*) – a dictionary query accepted by `tweepy.Stream()` with keys `'follow'`, `'track'`, `'locations'`, `'languages'`, and/or `'async'`.

classmethod trends(*api*, ***kwargs*)

Retrieve Twitter trends.

```
>>> t = x.trends(**kwargs)
```

Keyword Arguments

- **daily, weekly** (*current,*) – set to return current, daily, and weekly trends respectively.
- **lat,long,id** – see `_twitter.trends()`.

Note: Depending on the keyword arguments above, it returns either: - the trending topics on a given timescale, or - the trending topics in a specific WOEID, when information is available, or - the trending topics closest to a specified location, or - a set of locations that Twitter has trending topic information available

and reachable,

class fetch.twitter.**_Pytwitter**

class fetch.twitter.**_Twitterapi**

Twitter API using *Python* `TwitterAPI` module.

Note: See https://github.com/geduldig/TwitterAPI for available methods.

2.4 In-house data access and storage

2.4.1 Purpose

The *store* package aims at providing various functionalities for loading and saving collected data, *i.e.* seamlessly storing and accessing data on independent (SQL or NoSQL) databases.

Main features:

- relies on unified API for multiple data stores,
- implement simple core methods (`get()`, `put()`, `delete()`, *query()*),
- use hierarchical key scheme inspired by file systems and Google AppEngine model to represent data collected by a given query.

2.4.2 Modules

dbs Package

Package for database configuration.

Description

Global variables to store configuration parameters ('hard' coded in the script) that are common to all DBs.

Usage

```
>>> from kinki.dbs import ALL_DBS, DEF_OWNER_NAME, DEF_DB_HOST, DEF_DB_PORT
```

Contents

dbs.**ALL_DBS**

List of currently supported database/filesystems: `['mongodb']`.

dbs.**DEF_OWNER_NAME**

Application default owner: `{'last': 'qool', 'first': 'deap'}`.

dbs.**DEF_DB_HOST**

Default host machine: `vap-kowalski.jrc.it`.

dbs.**DEF_DB_PORT**

Default port: `27017`.

mongodb Module

Global variables to (hard) store MongoDB personal configuration.

About

credits: grazzja

version: 0.1 – *since*: Wed Jul 09 17:32:10 2014

Usage

```
>>>            from dbs import mongodb
```

Dependencies

***call*: dbs, timing, location,** quotation, status

store Package

Package for database storage facilities.

Usage

```
>>> from kinki.store.mongodb import MongoDB
>>> from kinki.store.filesys import FileSys
```

Contents

mongodb Module

filesys Module

tables Module

2.5 Database and API client

2.5.1 Purpose

The `client` package aims at providing various functionalities for interrogating data (whenever they come from stored or online), *i.e.* seamlessly formulating a query that is understood by all and enables data collection.

Main features:

- seemlessly query APIs and databases,
- ensure that identical queries (at least very similar) are sent to the different APIs and/or databases.

2.5.2 Modules

query Package

Package for formulating explicit queries to APIs or databases.

Usage

```
>>> from kinki.query.api import API
>>> from kinki.query.dbs import DBs
```

Contents

dbs Module

Module for querying databases through exploration of models.

About

credits: grazzja

version: 0.1 – *since*: Sun May 4 10:49:58 2014

Description

Create well-structured and universal (generic) query dictionaries to be used by the APIs searching through social media

Dependencies

call: `media`, `dbs`, `base`, `dskey`, `dsquery`, `utils`, `decorators`

optional `datastore`, `datastore.core`

Contents

class `query.dbs.DBs`(**args*, ***kwargs*)

Bases: `query.base.Base`

Query class used to form queries based on location, quotation (text) and timing.

STATUS_UNIQIFY = ['bbox', 'since', 'until', 'text', 'name']

_clear_mapping(*key=None*, **media*)

Clear the search, stream and/or filter mappings for given media of this instance.

```
>>> x._clear_mapping(key=None, media=None)
```

Parameters **media** (*key,*) – see *_set_mapping()*.

_get_default_mapping(*key*, **media*, ***kwargs*)

Load default search, stream and filter mappings, respectively from media specific variables `KWARGS_API_SEARCH`, `KWARGS_API_STREAM` and `KWARGS_API_FILTER`.

```
>>> mapfilter = x._get_default_mapping(key, *media, **kwargs)
```

Parameters **media** (*key,*) – see *_set_mapping()*.

Keyword Arguments **package** –

_get_mapping(*key=None*, **media*)

Get the search, stream and filter mappings for given media.

```
>>> mapping = x._get_mapping(key=None, *media)
```

Parameters **media** (*key,*) – see *_set_mapping()*.

_set_mapping(*key=None*, *media=None*, **args*, ***kwargs*)

Set the search, stream and filter mappings as the mapping from a universal (generic) query to appropriate search/stream/filter with keywords specific to the media under consideration.

```
>>> x._set_mapping(key=None, media=None, *args, **kwargs)
```

Parameters

- **key** (*str*) – any string among `Filter`, `Stream` and `Search`.
- **media** (*str*) – names(s) of the media whose mapping dictionaries are considered.
- **args** (*dict*) – complementary arguments used for formulating the query.

Keyword Arguments **kwargs** –

clear(**media*)

Clear all fields of a basic query instance.

```
>>> x.clear(*media)
```

Parameters **media** –

clearIdKey()

Clear the Id Key of a basic query instance.

```
>>> x.clearIdKey()
```

clearMapFilter(**media*)

clearMapSearch(**media*)

clearMapStream(**media*)

clearQuery()

Clear the internal query dictionary.

```
>>> x.clearQuery()
```

Parameters **media** –

enquiries_map(**media*)

filter(**status*, ***kwargs*)

Filter a given status so that is is formatted into a set of appropriate search, stream and/or filters that can be applied to given media.

```
>>> query = x.filter(*status, **kwargs)
```

Parameters **status** –

Keyword Arguments **MEDIA** –

get(***kwargs*)

Get/retrieve the (id,query) fields of a basic query instance.

```
>>> idkey, query = x.get(**kwargs)
```

getDefaultMapEnquiry(***kwargs*)

getDefaultMapFilter(**media*, ***kwargs*)

getDefaultMapSearch(**media*, ***kwargs*)

getDefaultMapStream(**media*, ***kwargs*)

getIdKey(***kwargs*)

Get the key string identifier of a query status.

```
>>> idkey = x.getIdKey(quote=False)
```

Parameters **quote** (*bool*) – boolean set to True when the search quote (possibly a full sentence) is to put in the key string.

Returns **idkey** (*str*) – a string representing the key associated to the query.

getMapEnquiry(***kwargs*)

getMapFilter(**media*)

Retrieve the key mapping used to express a filter for a specific media from a dictionary of generic filter keys.

```
>>> keys_map = q.getMapFilter(*media)
```

Parameters **media** (*list of str, str*) – names(s) of the media whose mapping dictionary is retrieved; if not `None`, it should be a subset of the `media` attribute of the query `q`.

Returns **keys_map** (*dict*) – mapping from generic filter keys (*e.g.* keys in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`) to filter keys specific to the media in *`media`*.

getMapSearch(**media*)

Retrieve the key mapping used to express a search query for a specific media from a dictionary of generic query keys.

```
>>> keys_map = q.getMapSearch(*media)
```

Parameters **media** (*list of str, str*) – see `filter_map()`.

Returns **keys_map** (*dict*) – mapping from generic query keys (*e.g.* keys in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until',`

'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']) to search query keys specific to the media in *media*.

getMapStream(**media*)

Retrieve the key mapping used to express a streaming query (possibly different from a regular query) for a specific media from a dictionary of generic query keys.

```
>>> keys_map = q.getMapStream(*media)
```

Parameters **media** (*list of str, str*) – see `filter_map()`.

Returns **keys_map** (*dict*) – mapping from generic query keys (*e.g.* keys in ['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']) to streaming query keys specific to the media in *media*.

getQuery(**media*, ***kwargs*)

Get/retrieve a well-formated query expressed for a given media from a dictionary of generic requests.

```
>>> query = x.getQuery(*media, **kwargs)
```

Parameters **media** (*str*) – the name of the media for whih the query is expressed, e.g. any string among supported media names in :literal:“.

idkey

ID key attribute (`getter`/`setter`).

mappings

Internal mapping property (`getter`) – defines the mappings between generic query keys (filter, search and/or stream) and media-specific query keys.

Dictionary indexed by the list of media defined for the current `{QUERY}` instance (see property attribute `media`).

media

Media name attribute (`getter`/`setter`).

query

Query property (`getter`).

search(*dbs*, ***kwargs*)

Search.

set(**status*, ***kwargs*)

Set the status fields of a basic query instance.

```
>>> x.set(*status, **kwargs)
```

Parameters **status** –

Keyword Arguments

- **kwargs** – keyword parameters used to update the current query.
- **MEDIA** –

setIdKey(***kwargs*)

Set the unique key string identifier of a basic query instance.

```
>>> x.setIdKey(**kwargs)
```

Parameters **kwargs** – See **:method:'getIdKey'**.

setMapEnquiry(***kwargs*)

setMapFilter(*media*, **args*, ***kwargs*)

Define the mapping between generic sketching keys and media-specific used for filter query.

```
>>> q.setMapFilter(media, *args, **kwargs)
```

setMapSearch(*media*, **args*, ***kwargs*)

setMapStream(*media*, **args*, ***kwargs*)

setQuery(***kwargs*)

Set the internal query dictionary following filter, search and stream arguments.

```
>>> x.setQuery(**kwargs)
```

Keyword Arguments **kwargs** –

status

Status attribute (`getter`/`setter`).

Dictionary of properties sketching a 'profile' for the query, e.g. setting a location, a timing, a quote; see `kinki.sketch`.

update(**status*, ***kwargs*)

Update the status of a basic query instance.

```
>>> x.update(*status, **kwargs)
```

Parameters **status** –

Keyword Arguments

- **kwargs** – keyword parameters used to update the current query.
- **MEDIA** –

updateMapEnquiry(***kwargs*)

`api` Module

Module for querying social media through exploration of status.

About

credits: grazzja

version: 0.1 – *since*: Sun May 4 10:49:58 2014

Description

Create well-structured and universal (generic) query dictionaries to be used by the APIs searching through social media.

Dependencies

call: `dbs`, `media`, `base`, `base`, `imports`, `decorators`

Contents

class `query.api.`**`API`**(**sketch*, ***kwargs*)

Bases: `query.base.Base`

Query class used to form queries of social media based on location, quotation (text) and timing

Initialisation of a `{QUERY}` instance: set all status based attributes.

```
>>> q = {QUERY}(*sketch, **kwargs)
```

Parameters **`sketch`** (tuple of dict, *`sketch.status.Status`*) – arguments sketching of a profile, e.g. based on a location, a timing, a quote; see `kinki.sketch`; possibly `{}` or `None` if set later.

Keyword Arguments **`MEDIA`** (*`str, list of str`*) – names(s) of the media for which a query will be expressed; possibly `''` or `None` if set later.

`STATUS_UNIQIFY = ['bbox', 'since', 'until', 'text', 'name']`

`_clear_mapping`(*key=None*, **media*)

Clear the search, stream and/or filter mappings for given media of this instance.

```
>>> x._clear_mapping(key=None, media=None)
```

Parameters **`media`** (*`key,`*) – see *`_set_mapping()`*.

`_get_default_mapping`(*key*, **media*, ***kwargs*)

Load default search, stream and filter mappings, respectively from media specific variables `KWARGS_API_SEARCH`, `KWARGS_API_STREAM` and `KWARGS_API_FILTER`.

```
>>> mapfilter = x._get_default_mapping(key, *media, **kwargs)
```

Parameters **`media`** (*`key,`*) – see *`_set_mapping()`*.

Keyword Arguments **`package`** –

`_get_mapping`(*key=None*, **media*)

Get the search, stream and filter mappings for given media.

```
>>> mapping = x._get_mapping(key=None, *media)
```

Parameters **media** (*key,*) – see *_set_mapping()*.

_set_mapping(*key=None*, *media=None*, **args*, ***kwargs*)

Set the search, stream and filter mappings as the mapping from a universal (generic) query to appropriate search/stream/filter with keywords specific to the media under consideration.

```
>>> x._set_mapping(key=None, media=None, *args, **kwargs)
```

Parameters

- **key** (*str*) – any string among `Filter`, `Stream` and `Search`.
- **media** (*str*) – names(s) of the media whose mapping dictionaries are considered.
- **args** (*dict*) – complementary arguments used for formulating the query.

Keyword Arguments **kwargs** –

clear(**media*)

Clear all fields of a basic query instance.

```
>>> x.clear(*media)
```

Parameters **media** –

clearIdKey()

Clear the Id Key of a basic query instance.

```
>>> x.clearIdKey()
```

clearMapFilter(**media*)

clearMapSearch(**media*)

clearMapStream(**media*)

clearQuery()

Clear the internal query dictionary.

```
>>> x.clearQuery()
```

Parameters **media** –

filter(**status*, ***kwargs*)

Filter a given status so that is is formatted into a set of appropriate search, stream and/or filters that can be applied to given media.

```
>>> query = x.filter(*status, **kwargs)
```

Parameters **status** –

Keyword Arguments **MEDIA** –

get (***kwargs*)

Get/retrieve the (id,query) fields of a basic query instance.

```
>>> idkey, query = x.get(**kwargs)
```

getDefaultMapFilter (**media*, ***kwargs*)

getDefaultMapSearch (**media*, ***kwargs*)

getDefaultMapStream (**media*, ***kwargs*)

getIdKey (***kwargs*)

Get the key string identifier of a query status.

```
>>> idkey = x.getIdKey(quote=False)
```

Parameters **quote** (*bool*) – boolean set to True when the search quote (possibly a full sentence) is to put in the key string.

Returns **idkey** (*str*) – a string representing the key associated to the query.

getMapFilter (**media*)

Retrieve the key mapping used to express a filter for a specific media from a dictionary of generic filter keys.

```
>>> keys_map = q.getMapFilter(*media)
```

Parameters **media** (*list of str, str*) – names(s) of the media whose mapping dictionary is retrieved; if not `None`, it should be a subset of the `media` attribute of the query `q`.

Returns **keys_map** (*dict*) – mapping from generic filter keys (*e.g.* keys in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']`) to filter keys specific to the media in *media*.

getMapSearch (**media*)

Retrieve the key mapping used to express a search query for a specific media from a dictionary of generic query keys.

```
>>> keys_map = q.getMapSearch(*media)
```

Parameters **media** (*list of str, str*) – see `filter_map()`.

Returns **keys_map** (*dict*) – mapping from generic query keys (*e.g.* keys in `['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang',`

'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']) to search query keys specific to the media in *media*.

getMapStream(**media*)

Retrieve the key mapping used to express a streaming query (possibly different from a regular query) for a specific media from a dictionary of generic query keys.

```
>>> keys_map = q.getMapStream(*media)
```

Parameters **media** (*list of str, str*) – see `filter_map()`.

Returns **keys_map** (*dict*) – mapping from generic query keys (*e.g.* keys in ['place', 'bbox', 'bboxLl', 'lat', 'Lon', 'alt', 'sw', 'ne', 'radius', 'grain', 'accuracy', 'unit', 'woeid', 'street', 'address', 'country_id', 'postal_id', 'timedelta', 'timestamp', 'datetime', 'since', 'until', 'unix_since', 'unix_until', 'span', 'date', 'time', 'text', 'language', 'lang', 'locale', 'name', 'email', 'user', 'user_id', 'first_name', 'last_name']) to streaming query keys specific to the media in *media*.

getQuery(**media, **kwargs*)

```
>>> query = x.getQuery(*media, **kwargs)
```

idkey

ID key attribute (`getter`/`setter`).

mappings

Internal mapping property (`getter`) – defines the mappings between generic query keys (filter, search and/or stream) and media-specific query keys.

Dictionary indexed by the list of media defined for the current `{QUERY}` instance (see property attribute `media`).

media

Media name attribute (`getter`/`setter`).

query

Query property (`getter`).

search(*api, **kwargs*)

set(**status, **kwargs*)

Set the status fields of a basic query instance.

```
>>> x.set(*status, **kwargs)
```

Parameters **status** –

Keyword Arguments

- **kwargs** – keyword parameters used to update the current query.
- **MEDIA** –

setIdKey(***kwargs*)
Set the unique key string identifier of a basic query instance.

```
>>> x.setIdKey(**kwargs)
```

Parameters **kwargs** – See **:method:'getIdKey'**.

setMapFilter(*media*, **args*, ***kwargs*)
Define the mapping between generic sketching keys and media-specific used for filter query.

```
>>> q.setMapFilter(media, *args, **kwargs)
```

setMapSearch(*media*, **args*, ***kwargs*)

setMapStream(*media*, **args*, ***kwargs*)

setQuery(***kwargs*)
Set the internal query dictionary following filter, search and stream arguments.

```
>>> x.setQuery(**kwargs)
```

Keyword Arguments **kwargs** –

status
Status attribute (`getter/setter`).

Dictionary of properties sketching a 'profile' for the query, e.g. setting a location, a timing, a quote; see `kinki.sketch`.

update(**status*, ***kwargs*)
Update the status of a basic query instance.

```
>>> x.update(*status, **kwargs)
```

Parameters **status** –

Keyword Arguments

- **kwargs** – keyword parameters used to update the current query.
- **MEDIA** –

exception `query.api.`**ApiError**(*message*, **arg*)
Bases: `query.base.BaseError`

exception `query.api.`**ApiWarning**(*msg*, *expr=None*)
Bases: `query.base.BaseWarning`

class `query.api.`**Query**(*key=None*, ***kwargs*)
Bases: `query.base.Query`

Query class overriding original `datastore.core.query.Query` in `datastore` DAL module.

dict()
Returns a dictionary representing this query.

enquiry(**args*)

classmethod from_dict(*dic*)
Constructs a query from a dictionary.

client Package

2.6 Data analysis and visualisation

2.6.1 Purpose

The probe environment consists in various modules that implement different methodological approaches aiming at data analysis and visualisation.

These approaches have been identified as:

- semantical (*semantics* package): text and tags analysis,
- statistical (*statistics* package): statistics, pattern recognition, machine learning,
- geospatial (*geoprocs* package): raster and vector data analysis, geospatial analysis, mapping,
- contextual (*networks* package): network and graph analysis,
- graphics (*graphics* package): virtual representation and web visualisation.

2.6.2 Modules

semantics Package

Package for data semantics analysis.

Usage

```
>>> from kinki.semantics.nlp import NLP
>>> from kinki.semantics.tags import Cloud
```

nlp Module

tags Module

geoprocs Package

Package for data mapping and raster and vector analysis.

Usage

```
>>> from kinki.geoprocs.maps import Heatmap, Folmap, Netmap, KMLmap
>>> from kinki.geoprocs.rcvs import cvapplier
>>> from kinki.geoprocs.lulc import ITagApi
```

Contents

`maps` Module

Module for geospatial analysis and geographic vizualization.

About

credits: grazzja

version: 0.1 – *since*: Sun Aug 03 16:35:14 2014

Description

Dependencies

call: `imports`, `kinki.sketch.location`, `heatmap`

require: `math`, `numpy`, `re`

optional `geopy`, `folium`, `googlemaps`, `googleplaces`

Contents

class `geoprocs.maps.`**`Heatmap`**(***kwargs*)

Bases: *`geoprocs.maps.Heatmap`*

Class emulating `heatmap` external program for generating heat map images and animations from geographic locations.

See http://www.sethoscope.net/heatmap.

Note: Rely on `Configuration()`, `ImageMaker()` and `ImageSeriesMaker()` implementations.

Initialisation method.

```
>>> x = Heatmap(**kwargs)
```

class `geoprocs.maps.`**`Folmap`**(***kwargs*)

Bases: `folium.folium.Map`

Class overidding the `folium.Map` of `folium` so as to deal with `Location` instances.

Note: The original library Folium enables to visualize data on an interactive Leaflet map. It enables both the binding of data to a map for choropleth visualizations as well as passing Vincent/Vega visualizations as markers on the map.

Folium has a number of built-in tilesets from *OpenStreetMap*, *MapQuest Open*, *MapQuest Open Aerial*, *Mapbox*, and *Stamen*, and supports custom tilesets with *Mapbox* or *Cloudmade* API keys. Folium supports both GeoJSON and TopoJSON overlays, as well as the binding of data to those overlays to create choropleth maps with color-brewer color schemes.

See http://folium.readthedocs.org/en/.

Example

Create (in the local working directory) a simple map 'paris.html' centered over Paris and displaying a marker pointing Melun using *Open Street Map*:

```
>>> paris = Location(place='Paris, France')
>>> melun = Location(place='Melun, France')
>>> x = maps.Folmap(location=paris)
>>> x.simple_marker(location=melun)
>>> x.create_map(path='paris.html')
```

add_tile_layer(***kwargs*)

choropleth(***kwargs*)

fit_bounds(***kwargs*)

keep_in_front(***kwargs*)

render(***kwargs*)

class geoprocs.maps.**Netmap**(*net*, ***kwargs*)

Bases: object

Class for network relationships (e.g. connections over social networks) visualisation, e.g. build a network of geolocalised relationships.

path

str – see above.

relnet

dict – dictionary of connections between locations of the form `{((latO,LonO),(latD,LonD)): qtd, ...}` where:

- `(latO,LonO)` are the (float) latitude and longitude coordinates of the origin location represented by the parameter `lLO` in `net`,
- `(latD,LonD)`, ibid for destiny location represented by the parameter `lLD` in `net`,
- `qtd` is the number of connections equal to `net[(lLO,lLD)]`.

Initialisation of geolocated relationships network.

```
>>> x = Netmap(net, **kwargs)
```

Parameters **net** (*dict*) – dictionary of connections between locations of the form `{(lLO,lLD): qtd, ...}` where:

- `lLO` is the location of the origin location: it can be either a tuple of (float) latitude and longitude coordinates, or a location name (string), or `Location` instance, or a `_geoLocatorAPI` instance
- `lLD`, ibid for destination location,
- `qtd` is an integer that represents the number of connections between both locations.

Keyword Arguments **path** (*str*) – basename of the output file (HTML or JPEG) where to store the displayed network.

create_image(***kwargs*)

Produce a visualisation of geolocated relationships within a network in a map created as a JPEG image.

```
>>> x.create_image(**kwargs)
```

Keyword Arguments

- **n_conn** (*int*) – threshold on the number of connections (i.e. on `qtd`) for which a link will be displayed: below this number, a link is ignored; defaults to 0.
- **path** (*str*) – path to the output (RGB) image (e.g. JPEG or PNG format); if just basename, stored in local working directory; default to `'netmap.jpg'`.
- **size** (*int, tuple, list*) – size of the output image; defaults to 512, i.e. the output image is of dimension [512,512].
- **mode_cm** (*str*) – colormap of output, any in {CMAPS}; default to `'jet'`.

Note: Check original implementation at https://github.com/marcelcaraciolo/Geo-Friendship-Visualization.

create_map(***kwargs*)

Produce a visualisation of geolocated relationships within a network in a map created as a HTML page.

```
>>> x.create_map(**kwargs)
```

Keyword Arguments **path, size, mode_cm** (*n_conn,*) – see **:method:'create_image'**.

create_map_dir(***kwargs*)

Produce a representation of quantified origins/destinations relationships within a network in a map created as a HTML page.

```
>>> x.create_map(**kwargs)
```

Keyword Arguments **path, size, mode_cm** (*n_conn,*) – see **:method:'create_image'**.

class geoprocs.maps.**KMLmap**

Bases: `object`

Class of KML object creation.

classmethod build_map(*title*, **locations*)

Build a KML representation.

```
>>> KMLmap.build_map(title, loc2coords)
```

classmethod create_map(*items*, *centroid_color='ff0000ff'*)

Create a KML representation.

```
>>> KMLmap.create_map(items, centroid_color='ff0000ff')
```

Loosely adapted from http://code.google.com/apis/kml/articles/csvtokml.html

classmethod create_placemark(*kmlDoc*, *d*)

Create a placemark in kml document.

```
>>> KMLmap.create_placemark(kmlDoc, d)
```

rcvs Module

lulc Module

Land cover/Land use analysis.

About

credits: grazzja

version: 0.1 – *since*: Fri Sep 12 14:43:10 2014

Description

Dependencies

call: `location`, `imports`

require: `urllib2`, `urllib`, `json`

***optional*: <put_here_optional_modules>** `numpy`, `scipy`, `matplotlib`, `pylab`, `pickle`, `cPickle`

Contents

class `geoprocs.lulc.`**`ITagApi`**(*url='http://mapshup.com/projects/itag/'*)

Bases: `object`

Quick and dirty API to *iTag* semantic geographical tagger: access an online instance of iTag as a web service.

Automatically tag a geographical footprint against location, land cover, etc. iTag can tag a footprint with the following information: continents, countries, cities, regions and states, geophysical plates, volcanoes, land cover (i.e. forest, water, urban, cultivated, herbaceous, desert, snow, flooded), and population count

Example

```
>>> itag = ITagApi()
>>> footprint = '(6.487426757812523 45.76081241294796,
...               6.487426757812523 46.06798615804025,
...               7.80578613281244 46.06798615804025,
...               7.80578613281244 45.76081241294796,
...               6.487426757812523 45.76081241294796)'
>>> kwargs = {'footprint': footprint}
>>> url, resp = itag(**kwargs)
>>> print resp
    '{"type":"FeatureCollection",
      "features":[{"type":"Feature",
                   "geometry":{"type":"Polygon",
                               "coordinates":[[[6.48742675781,45.
↪7608124129],
                                               [6.48742675781,46.
↪067986158],
                                               [7.80578613281,46.
↪067986158],
```

(continues on next page)

(continued from previous page)


```
                                                    [7.80578613281,45.
↪7608124129],
                                                    [6.48742675781,45.
↪7608124129]]]
                                 },
                   "properties":{"political":{"countries":[{"name":
↪"Switzerland"},
                                                            {"name":
↪"France"},
                                                            {"name":"Italy
↪"}],
                                                 "continents":["Europe"]},
                                 "landCover":{"area":"0.404965386606438",
                                                 "landUse":["Forest","Desert
↪"],
                                                 "landUseDetails":[{"name":
↪"Tree Cover,broadleaved, deciduous, closed",
                                                                    "code
↪":2,
                                                                    "pcover
↪":16.14},
                                                                    {"name
↪":"Tree Cover, needle-leaved, evergreen",
                                                                    "code
↪":4,
                                                                    "pcover
↪":18.63},
                                                                    {"name
↪":"Tree Cover, mixed leaf type",
                                                                    "code
↪":6,
                                                                    "pcover
↪":13.68},
                                                                    {"name
↪":"Herbaceous Cover, closed-open",
                                                                    "code
↪":13,
                                                                    "pcover
↪":10.46},
                                                                    {"name
↪":"Sparse Herbaceous or sparse Shrub Cover",
                                                                    "code
↪":14,
                                                                    "pcover
↪":26.2},
                                                                    {"name
↪":"Cultivated and managed areas",
                                                                    "code
↪":16,
                                                                    "pcover
↪":1.81},
                                                                    {"name
↪":"Mosaic - Cropland \/ Shrub or Grass Cover",
                                                                    "code
↪":18,
                                                                    "pcover
↪":0.01},
```


(continues on next page)

(continued from previous page)

```
                                                                    {"name
↪":"Water Bodies",
                                                                    "code
↪":20,
                                                                    "pcover
↪":0.03},
                                                                    {"name
↪":"Snow and Ice",
                                                                    "code
↪":21,
                                                                    "pcover
↪":15.2},
                                                                    {"name
↪":"Artificial surfaces and associated areas",
                                                                    "code
↪":22,
                                                                    "pcover
↪":0.68}]}
                                  }
                    }]
     }'
>>> print url
    http://mapshup.com/projects/itag/?geophysical=false&countries=true&
↪regions=false&landcover=true&cities=true&population=false&
↪footprint=POLYGON((6.48742675781 45.7608124129,6.48742675781 46.
↪067986158,7.80578613281 46.067986158,7.80578613281 45.7608124129,6.
↪48742675781 45.7608124129))
```

Note: See https://github.com/jjrom/itag.

PARAMETERS = ['countries', 'cities', 'population', 'geophysical', 'regions', 'land

PRETTY_PARAMETERS = ['hierarchical', 'ordered', 'output']

URL = 'http://mapshup.com/projects/itag/'

fetch_data(*url*, *json=False*)

Fetch data from (well formed) URL to *iTag*

```
>>> resp = x.fetch_data(url, json=False)
```

Parameters **url** (*str*) – link to iTag web service to submit the specified query, e.g. the URL output by *get_url()*.

Keyword Arguments **json** (*bool*) – set to `True`, when the output is read as a JSON response; defaults to `False`.

Returns **resp** (*dict*) – dictionary of land cover/land use tags regarding the specified geographical footprint.

get_url(*domain*, ***kwargs*)

Create the query URL to *iTag* web service.

```
>>> url = x.get_url(domain, **kwargs)
```

Parameters **domain** (*str*) – domain of *iTag* web service; defaults to http://mapshup.com/projects/itag/.

Keyword Arguments

- **footprint** (*str,list,tuple*) – a string or a list defining the bounding box of the location to tag as a polygon of `(Long,lat)` coordinates (in this order); this can be `None`, in which case the location should be provided through the use of any keyword in `['bbox', 'lat', 'Lon', 'woeid', 'locality', 'country']` (see also `Geographer`).
- **kwargs** (`dict`) – define the geographical location (see `footprint` above) to tag as well as the output features and the display format:
  - for the output features, available parameters for web service are those in `['countries', 'cities', 'population', 'geophysical', 'regions', 'landcover']`,
  - the display format is specified through keys in `['hierarchical', 'ordered', 'output']`.

Returns **url** (*str*) – link to iTag web service to submit the specified query.

exception `geoprocs.lulc.`**`LULCError`**(*message*, **arg*)
Bases: `kinki.misc.imports.Error`

exception `geoprocs.lulc.`**`LULCWarning`**(*msg*, *expr=None*)
Bases: `kinki.misc.imports.MiscWarning`

`networks` Package

Package for networks analysis.

`cliques` Module

About

credits: grazzja

version: 0.1 – *since*: Tue Apr 08 13:29:52 2014

Description

Dependencies

call: `imports`

***require*: `networkx`,** `os`, `sys`, `string`, `time`, `inspect`, `warnings`, `re`, `operator`, `collections` `math`, `itertools`, `json`/`simplejson`

optional `numpy`, `matplotlib`, `scipy`, `pylab`

Contents

class `networks.cliques.`**`Cliques`**(*directed=False*)
Bases: `object`

Cliques from connections, e.g. social relationships.

`cliques`

compute(**args*)

create(*mutual_connection*, *_id='id'*)

filter()

graph

multigraph

static stringify(*chain*)

exception networks.cliques.**CliquesError**(*message*, **arg*)
Bases: kinki.misc.imports.Error

exception networks.cliques.**CliquesWarning**(*msg*, *expr=None*)
Bases: kinki.misc.imports.MiscWarning

graphics Package

Package for data graphics and visualization.

base Module

Basic tools for data visualisation and display.

About

credits: grazzja

version: 0.1 – *since*: Wed Sep 03 10:16:09 2014

Description

```
from kinki.misc.imports import imports
matplotlib = imports('matplotlib', web='http://matplotlib.org/')
mpld3  = imports('mpld3 ', web='http://mpld3.github.io/')
pandas = imports('pandas', web='http://pandas.pydata.org/')
vispy = imports('vispy', web='http://vispy.org/gallery.html')
bokeh = imports('bokeh', web='http://bokeh.pydata.org/')
nvd3 = imports('nvd3', web='https://github.com/areski/python-nvd3')
vincent = imports('vincent', web='https://github.com/wrobstory/
↪vincent')
ggplot = imports('ggplot', web='https://github.com/yhat/ggplot')
```

Dependencies

call: imports, decorators

require: os, inspect, warnings, re

optional: numpy, scipy, matplotlib, pylab, mpld3, pandas, vispy, nvd3, ggplot, vincent, bokeh

Contents

statistics Package

Package for statistical analysis.

base Module

Basic tools for statistical data analysis.

About

credits: grazzja

version: 0.1 – *since*: Wed Sep 03 11:30:29 2014

Description

```
from kinki.misc.imports import imports
matplotlib = imports('matplotlib', web='http://matplotlib.org/')
pandas = imports('pandas', web='http://pandas.pydata.org/')
sklearn = imports('sklearn', web='https://github.com/scikit-learn/
↪scikit-learn'})
mlpy  = imports('mlpy ', web='http://sourceforge.net/projects/mlpy/
↪')
seaborn = imports('seaborn', web='https://github.com/mwaskom/seaborn
↪')
pyNN = imports('pyNN', web='https://github.com/NeuralEnsemble/PyNN/
↪')
rpy2 = imports('rpy2', web='https://pypi.python.org/pypi/rpy2')
```

Dependencies

call: `imports`, `decorators`

require: `os`, `inspect`, `warnings`, `re`

optional: `numpy`, `scipy`, `matplotlib`, `pylab`, `pandas`, `sklearn`, `mlpy`, `seaborn`, `pyNN`

Contents

2.7 Miscellaneous utilities and third-party functionalities

2.7.1 Modules

misc Package

Package for miscellaneous utility functions.

Usage

```
>>> from kinki.misc.imports import Message, Module, imports, kimports,␣
↪mimports
>>> from kinki.misc.decorators import Decorator, class_decorator,␣
↪DecoratorConditions
>>> from kinki.misc.pools import Pools
>>> from kinki.misc import utils
```

Contents

decorators Module

Module providing with various class, functions and methods decorators.

About

credits: grazzja

version: 0.1 – *since*: Wed May 07 13:57:51 2014

Description

See useful resources at https://wiki.python.org/moin/PythonDecoratorLibrary.

Contents

class misc.decorators.**Docstring**

Bases: object

A bunch of docstrings classes, metaclasses and methods decorators.

classmethod decorate(*attr_names=None*, ***kwargs*)

Universal decorator that enables to parameterize the docstring of any class by rewritting and reformating (possibly emptying) the __doc__ special attributes (and only this) of all its members.

```
>>> new_cls = Docstring.decorate(attr_names=None, **kwargs)(cls)
```

Examples

Note: Inside this project, we also use the *decorate()* method to decorate, through the special attribute __metaclass__, all methods inherited from base classes, namely:

```
class Base(object):
    def dummy_method_from_Base():
        """A certainly {quality} documentation that is decorated."""
        return

class Derived(Base):
    @Docstring.decorate(Base.__dict__.keys(),quality='useful')
    class __metaclass__(type): pass
    def dummy_method_from_Derived():
        """A certainly {quality} documentation that is left as it␣
↪is."""
        return
```

as indeed, this way, the methods inherited from Base, and those only, are decorated:

```
>>> print Derived.dummy_method_from_Derived.__doc__ # not decorated
    A certainly {quality} documentation that is left as it is.
>>> print Derived.dummy_method_from_Base.__doc__
    A certainly useful documentation that is decorated.
```

classmethod format(***kwargs*)

Universal decorator that enables to parameterize the docstring of any method or class by rewritting and reformating (possibly emptying) its __doc__ special attribute (and only this).

```
>>> new_obj = Docstring.format(**kwargs)(obj)
```

Examples

The format conversion method can be applied indifferently over classes or methods:

```
>>> class dummy_class(object):
...     """A certainly {quality} documentation."""
...     pass
>>> useful_class = Docstring.format(quality='useful')(dummy_class)
>>> print useful_class.__doc__
    A certainly useful documentation.
>>> def dummy_method():
...     """A certainly {quality} documentation."""
...     return
>>> useless_method = Docstring.format(quality='useless')(dummy_
↪method)
>>> print useless_method.__doc__
    A certainly useless documentation.
```

Note: Mostly a copy/paste of `format_method()` and `format_class()` methods.

See also:

`format_method()`, `format_class()`

classmethod format_class(***kwargs*)

Ibid `format_method()` for decorating classes by rewritting and reformating (possibly emptying) their `__doc__` special attributes.

```
>>> new_cls = Docstring.format_class(**kwargs)(cls)
```

Examples

Let's consider a dummy function whose documentation will be decorated:

```
>>> class dummy_class(object):
...     """A certainly {quality} documentation."""
...     pass
```

```
>>> useless_class = Docstring.format_class(quality='useless')(dummy_
↪class)
>>> print useless_class.__name__
    dummy_class
>>> print useless_class.__doc__
    A certainly useless documentation.
```

However, in the case an `empty` keyword is passed, the docstring will be emptied (even in the case other keyword arguments are present)

```
>>> empty_class = Docstring.format_class(VOID=True)(dummy_class)
>>> empty_class.__doc__
    ''
>>> empty_class = Docstring.format_class(quality='useless',␣
↪VOID=True)(dummy_class)
```

(continues on next page)

(continued from previous page)

```
>>> empty_class.__doc__
    ''
```

See also:

format_method()

classmethod format_method(***kwargs*)

Method decorator that enables to parameterize the docstring of a method by rewritting and reformating (possibly emptying) its `__doc__` special attribute (and only this).

```
>>> new_func = Docstring.format_method(**kwargs)(func)
```

Example

Let's consider a dummy function whose documentation will be decorated:

```
>>> def dummy_method():
...     """A certainly {quality} documentation."""
...     return
```

We can then adapt this 'base' function:

```
>>> useless_method = Docstring.format_method(quality='useless
↪')(dummy_method)
>>> print useless_method.__doc__
    A certainly useless documentation.
>>> useful_method = Docstring.format_method(quality='useful')(dummy_
↪method)
>>> print useful_method.__doc__
    A certainly useful documentation.
```

We can even 'empty' the documentation using the `VOID` keyword:

```
>>> empty_method = Docstring.format_method(VOID=True)(dummy_method)
>>> empty_method.__doc__
    ''
```

Note that, as it is, the docstring of the base function looks rather irrelevant:

```
>>> print dummy_method.__doc__
    A certainly {quality} documentation.
```

In particular `staticmethod` and `classmethod` objects are also supported (i.e. instance methods as well as normal functions) thanks to the `wrapt` module and the `FunctionWrapper` class therein:

```
class A(object):

    @Docstring.format_method(quality='static')
    @staticmethod
    def decorated_static_method():
        """A {quality} method documentation."""
        return
```

(continues on next page)

(continued from previous page)

```
    @Docstring.format_class(quality='class')
    @classmethod
    def decorated_class_method():
        """A {quality} method documentation."""
        return
```

where the decorator is inserted prior to the method closure definition (special decorators `@staticmethod` and `@classmethod`). Therefore, the methods are decorated as expected:

```
>>> print A.decorated_static_method.__doc__
    A static method documentation.
>>> A.decorated_class_method.__doc__
    A class method documentation.
```

See also:

format_class()

class misc.decorators.**Slicing**

Bases: `object`

A bunch of docstrings classes, metaclasses and methods decorators.

static lapply(*func*)

Method decorator that enables to iterate a function that operates on single variable over multiple entries (e.g. `list` or `tuple`).

```
>>> new_func = Slicing.lapply(func)
```

Examples

Let's try to decorate a simple function and test it:

```
>>> alist = [1, 2, 3, 4, 5, 6]
>>> increment = lambda x: x + 1
>>> print increment(alist)
    Traceback (most recent call last):
    ...
    TypeError: can only concatenate list (not "int") to list
>>> apply_increment = Slicing.lapply(increment)
>>> print apply_increment(alist)
    [2, 3, 4, 5, 6, 7]
```

We can also naturally declare it as:

```
@Slicing.lapply
def apply_increment(X):
    return increment(X)
```

Note: The definition of this function was inspired by the 'lapply' (family of) function(s) implemented in R.

This is very useful for generalizing simple functions that apply on single entities, e.g. scaling

some basic operations in a non optimized way. It can be further combined with multiprocessing mapping for more efficient computations.

See also:

lclean(), *rapply()*

static lclean(*func*)

Method decorator that enables to filter out `None` elements passed to a function.

```
>>> new_func = Slicing.lclean(func)
```

Examples

Let's decorate a simple function and test it:

```
>>> alist = [1, None, 3, 4, None, 6, 7, None]
>>> print sum(alist)
    Traceback (most recent call last):
    ...
    TypeError: unsupported operand type(s) for +: 'int' and
↪'NoneType'
>>> clean_sum = Slicing.lclean(sum)
>>> print clean_sum(alist)
    21
```

Let's consider a multiply decorated function:

```
@Slicing.lclean
@Slicing.lapply
def apply_clean_increment(x):
    return x + 1
```

and check that both decorators are applied:

```
>>> print apply_clean_increment(alist)
    [2, 4, 5, 7, 8]
```

See also:

lapply(), *rapply()*

static rapply(*func*)

Method decorator that enables to recursively operate a function on the outputs it generates.

```
>>> new_func = Slicing.rapply(func)
```

Examples

Let's try to decorate a simple function and test it:

```
>>> alist = [1, 2, 3, 4, 5, 6]
>>> def useless(x, l=None, d=None):
...     if l is None:   l=[]
...     l.append(2*x)
```

(continues on next page)

(continued from previous page)

```
...     if d is None:   d={}
...     d.update({x: 2*x})
...     return l, d
>>> print useless(alist)
    Traceback (most recent call last):
    ...
    TypeError: unhashable type: 'list'
>>> apply_useless = Slicing.rapply(useless)
>>> print apply_useless(alist)
    ([2, 4, 6, 8, 10, 12],
     {1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12})
```

We can also naturally declare it as:

```
@Slicing.rapply
def apply_useless(X):
    return useless(X)
```

Note: Initial settings (likewise `if l is None: l=[]` or `if d is None: d={}` in the code above) are usually required to avoid unwanted side parameter effects.

See also:

lapply(), *lclean()*

class misc.decorators.**Conditioning**(*pre*, *post*, *use_conditions=True*)

Bases: `object`

Provide pre-/postconditions as function decorators. This code was taken from: https://wiki.python.org/moin/PythonDecoratorLibrary.

Example

We recycle here the examples from the original implementation:

```
def in_ge20(inval):
    assert inval >= 20

def out_lt30(retval, inval):
    assert retval < 30

@decorator_precondition(in_ge20) # equivalent to: Conditioning(in_ge20,␣
↪None)
@decorator_postcondition(out_lt30) # equivalent to: Conditioning(None,␣
↪out_lt30)
def apply_increment(x):
    return x+1

@Conditioning(in_ge20, out_lt30)
def apply_increment_sim(x):
    return x+1
```

and simply run:

```
>>> print apply_increment(1)
    Traceback (most recent call last):
    ...
    AssertionError
>>> print apply_increment(29)
    Traceback (most recent call last):
    ...
    AssertionError
>>> print apply_increment(25)
    26
>>> print apply_increment_sim(20)
    21
```

See also:

decorator_precondition(), *decorator_postcondition()*

class misc.decorators.**MetaclassProxy**

Bases: type

Decorate the class being created & preserve __metaclass__ of the parent.

It executes two callbacks: before and after creation of a class, that allows to decorate them.

Between two callbacks, it tries to locate any __metaclass__ in the parents (sorted in MRO).

If found — with the help of __new__ method - it mutates to the found base __metaclass__. If not found, it just instantiates the given class.

Note: see http://stackoverflow.com/questions/4651729/metaclass-mixin-or-chaining

classmethod post_new(*newclass*)

Decorate a class after creation

classmethod pre_new(*name*, *bases*, *attrs*)

Decorate a class before creation

misc.decorators.**class_decorator**(*decorator*, **attr_names*, ***kwargs*)

Class decorator used to automatically decorate a family of methods of a class.

```
>>> new_cls = class_decorator(decorator, *attr_names, **kwargs)(cls)
```

Parameters

- **decorator** (*callable*) – universal or specific function/class decorator used to decorate methods and subclasses of the class cls.
- **attr_names** (*str, list of str, optional*) – list of methods or classes to be decorated in the new class new_cls; if None or (), all instance/static/class methods as well as all classes of cls are decorated in new_cls, excepted those whose names are in excludes (if any, see below).

Keyword Arguments

- **excludes** (*list of str, optional*) – list of methods or classes to exclude from the list of decorated ones in the new class new_cls.
- **'base'** (*str, bool, optional*) – variable conveying the name of a parent class of the input decorated class cls whose attributes will be further

decorated; set to `True` when a default name is selecte (:literal:'Base') and `False` when the parent class should be ignored.

Returns **new_cls** (*class*) – class inherited from `new_cls` where methods and classes whose names match `attr_names` (all if `()`) are decorated using the decorator *`method_decorator`*.

Examples

Let's create a method decorator that increments the output of a function on integer:

```
>>> increment = lambda x: x+1
>>> def decorator_increment(func):
...     def decorator(*args, **kwargs):
...         return increment(func(*args, **kwargs))
...     return decorator
```

Let's consider a class with two methods:

```
>>> class A(object):
...     def increment_twice(self,x):
...          return increment(increment(x))
...     def multiply_by_2(self, x):
...          return 2*x
```

Let's implement different inherited classes using the class decorator:

```
>>> @class_decorator(decorator_increment)
... class B(A): # all methods of this class are decorated
...     pass
>>> @class_decorator(decorator_increment, 'multiply_by_2')
... class C(A): # only multiply_by_2 method is decorated
...     pass
```

and test it :

```
>>> a = A(); b = B(); c = C()
>>> print a.increment_twice(1)
    3
>>> print b.increment_twice(1) # decorated with one more call to␣
↪increment
    4
>>> print c.increment_twice(1) # not decorated
    3
>>> print a.multiply_by_2(1)
    2
>>> print b.multiply_by_2(1)
    3
>>> print c.multiply_by_2(1) # also decorated
    3
```

Note: See http://andrefsp.wordpress.com/2012/08/23/writing-a-class-decorator-in-python/

`misc.decorators.`**`method_decorator`**(*func_decorator=None*, **func_names*, ***func_exclude*)

Generic method decorator used to automatically decorate methods.

`misc.decorators.`**`metaclass_maker`**(*left_metas=()*, *right_metas=()*)

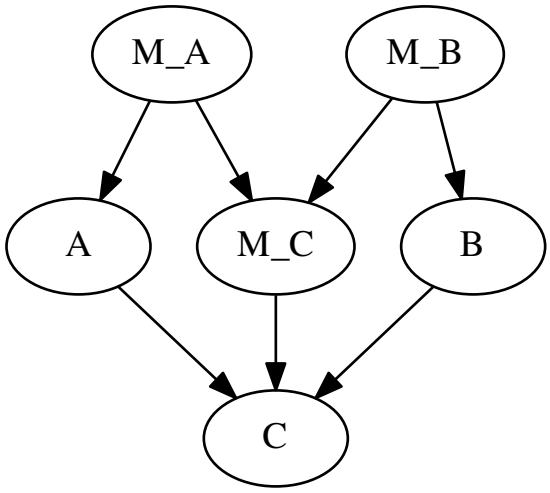
Deal with metaclass conflict, i.e. when "the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases"

```
>>> metaclass = metaclass_maker(left_metas=(), right_metas=())
```

Arguments: left_metas,right_metas : class

classes (not strings) representing the metaclasses to 'merge'; note that `left_metas` has priority over `right_metas`.

Note: The simplest case where a metatype conflict happens is the following. Consider a class `A` with metaclass `M_A` and a class `B` with an independent metaclass `M_B`; suppose we derive `C` from `A` and `B`. The question is: what is the metaclass of `C` ? Is it `M_A` or `M_B` ? The correct answer is `M_C`, where `M_C` is a metaclass that inherits from `M_A` and `M_B`, as in the following graph:



Note: see http://code.activestate.com/recipes/204197-solving-the-metaclass-conflict/

`misc.decorators.`**`metaclass_decorator`**(*method_decorator*, **func_names*)

Metaclass decorator used to automatically decorate the *metaclass* attribute of a class.

```
>>> new_cls = metaclass_decorator(method_decorator, *func_names)(cls)
```

Note: Note that when the class `cls` is inherited from another one, say `super_cls`, all methods `cls` that are inherited from `super_cls` (and not overriden or specific to the class) are decorated in the new class `new_cls`.

See http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Metaprogramming.html

`misc.decorators.`**`decorator_precondition`**(*precondition*, *use_conditions=True*)

See also:

decorator_postcondition()

`misc.decorators.`**`decorator_postcondition`**(*postcondition*, *use_conditions=True*)

See also:

decorator_precondition()

`imports` Module

Module for multicase import of modules, classes or global variables.

About

credits: grazzja

version: 0.1 – *since*: Thu Apr 03 14:14:44 2014

Description

See https://docs.python.org/3.5/library/importlib.html.

Dependencies

call: `imports`, `decorators`

require: `sys`, `webbrowser`, `warnings`, `logging`

optional `importlib`, `modulefinder`

Contents

exception `misc.imports.`**`MiscWarning`**(*msg*, *expr=None*)

Bases: `exceptions.Warning`

Basic class for warnings.

exception `misc.imports.`**`Error`**(*message*, **arg*)

Bases: `exceptions.Exception`

Basic class for exceptions.

class `misc.imports.`**`Logger`**(***kwargs*)

Bases: `object`

Basic logger class.

class `misc.imports.`**`Message`**(*msg=None*)

Bases: *misc.imports.Message*

Basic messaging class for raising errors, warning and displaying infos.

Examples

```
>>> m = Message("This is my message")
>>> m.raises(IOError)
    Traceback (most recent call last):
    ...
    IOError: This is my message
>>> m.warns()
>>> with open('kinki.log', 'r') as f: # check LOGGER file
...     last = f.readlines()[-1]
>>> print last
    WARNING: This is my message
>>> m.info()     # ibid, check LOGGER file
```

`misc.imports.`**`raiseMessage`**(**args*)

Raise message error of defined type.

```
>>> raiseMessage(*args)
```

Examples

```
>>> raiseMessage(TypeError, "This is a TypeError error")
    Traceback (most recent call last):
    ...
    TypeError: This is a TypeError error
>>> raiseMessage("This is a default IOError error")
    Traceback (most recent call last):
    ...
    IOError: This is a default IOError error
```

`misc.imports.`**`infoMessage`**(**args*)

Export info message into `LOGGER` file kinki.log.

```
>>> infoMessage(msg)
```

Example

```
>>> infoMessage("This provide some information")
>>> with open('kinki.log', 'r') as f:
...     last = f.readlines()[-1]
>>> print last
    INFO: This provide some information
```

`misc.imports.`**`warnMessage`**(**args*)

Export warn message into `LOGGER` file kinki.log.

```
>>> warnMessage(msg)
```

Example

```
>>> warnMessage("This is a warning")
>>> with open('kinki.log', 'r') as f:
```

(continues on next page)

(continued from previous page)

```
...     last = f.readlines()[-1]
>>> print last
    WARNING: This is a warning
```

class misc.imports.**Module**(**args*, ***kwargs*)

Bases: object

Module representation class used for various import operations.

requires

list of modules/classes required to be imported with the instance.

modules

list of modules finally imported with the instance if any.

classes

list of classes finally imported with the instance if any.

Initialisation of a *Module* instance:

```
>>> x = Module(*args, **kwargs)
```

Parameters **args** (*module, str, dict*) – list of (equivalent) modules to import, defined as either a module itself (loaded in *sys.modules*), or a string name or a dictionary (like that in kwargs, see Keyword Arguments below) representing a required module.

Keyword Arguments

- **imp** (*str*) – name of a module/package to import.
- **as** (*str*) – name of the alias of the import (like in import simplejson as json).
- **top** (*str*) – relative path of the import, e.g. when a submodule of a package is imported.
- **web** (*str*) – hyperlink to where the module/package can be found if the import failed.
- **err** (*bool*) – set to True if an error should be raised in case of failed import; default to False.

Examples

Import a module … which has been already imported indeed (i.e. already in *sys.modules*):

```
>>> import json
>>> module = Module(json)
>>> module.modules == [json]
    True
```

Using string representation:

```
>>> module = Module('json')
>>> print module.modules # even if already loaded...
    []
```

(continues on next page)

(continued from previous page)

```
>>> print module.requires
    [{'imp':'json', 'err':False, 'web':'', 'top':None, 'as_':''}]
>>> module = Module({ 'imp':'numpy', 'as_':'np', 'imp':'http://www.numpy.
↪org/'})
>>> print module.requires
    [{'imp':'numpy', 'err':False, 'web':'http://www.numpy.org/', 'top
↪':None, 'as_':'np'}]
>>> module = Module('numpy', as_='np')
>>> print module.requires
    [{'imp':'numpy', 'err':False, 'web':'', 'top':None, 'as_':'np'}]
```

Further:

```
>>> module = Module('a_dummy_non_existent_module', err=True)
>>> module.imports
    Traceback (most recent call last):
    ...
    ImportError: a_dummy_non_existent_module module is required
```

classmethod depth(*python_prog*, *report=False*)

Retrieve the set of modules imported by a script, similarly to `modulefinder.ModuleFinder`, and display some information regarding non/imported modules.

```
>>> ddp = Module.depth(python_prog, report=False)
```

Parameters **python_prog** (*str*) – name of python script; it will be searched for in all available directories (i.e. those in `sys.path[:]`); indicate an absolute or relative path if not in the *PYTHONPATH* environment variable.

Keyword Arguments **report** (*bool*) – set to `True` when a report list on loaded and non loaded modules is printed on the termimal.

Returns **ddp** (*dict*) – dictionary with keys :literal:'depgraph' and :literal:'types' storing the set of modules imported by the script `python_prog`.

Note: see modulefinder.ModuleFinder.

---

imports(**args*, ***kwargs*)

Note: *`imports()`* implicitely calls *`Module.imports()`* so that `imports(*args, **kwargs)` and `Module(*args, **kwargs).imports()` are equivalent; see further description and examples of *`imports()`* instead.

---

mimports(**args*, ***kwargs*)

Note: *`mimports()`* implicitely calls *`Module.mimports()`* so that `mimports(*args, **kwargs)` and `Module(*args, **kwargs).mimports()` are equivalent; see further description and examples of *`mimports()`* instead.

`misc.imports.imports(*args, **kwargs)`

Import the **first available module/class** from a list of modules/classes.

```
>>> res = imports(*args, **kwargs)
```

Parameters **args** (*module, str, dict*) – list of (supposedly equivalent) modules to import, defined as either a module itself (loaded in *sys.modules*), or a string name or a dictionary (like that in `kwargs`, see Keyword Arguments below) representing a required module. The import will be stopped after the first module in the list has been imported without error (hence, likewise size, order matters).

Keyword Arguments

- **imp** (*str*) – name of a module/package to import.
- **as** (*str*) – name of the alias of the import (like in `import simplejson as json`), whatever the name of the module/class eventually imported.
- **top** (*str*) – relative path of the import, e.g. when a submodule of a package is imported.
- **web** (*str*) – hyperlink to where the module/package can be found if the import failed.
- **err** (*bool*) – set to `True` if an error should be raised in case of failed import; default to `False`.

Returns **res** (*module, class, callable*) – whatever could be imported first without error.

Examples

'Preferential' modules can be imported:

```
>>> scientific_package = imports.imports('numpy', 'scipy')
>>> scientific_package.__name__=='numpy'
    True
>>> import numpy
>>> scientific_package == numpy
    True
```

A typical use-case is when a 'secondary' module is preferred instead of a standard one:

```
>>> json = imports('simplejson','json', **{'as_':'json'})
>>> oauth = imports('requests_oauthlib', 'oauth2', as_='oauth')
```

Classes and methods can also be imported as well, if not the whole module:

```
>>> method = imports('histogram2d', **{'top':'numpy'})
>>> callable(method)
    True
>>> method == numpy.histogram2d
    True
>>> MyNdArray = imports('ndarray', top='numpy')
>>> isinstance(MyNdArray,type) # not a class indeed... ok, proceed
    True
>>> MyNdArray([2,2],dtype=int) # print an array with random values
    array([[70386284, 30388636],
           [30388636, 38421824]])
```

(continues on next page)

(continued from previous page)

```
>>> MyArray = imports('array', top='numpy')
>>> callable(MyArray)
    True
>>> MyArray([[1,2],[3,4]])
    array([[1, 2],
           [3, 4]])
```

`misc.imports.`**`kimports`**(**key_names*, ***kwargs*)

Load global variables (constants) from a python module in `kinki` package, or a json file in `kinki` path.

```
>>> keys = kimports(*key_names, **kwargs)
```

Parameters **`key_names`** (*`list of str, str`*) – string representing the (known) names of the global variable to load.

Keyword Arguments

- **`package`** (*`str`*) – string indicating a subpackage name (e.g. a string like *`misc`*) or a subpackage path.
- **`module`** (*`str`*) – string indicating a python module name (e.g. a string like `imports`) or a json file basename.

Returns **keys** (*list*) – variables equal to the loaded global variables; `keys` is eitehr a list of same lenght as `key_names` or a single variable in the case `key_names` was reduced to a single string.

Examples

It can be used for loading global variables from specific modules:

```
>>> key = kimports('LOG_FILENAME', **{'module':'imports', 'package':'misc
↪'})
>>> print key
    kinki.log
>>> key == imports.LOG_FILENAME
    True
>>> key = kimports('KW_MODULE', 'KW_PACKAGE', module='imports', package=
↪'misc')
>>> print key
    ['module', 'package']
>>> key == [imports.KW_MODULE, imports.KW_PACKAGE]
    True
```

It can be used to look through json files as well:

```
>>> var = kimports('organization', module='metadata', package='..')
>>> print var
    European Commission (EC - DG ESTAT)
```

`misc.imports.`**`mimports`**(**args*, ***kwargs*)

Import **all available modules/classes** from a list of modules/classes.

```
>>> res = mimports(*args, **kwargs)
```

Parameters **args** (*module, str, dict*) – same as *imports()* methods. However, all modules will be imported if possible.

Keyword Arguments **imp,top,web,err** – see *imports()*.

Returns **res** (*dict*) – a dictionary of classes/modules and/or methods that have been loaded and imported indexed by the (lower case) names of the objects.

Examples

All modules are imported, no preference:

```
>>> scientific_packages = mimports('numpy', 'scipy')
>>> isinstance(scientific_packages,dict)
    True
>>> all([mod in ('numpy', 'scipy') for mod in mod.scientific_packages()]
    True
>>> scientific_package['numpy'] == numpy
    True
>>> jsons = mimports('simplejson', 'json')
>>> print [type(j) for j in jsons.values()]
    [<type 'module'>, <type 'module'>]
```

Ibid for classes and methods:

```
>>> MyArrays = mimports({'imp':'ndarray', 'top':'numpy'},
...                     {'imp':'MaskedArray', 'top':'numpy.ma'})
>>> MyArrays # note the lower case name
    {'maskedarray': <class 'numpy.ma.core.MaskedArray'>, 'ndarray':
↪<type 'numpy.ndarray'>}
>>> MyArrays['ndarray'].__name__ == 'ndarray'
    True
>>> MyArrays['MaskedArray'].__name__ == 'ndarray'
    True
```

`pools` Module

Pool workers for parallel mapping and task running.

About

credits: grazzja

version: 0.1 – *since*: Wed Jul 30 11:33:07 2014

Dependencies

require: `numpy`, `multiprocessing`

Contents

class `misc.pools.`**Parallel**

Bases: `object`

Class of pool workers for parallel mapping.

static map_tasks (*function, sequence, numcores=None*)

A parallelized version of the native Python map function that utilizes the Python multiprocessing module to divide and conquer sequence.

```
>>> res = map_tasks(function, sequence, numcores)
```

Parameters

- **function** – callable function that accepts argument from iterable.
- **sequence** – iterable sequence.
- **numcores** – number of cores to use.

Note: :meth :~*Parallel.map_tasks* does not yet support multiple argument sequences.

---

static run_tasks(*procs*, *err_q*, *out_q*, *num*)

A function that executes populated processes and processes the resultant array. Checks error queue for any exceptions.

```
>>> res = run_tasks(procs, err_q, out_q, num)
```

Parameters

- **procs** – list of Process objects.
- **out_q** – thread-safe output queue.
- **err_q** – thread-safe queue to populate on exception.
- **num** – length of resultant array.

static worker(*f*, *ii*, *chunk*, *out_q*, *err_q*, *lock*)

A worker function that maps an input function over a slice of the input iterable.

```
>>> res = worker(f, ii, chunk, out_q, err_q, lock)
```

Parameters

- **f** – callable function that accepts argument from iterable.
- **ii** – process ID.
- **chunk** – slice of input iterable.
- **out_q** – thread-safe output queue.
- **err_q** – thread-safe queue to populate on exception.
- **lock** – thread-safe lock to protect a resource (useful in extending `map_tasks()`).

class misc.pools.**MapReduce**(*map_func*, *reduce_func*, *num_workers=None*)

Bases: `object`

Multi-processor single-server implementation of the *MapReduce* schema. Use the `Pool` class to create a simple single-server

---

Note: In a MapReduce-based system: * input data is broken down into chunks for processing by different worker

instances,

- each chunk of input data is mapped to an intermediate state using a simple transformation,
- the intermediate data is then collected together and partitioned based on a key value so that all of the related values are together,
- finally, the partitioned data is reduced to a result set.

Initialise a pool of *MapReduce* implementation.

```
>>> mpr = MapReduceSingle(map_func, reduce_func, num_workers=None)
```

Parameters

- **map_func** (*callable*) – function to map inputs to intermediate data; it takes as argument one input value and returns a tuple with the key and a value to be reduced.
- **reduce_func** (*callable*) – function to reduce partitioned version of intermediate data to final output; it takes as argument a key as produced by `map_func` and a sequence of the values associated with that key.

Keyword Arguments **num_workers** (*int*) – number of workers to create in the pool; defaults to the number of CPUs available on the current host.

partition(*mapped_values*)

Organize the mapped values by their key.

```
>>> pt = mpr.partition(mapped_values)
```

Returns **pt** (`defaultdict`) – unsorted sequence of tuples with a key and a sequence of values.

`utils` Module

Various utility functions for:

- system management,
- string manipulations,
- structure utilities

About

credits: grazzja

version: 0.1 – *since*: Sat Apr 19 10:42:30 2014

Description

Dependencies

call: `imports`, `decorators`

require: `os`, `sys`, `time`, `warnings`, `re`, `numpy`

optional `datetime`

Contents

class misc.utils.**DictDiffer**(*current_dict*, *past_dict*)

Bases: object

A dictionary difference calculator.

```
>>> diff = DictDiffer(current_dict, past_dict)
```

Calculate the difference between two dictionaries as: (1) items added (2) items removed (3) keys same in both but changed values (4) keys same in both and unchanged values

Example

```
>>> past = {'a': 1, 'b': 1, 'c': 0}
>>> cur = {'a': 1, 'b': 2, 'd': 0}
>>> diff = DictDiffer(cur, past)
>>> print "Added:", diff.added()
    Added: set(['d'])
>>> print "Removed:", diff.removed()
    Removed: set(['c'])
>>> print "Changed:", diff.changed()
    Changed: set(['b'])
>>> print "Unchanged:", diff.unchanged()
    Unchanged: set(['a'])
```

Note: Originally posted as: http://stackoverflow.com/questions/1165352/fast-comparison-between-two-python-dictionary/1165552#1165552

added()

List items added (added keys).

changed()

List items changed (same keys with different values)

removed()

List items removed (removed keys).

unchanged()

List unchanged items (same keys, same values).

misc.utils.**NumPyTypes = ['uint8', 'int8', 'uint16', 'int16', 'uint32', 'int32', 'float**

NumPy types names

misc.utils.**PPTypes = ['B', 'b', 'H', 'h', 'I', 'i', 'f', 'd']**

Python pack types names.

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'u'	Py_UNICODE	Unicode character	2
'B'	unsigned char	int	1
'b'	signed char	int	1
'H'	unsigned short	int	2
'h'	signed short	int	2
'I'	unsigned int	long	2
'i'	signed int	int	2
'L'	unsigned long	long	4
'l'	signed long	int	4
'f'	float	float	4
'd'	double	float	8

See http://www.python.org/doc//current/library/struct.html and http://docs.python.org/2/library/array.html.

misc.utils.**TYPES = {'float': 'float32', 'float32': 'float32', 'float64': 'float64',**

Variables of useful types conversions

exception misc.utils.**UtilsError**(*message*, **arg*)

Bases: *misc.imports.Error*

exception misc.utils.**UtilsWarning**(*msg*, *expr=None*)

Bases: *misc.imports.MiscWarning*

misc.utils.**axis2**(*data*, *outaxis*, *inaxis=''*)

Perform axis (inv-)conversions of `array` structures; for 3D data only.

```
>>> output = axis2(data, outaxis, inaxis='')
```

Parameters **outaxis, inaxis** (*str*) – variables stating the input and output axis configuration; the ordering of the *x*, *y* and *z* axes is expected, like in strings from the list `['zyxlist', 'zyxarray', 'list', '-yxzarray', '-yxzlist', 'yxzlist', 'dict', 'zyxdict', 'array', 'nparray', 'yxzarray']`.

misc.utils.**basename**(*filepath*)

Extract the base name of a file.

```
>>> base = basename(filepath)
```

misc.utils.**datetimeformat**(***kwargs*)

Determine a datetime format from date and time formats, and their combination.

```
>>> dt = datetimeformat(**kwargs)
```

Keyword Arguments

- **date, time** (*str*) – strings specifying the date and time (written) formats; default is `'%Y-%m-%d'` for `date` and `'%H:%M:%S'` for `time`.
- **fmt** (*str*) – string specifying how `date` and `time` format are combined; default is: `'%sT%s'`, i.e. the char `'T'` separates date and time.

Returns **dt** (*str*) – string defining the format of datetime as `fmt.format(date, time)`; default to ISO 8601 format: `'%Y-%m-%dT%H:%M:%S'`.

`misc.utils.`**`encode`**(*string*)
Encode a unicode string: **dummy**.

`misc.utils.`**`exceptioninfo`**(*maxTBlevel=0*)
Get info about the last exception.

```
>>> info_str = exceptioninfo(maxTBlevel=0)
```

`misc.utils.`**`extname`**(*filepath*, *newext*)
Change the extension of a file.

```
>>> ext = extname(filepath)
```

`misc.utils.`**`fileinfo`**(*filepath*, ***kwargs*)
Provide various information about a file.

```
>>> info = fileinfo(filepath, **kwargs)
```

Keyword Arguments **`date,time,fmt`** (*`str`*) – see *`datetimeformat()`*.

Returns **info** (*dict*) – dictionary containing various information about `filepath`, namely its *`uuid`*, `size`, date of modification (`datemodified`), date of creation (`datecreated`), date of last access (`dateaccessed`), the owner's id (`ownerid`) and the owner's name (`ownername`).

`misc.utils.`**`filename`**(*filepath*)
Retrieve the... file name.

```
>>> name = filename(filepath)
```

`misc.utils.`**`filexists`**(*filepath*, *return_path=False*)
A case insensitive file existence checker.

```
>>> a, p = filexists(return_path)
```

Parameters **`return_path`** (*`bool, optional`*) – flag set to return the case sensitive path; def.: return_path=False.

Returns

- **a** (*bool*) – *True*/*False* answer to the existence.
- **p** (*str*) – optional full path to the case sensitive path.

`misc.utils.`**`fillpath`**(*rootpath*, *relfolderpath*, *relfilepath*)
Determine the absolute path of a file given by its relative path wrt a folder itself given by a relative path wrt root path.

```
>>> path = fillpath(rootpath, relfolderpath, relfilepath)
```

`misc.utils.`**`findfile`**(*directory*, *pattern*)
Find files in a directory and all subdirectories that match a given pattern.

```
>>> lst_files = findfile(directory, pattern)
```

misc.utils.**firstmatch**(*pattern*, *strings*)

Return the index of the first element in a list of strings that matches a given pattern, or the lenght of that list if no match was found.

```
>>> resi = firstmatch(pattern, strings)
```

Note: see also *firstnonmatch()*, *wherematch()*.

misc.utils.**firstnonmatch**(*pattern*, *list_string*)

Return the index of the first element in a list of strings that does not match a given pattern, or the lenght of that list if all strings match.

```
>>> resi = firstnonmatch(pattern, strings)
```

Note: see also *firstmatch()*, *wherematch()*.

misc.utils.**flat**(**structure*)

Flatten a structure recursively.

```
>>> res = flat(*structure)
```

misc.utils.**flatten**(*structure*, *key=''*, *path=''*, *flattened=None*, *indexed=False*)

Flatten any structure of any type (list, dict, tuple) and any depth in a recursive manner.

```
>>> res = flatten(structure, key="", path="", flattened=None)
```

misc.utils.**format2**(*data*, *outform*, *inform=None*)

Perform structure conversions from/to: `list`, `tuple`, `array` and `dict`.

```
>>> output = format2(data, outform, inform=None)
```

Parameters **outform** (*str*) – string specifying the desired output format; it can any string in `['array', 'dict', 'list', 'tuple']`.

misc.utils.**formatAndConvert**(***kwargs*)

Define an operator to perform a conversion from given type, format and axis.

```
>>> func = formatAndConvert('format_as'=None, 'type_as'=None,␣
↪KEYOFFSET=0,
                            'format_in'='zyxarray')
```

See also:

format2(), *type2()*, *axis2()*

misc.utils.**formatCheck**(*fmt*)

Determine if a variable designates a format; it can be a `type`, or a string corresponding either to a type name, or any element from the list `['zyxlist', 'zyxarray', 'list', '-yxzarray', '-yxzlist', 'yxzlist', 'dict', 'zyxdict', 'array', 'nparray', 'yxzarray']`.

```
>>> resp = formatCheck(fmt)
```

misc.utils.**formatraceback**(*trbk*, *maxTBlevel*)

Format traceback.

```
>>> trace_str = formatraceback(trbk, maxTBlevel)
```

misc.utils.**guesstype**(*filepath*, *option='type'*)

Coarsely retrieve a file MIME type.

```
>>> out = guesstype(filepath, option)
```

Parameters **option** (*str, optional*) – string setting the output to be returned; it is either the 'type', the 'subtype' or the 'encoding', or the whole 'mime' figure; def.: 'type'.

Returns **out** (*str, bool*) – MIME figure caracterizing the input file as stated by the 'guess_type' function of mimetypes; any of the results above depending on the input option; if the file does not exist, False is returned, if the MIME type is not recognized, True is returned.

misc.utils.**instanceUpdate**(*instance*, *flag*, *args*, ***kwargs*)

Update or retrieve an instance (field) of the data class through the parameters passed in kwargs.

```
>>> res = instanceUpdate(instance, flag, args, **kwargs)
```

Parameters

- **instance** (*object*) – instance of a class with appropriated fields (eg. 'self' is passed).
- **flag** (*bool*) – flag set to *True* when the given instance(s) is (are) updated from the parameters passed in kwargs when the parameters are not *None*, *False* when they are retrieved from *not None* instance(s);
- **args** (*dict*) – provide for each key set in kwargs, the name of the instance to consider.

Keyword Arguments **kwargs** (dict) – dictionary with values used to set/retrieve instances in/from the data.

Returns **res** (*dict*)

Note: This is equivalent to perform:

```
>>> key = args.keys()[i]
>>> val = kwargs.pop(key,None)
>>> if flag is False and val is None:
...     val = getattr(self,args[key])
>>> if flag is True and val is not None:
...     setattr(self,args[key]) = val
>>> res = val
```

for all keys i of args.

Examples

```
>>> param = { my_key: 'name_of_my_instance' }
>>> kwarg = { my_key: my_not_None_value }
```

Set the `name_of_my_instance` attribute of the class to `my_value` and return `my_not_None_value`:

```
>>> res = instanceUpdate(instance, True, param, **kwarg)
```

Retrieve `name_of_my_instance` attribute:

```
>>> res = instanceUpdate(instance, False, param, **kwarg)
```

`misc.utils.`**`isonlyNone`**(*data*, *deep=False*)

Determine if a structure (of any type and any depth) is made of None entries only. Return True if None entries are present only.

```
>>> resp = isonlyNone(data, deep=False)
```

`misc.utils.`**`isonlyScalar`**(*data*, *integer=False*)

Determine if some data is made of scalar entries only.

```
>>> resp = isonlyScalar(data, integer=False)
```

`misc.utils.`**`isrunning`**(*pid*)

Check that a process given by it PID is running.

```
>>> res = isrunning(pid)
```

`misc.utils.`**`istype`**(*instance*, *str_cls*)

Determine if a given instance is of a certain type defined by a string (instead of a `type` like in `isintance()`).

`misc.utils.`**`iterSubclasses`**(*cls*, *_seen=None*)

Generator over all subclasses of a given class, in depth first order.

```
>>> iter = iterSubclasses(cls, _seen=None)
```

Example

```
>>> list(iterSubclasses(int)) == [bool]
    True
>>> class A(object): pass
>>> class B(A): pass
>>> class C(A): pass
>>> class D(B,C): pass
>>> class E(D): pass
>>>
>>> for cls in iterSubclasses(A):
...     print(cls.__name__)
    B
    D
    E
```

(continues on next page)

(continued from previous page)

```
    C
>>> # get ALL (new-style) classes currently defined
>>> [cls.__name__ for cls in iterSubclasses(object)]
    ['type', ...'tuple', ...]
```

`misc.utils.`**`list2dict`**(*alist*, ***kwargs*)

Create a dictionary from multiple levels (recursive or nested) lists.

```
>>> new_dict = list2dict(alist, **kwargs)
```

Keyword Arguments **`nest,rec`** (*bool*) – keys used to specify if is the list represents a nested (default `nest=True`) or recursive (`rec=True`) dictionary.

Example

Transform a multiply nested list into a nested dictionary (of dictionaries of… of lists):

```
>>> alist = [['a', [['b', ['x', 1]], ['c', [2, 'y']]]], ['d', [1, 2, 3]]]
>>> list2dict(alist, nest=True)
    {'a': {'c': [2, 'y'], 'b': ['x', 1]}, 'd': [1, 2, 3]}
```

Transform a list into a 'recursive' dictionary (of dictionaries of dictionaries…):

```
>>> alist = ['a', ['b', ['x', '1'], ['y', '2']]]
>>> list2dict(alist, rec=True)
    {'a': {'b': {'y': '2', 'x': '1'}}}
```

`misc.utils.`**`makedir`**(*directory*)

Safe directory creation (mkdir) command.

```
>>> makedir(directory)
```

`misc.utils.`**`mergedict`**(**dicts*)

Recursively merge an arbitrary number of dictionaries.

```
>>> new_dict = mergedict(*dicts)
```

Example

```
>>> d1 = {'a': {'b': {'x': '1', 'y': '2'}}}
>>> d2 = {'a': {'c': {'gg': {'m': '3'},'xx': '4'}}}
>>> print mergedict(d1, d2)
    {'a': {'b': {'x': '1','y': '2'}, 'c': {'gg': {'m': '3'}, 'xx': '4'}}}
```

`misc.utils.`**`normcase`**(*filepath*)

Normalize case of pathname by making all characters lowercase and all slashes into backslashes.

```
>>> new = normcase(filepath)
```

`misc.utils.`**`normpath`**(*filepath*)

Normalize path, eliminating double slashes, etc.

```
>>> new = normpath(filepath)
```

misc.utils.**npy2ppt = {'float32': 'f', 'float64': 'd', 'int16': 'h', 'int32': 'i',**
Dictionary of Numpy -> Python pack types conversions

misc.utils.**offsetCheck**(*off*)
Check if a variable designates an offest, i.e. a positive integer.

```
>>> resp = offsetCheck(offset)
```

misc.utils.**picklines**(*thefile*, *whatlines*)
Utility for picking numbered lines from file.

misc.utils.**ppt2npy = {'B': 'uint8', 'H': 'uint16', 'I': 'uint32', 'b': 'int8', 'd':**
Dictionary of Python pack types -> Numpy conversions.

See http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html and http://docs.scipy.org/doc/numpy/reference/arrays.scalars.html.

misc.utils.**readcomments**(*opened_file*, *char_comment*)
Given a char designing comments, retrieve all comments from an opened file.

```
>>> readcomments = realpath(opened_file, char_comment)
```

misc.utils.**realpath**(*filepath*)
Return the absolute version of a path.

```
>>> real = realpath(filepath)
```

Note: os.path.realpath/os.path.abspath returns unexpected results on windows if filepath[-1]==':'

misc.utils.**remove**(*filepath*)
Safe remove (rm) command.

```
>>> remove(filepath)
```

misc.utils.**runcmd**(*cmd*, *format='s'*)
Run a command.

```
>>> exit_code, stdout, stderr = runcmd(cmd, format='s')
```

Parameters

- **cmd** (*str*) – command (inc arguments) to run.
- **format** (*str, optional*) – ['s','l'] for (string or list format); default is 's'.

Returns *Returns (exit_code,stdout,stderr)*

misc.utils.**subpatterns**(*strings*, *pattern*)
Return a list of string matching a certain pattern in a list of strings.

```
>>> res = subpatterns(strings, pattern)
```

misc.utils.**timestamp**(***kwargs*)
Return a timestamp.

```
>>> dtnow = timestamp(**kwargs)
```

Keyword Arguments **date,time,fmt** (*str*) – see *datetimeformat()*.

Returns **dtnow** (*str*) – timestamp estimated as the datetime representation of *now* (i.e. at the time the method is called).

`misc.utils.`**`type2`**(*data*, *dtype*, *view=None*)

Perform type conversions of various structures (either `list`, `tuple`, `array` or `dict`).

```
>>> output = type2(data, dtype, view=None)
```

Parameters **dtype** (*str*) – string indicating the desired output type; this can be any string in `['int32', 'object', 'int', 'float', 'uint8', 'int16', 'float64', 'long', 'uint16', 'uint32', 'float32']`.

`misc.utils.`**`typeCheck`**(*typ*)

Determine if a variable designates a type; it can be a `type`, or a string corresponding either to a type name, or any element from the list `['int32', 'object', 'int', 'float', 'uint8', 'int16', 'float64', 'long', 'uint16', 'uint32', 'float32']`.

```
>>> resp = typeCheck(typ)
```

`misc.utils.`**`typename`**(*instance*)

Return the class name of a given instance: nothing else than `instance.__class__.__name__`.

```
>>> name = typename(instance)
```

`misc.utils.`**`uncpath`**(*filepath*)

Convert file path to UNC (if on Windows): **dummy**.

`misc.utils.`**`uniqify`**(*stru*, *key=''*, *path=''*, *flattened=None*, *indexed=False*)

'Uniqify' and flatten the values of a structure of any type (list, dict, tuple) and any depth in a recursive manner.

```
>>> res = uniqify(structure, key="", path="", flattened=None)
```

`misc.utils.`**`uuid`**(*filepath*)

Generate a uuid reproducible based on filename.

```
>>> id = uuid(filepath)
```

`misc.utils.`**`wherematch`**(*pattern*, *strings*)

Return the indexes of the elements in a list of strings that match a given pattern.

```
>>> resi = wherematch(pattern, strings)
```

Note: see also *firstmatch()*, *firstnonmatch()*.

`misc.utils.`**`which`**(*name*, *return_first=True*, *flags=1*, *path=None*)

Search path for executable files with a given name.

```
>>> res = which(name, return_first=True, flags=os.F_OK | os.X_OK,␣
↪path=None)
```

Parameters

- **name** (*str*) – name for which to search.
- **return_first** (*bool, optional*) – flag to return the first executable found.
- **flags** – arguments to os.access on MS-Windows, the only flag that has any meaning is os.F_OK, any other flag will be ignored; default flag is `os.F_OK|os.X_OK`.

Returns **path** (*str*) – full path to the first matching file found or a list of the full paths to all files found, in the order in which they were found.

Notes

On newer versions of MS-Windows, the PATHEXT environment variable will be set to the list of file extensions for files considered executable. This will normally include things like ".EXE". This fuction will also find files with the given name ending with any of these extensions.

Derived mostly from U{http://code.google.com/p/waf/issues/detail?id=531} with additions from Brian Curtins patch - U{http://bugs.python.org/issue444582}

`misc.utils.`**`writable`**(*filepath*)

Determine if a temporary file can be written in the same directory as a given file.

```
>>> resp = writable(filepath)
```

`misc.utils.`**`yieldlines`**(*thefile*, *whatlines*)

Utility for yielding numbered lines from file.

`ext` Package

Module of external(ly developed) libraries/modules accessed in *KINKi*.

About

credits: grazzja

version: 0.1 – *since*: Wed Apr 09 17:19:42 2014

Description

Set of useful external libraries/modules that are also compatible with the *European Union Public Licence* (EUPL) of *KINKi* project.

Disclaimers

Massachussets Institute of Technology license

Permission is hereby granted, free of charge, to any
person obtaining a copy of this software and associated
documentation files (the "Software"), to deal in the
Software without restriction, including without limitation

the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the
Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
#
The above copyright notice and this permission notice
shall be included in all copies or substantial portions of
the Software.
#
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS
OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

BY-NC-SA Creative Commons license

Attribution-NonCommercial-ShareAlike 3.0
Attribution — You must give appropriate credit, provide a link
#
to the license, and indicate if changes were made. You may do
so in any reasonable manner, but not in any way that suggests
the licensor endorses you or your use.
#
NonCommercial — You may not use the material for commercial purposes.
ShareAlike — If you remix, transform, or build upon the material, you
must distribute your contributions under the same license as the original.

Simplified BSD (2-clause) license

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
#
* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
#
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

BSD 3-clause license

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
#
* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
* Neither the name of "dryan" nor Daniel Ryan nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.
#
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

Apache License, Version 2.0

Licensed under the Apache License, Version 2.0 (the "License"); you may
not use this file except in compliance with the License. You may obtain

a copy of the License at
#
http://www.apache.org/licenses/LICENSE-2.0
#
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
License for the specific language governing permissions and limitations
under the License.

Note

The list EUPL compatible open source licences is available at https://joinup.ec.europa.eu/software/page/eupl/eupl-compatible-open-source-licences.

Contents

`_datasource` Module

Numpy resource for generic data source file representation.

Description

Local copy/paste of numpy/lib/_datasource.py source file.

Forked from: https://github.com/numpy/numpy/blob/master/numpy/lib/_datasource.py. More information at http://docs.scipy.org/doc/numpy/reference/routines.io.html.

License

BSD 3 license – Copyright (c) 2005-2011, NumPy Developers.

`alchemyapi` Module

External SDK for AlchemyAPI.

Description

Wrapper forked from: https://github.com/AlchemyAPI/alchemyapi_python.

License

Apache license – Copyright 2010, AlchemyAPI.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

`browser` Module

External resource for browsing

Description

Forked from: https://github.com/pkrumins/xgoogle/blob/master/xgoogle/browser.py.

More information at http://www.catonmat.net/blog/python-library-for-google-search.

License

MIT license – Copyright (c) 2008 Peteris Krumins

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

`dal` Module

External library providing support for many relational databases.

Description

Forked from: https://github.com/web2py

This file contains the DAL support for many relational databases, including:

- SQLite & SpatiaLite
- MySQL
- Postgres
- Oracle
- MS SQL
- CouchDB (experimental)
- MongoDB (in progress)
- Google:nosql
- Google:sql

See further description at https://github.com/web2py/blob/master/gluon/dal.py.

License

GNU Lesser General Public License – Copyright (c) Massimo Di Pierro.

This file is part of the web2py Web Framework
License: LGPLv3 (http://www.gnu.org/licenses/lgpl.html)

Notes

This software **CANNOT BE DISTRIBUTED WITH KINKi** project owing to the incompatibility of GNU LGPL and EUPL licenses.

`dskey` Module

External module for unique key creation.

Description

Module forked from datastore, a unified API for multiple data stores.

Copied from https://github.com/datastore/datastore/blob/master/datastore/core/key.py and https://github.com/datastore/datastore/blob/master/datastore/core/util/fasthash.py.

A Key represents the unique identifier of an object. The Key scheme implemented in datastore is inspired by file systems and the Google App Engine key model. Keys are meant to be unique across a system. Keys are hierarchical, incorporating increasingly specific namespaces. Thus keys can be deemed 'children' or 'ancestors' of other keys. Also, every namespace can be parametrized to embed relevant object information. Specific namespace can include some fields

License

MIT license – Copyright (c) 2011, Juan Batiz-Benet.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

See also `dsquery`.

`dsquery` Module

External module for implementing a query operator.

Description

Module forked from datastore, a unified API for multiple data stores.

Copied from https://github.com/datastore/datastore/blob/master/datastore/core/query.py.

A query returns an iterable of objects matching criteria expressed in query. Implementations of query will be the largest differentiating factor amongst datastores. All datastores must implement query, even using query's worst case scenario.

License

MIT license – Copyright (c) 2011, Juan Batiz-Benet.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

See also `dskey`.

`extractor` Module

A module for including Tweet parsing in a class.

Description

Provide functions for the extraction and processing of usernames, lists, URLs and hashtags.

Copy/paste from twitter-text-py, functions extractor.py and unicode.py.

See https://github.com/dryan/twitter-text-py for more information.

License

BSD 3 license – Copyright (c) 2010, Daniel Ryan.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

`fbfql` Module

External library for FQL query on Facebook graph.

Description

Wrapper forked from https://github.com/ptwobrussell/Mining-the-Social-Web/blob/master/web_code/facebook_gae_demo_app/fql.py.

License

Simplified BSD license – Copyright (c) 2013, Matthew A. Russell – All rights reserved.

`fbquery` Module

`geolocation` Module

Library of geolocation tools used to represent coordinates on Earth, as suggested in http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates.

Description

Library forked from https://github.com/jfein/PyGeoTools/blob/master/geolocation.py.

See https://github.com/jfein/PyGeoTools.

License

This class is based from the code smaple in this paper:
http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates
The owner of that website, Jan Philip Matuschek, is the full owner of
his intellectual property. This class is simply a Python port of his very
useful Java code. All code written by Jan Philip Matuschek and ported herein
(which is all of this class) is owned by Jan Philip Matuschek.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

`geoplanetapi` Module

External library for interacting with Yahoo! GeoPlanet.

Description

Wrapper forked from https://github.com/ryanwi/python-geoplanet/blob/master/geoplanet/api.py

License

MIT license – Copyright (c) 2008, Mike Verdone

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

See also `yahooGeoPlanet`.

`geosocial_mapper` Module

External library to visualize connections over social networks: https://github.com/marcelcaraciolo/Geo-Friendship-Visualization.

Description

Wrapper forked from https://github.com/marcelcaraciolo/Geo-Friendship-Visualization/blob/master/gesocial_mapper.py

License

Simplified BSD license – Copyright (c), M. Caraciolo.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

`geoyahoo` Module

External library for using the geocoder of Yahoo! PlaceFinder.

Description

Wrapper forked from https://gist.github.com/tcabrol/1981588.

License

Created by Thomas Cabrol on 2012-02-08.

`heatmap` Module

External library to generate heat map images and animations from geographic data.

Description

Wrapper forked from http://www.sethoscope.net/heatmap/heatmap.py.

License

GNU Affero General Public License – Copyright (c) 2010, Seth Golub.

Copyright 2010 Seth Golub
http://www.sethoscope.net/heatmap/
#
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.
#
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Affero General Public License for more details.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

This software **CANNOT BE DISTRIBUTED WITH KINKi** project owing to the incompatibility of GNU AGPL and EUPL licenses.

`panoramioapi` Module

External library for interacting with Panoramio API.

Description

Wrapper forked from https://github.com/ramusus/python-panoramio/blob/master/panoramioapi/__init__.py

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

`platform` Module

Retrieve as much platform-identifying data as possible and make this information available via function APIs.

Description

Forked from http://www.egenix.com/files/python/platform.py

If called from the command line, it prints the platform information concatenated as single string to stdout. The output format is useable as part of a filename.

License

This module is maintained by Marc-Andre Lemburg.
Copyright (c) 1999-2000, Marc-Andre Lemburg.
Copyright (c) 2000-2010, eGenix.com Software GmbH.

`ratings` Module

External module for online movie ratings

Description

Forked from https://gist.github.com/tcabrol/2333075

License

Copyright (c) 2012 Thomas Cabrol. All rights reserved.

`regex` Module

A collection of regular expressions for parsing Tweet text.

Description

Forked from https://github.com/dryan/twitter-text-py.

License

BSD 3 license – Copyright (c) 2010, Daniel Ryan.

Notes

The regular expression list is frozen at load time to ensure immutability.

`string_distance_measures` Module

External library for Levenshtein-based string distance metrics for string comparisons.

Description

Forked from https://github.com/ianozsvald/string_distance_metrics.

A selection of unicode string distance metrics including uni/bi/trigrams, cosine similarity and edit distances.

Use methods of package python-Levenshtein.

Notes

Modifications to original code are inserted in between (`##KINKi` `##endKINKi`) markups.

See also package FuzzyWuzzy for operations on string matching/comparison.

`timeformatutils` Module

Library of utility functions for time formats conversion.

Description

Forked from https://github.com/csparpa/pyowm, the Python Open Weather Map API.

License

MIT license – Copyright (c) 2013, Claudio Sparpaglione.

`timezones` Module

External module for time zones definitions.

Description

Forked from https://github.com/sixohsix/twitter/blob/master/twitter/timezones.py

License

MIT license – Copyright (c) 2008, Mike Verdone.

`translate` Module

`weather` Module

External library of tools for weather reports from Yahoo! Weather

Description

Wrapper forked from https://github.com/tupton/python-yahoo-weather.

See http://www.thomasupton.com/blog/?p=202 for more information.

License

BY-NC-SA CC license – Written by Thomas Upton, with contributions from Chris Lasher.

`wordcloud` Module

`yahooGeoPlanet` Module

External library to get information from Yahoo! GeoPlanet.

Description

Wrapper forked from https://github.com/felHR85/yahooGeoPlanet/blob/master/src/yahooGeoPlanet.py.

License

MIT license – Copyright (c) 2013, Felipe Herranz.

Notes

Modifications to original code are inserted in between (`##KINKi ##endKINKi`) markups.

See also `geoplanetapi`.

2.8 Indices and tables

- genindex
- modindex
- search

Python Module Index

d

dbs, 135

e

ext, 184
ext._datasource, 187
ext.alchemyapi, 187
ext.browser, 187
ext.dal, 188
ext.dskey, 188
ext.dsquery, 189
ext.extractor, 189
ext.fbfql, 190
ext.geolocation, 190
ext.geoplanetapi, 190
ext.geosocial_mapper, 191
ext.geoyahoo, 191
ext.heatmap, 191
ext.panoramioapi, 192
ext.platform, 192
ext.ratings, 192
ext.regex, 193
ext.string_distance_measures, 193
ext.timeformatutils, 193
ext.timezones, 193
ext.weather, 194
ext.yahooGeoPlanet, 194

f

fetch, 91
fetch.facebook, 91
fetch.flickr, 100
fetch.foursquare, 101
fetch.linkedin, 110
fetch.panoramio, 118
fetch.twitter, 121

g

geoprocs, 147
geoprocs.lulc, 151
geoprocs.maps, 148
graphics, 155
graphics.base, 155

m

media, 77
media.alchemy, 77
media.facebook, 78
media.flickr, 78
media.foursquare, 82
media.geonames, 82
media.google, 83
media.linkedin, 85
media.owm, 85
media.panoramio, 87
media.twitter, 88
media.yahoo, 88
metadata, 3
misc, 156
misc.decorators, 156
misc.imports, 166
misc.pools, 172
misc.utils, 174

n

networks, 154
networks.cliques, 154

q

query, 136
query.api, 142
query.dbs, 137

s

semantics, 147
sketch, 5
sketch.location, 5
sketch.quotation, 43
sketch.status, 72

sketch.timing, 27
sketch.user, 69
sketch.weather, 58
statistics, 155
statistics.base, 156
store, 136

Index

Symbols

_Birdy (class in fetch.twitter), 132
_Facebook (class in fetch.facebook), 100
_Foursquare (class in fetch.foursquare), 109
_Linkedin (class in fetch.linkedin), 118
_Panoramioapi (class in fetch.panoramio), 121
_Pyfacebook (class in fetch.facebook), 100
_Pyfoursquare (class in fetch.foursquare), 109
_Pylinkedin (class in fetch.linkedin), 118
_Pytwitter (class in fetch.twitter), 135
_Tweepy (class in fetch.twitter), 133
_Twitter (class in fetch.twitter), 130
_Twitterapi (class in fetch.twitter), 135
_Twittersearch (class in fetch.twitter), 133
_Twython (class in fetch.twitter), 133
_clear_mapping() (query.api.API method), 142
_clear_mapping() (query.dbs.DBs method), 137
_extractor (class in sketch.quotation), 55
_geoCoderAPI (class in sketch.location), 21
_geoLocatorAPI (class in sketch.location), 22
_geoPlanetAPI (class in sketch.location), 20
_get_default_mapping() (query.api.API method), 142
_get_default_mapping() (query.dbs.DBs method), 137
_get_mapping() (query.api.API method), 142
_get_mapping() (query.dbs.DBs method), 137
_googleMapsAPI (class in sketch.location), 18
_googlePlacesAPI (class in sketch.location), 20
_googleTranslator (class in sketch.quotation), 55
_set_mapping() (query.api.API method), 143
_set_mapping() (query.dbs.DBs method), 137
_status (sketch.status.Status attribute), 73

A

access_methods (fetch.facebook.Facebook attribute), 92
access_methods (fetch.foursquare.Foursquare attribute), 101
access_methods (fetch.linkedin.Linkedin attribute), 110
access_methods (fetch.twitter.Twitter attribute), 122
access_token (fetch.facebook.Facebook attribute), 92
access_token (fetch.foursquare.Foursquare attribute), 101
access_token (fetch.linkedin.Linkedin attribute), 110
access_token (fetch.twitter.Twitter attribute), 122
access_token_secret (fetch.facebook.Facebook attribute), 92
access_token_secret (fetch.foursquare.Foursquare attribute), 101
access_token_secret (fetch.linkedin.Linkedin attribute), 110
access_token_secret (fetch.twitter.Twitter attribute), 122
access_urls (fetch.facebook.Facebook attribute), 92
access_urls (fetch.foursquare.Foursquare attribute), 102
access_urls (fetch.linkedin.Linkedin attribute), 111
access_urls (fetch.twitter.Twitter attribute), 122
accuracy (sketch.location.Location attribute), 12
add_tile_layer() (geoprocs.maps.Folmap method), 149
added() (misc.utils.DictDiffer method), 175
address (sketch.location.Location attribute), 12
ALL_DBS (in module dbs), 135
ALL_MEDIA (in module media), 77
altitude (sketch.location.Location attribute), 12
ang_unit (sketch.location.Location attribute), 12
ang_units_to() (sketch.location._geoLocatorAPI class method), 22
API (class in query.api), 142
api_application_defs (fetch.facebook.Facebook attribute), 92
api_application_defs (fetch.foursquare.Foursquare attribute), 102
api_application_defs (fetch.linkedin.Linkedin attribute), 111
api_application_defs (fetch.panoramio.Panoramio attribute), 119
api_application_defs (fetch.twitter.Twitter attribute), 122
API_KEY (in module media.alchemy), 78
API_KEY (in module media.facebook), 78
API_KEY (in module media.flickr), 82
API_KEY (in module media.foursquare), 82
API_KEY (in module media.google), 83
API_KEY (in module media.linkedin), 85

API_KEY (in module media.owm), 87
API_KEY (in module media.twitter), 88
API_KEY (in module media.yahoo), 90
API_KEY_2 (in module media.alchemy), 78
api_query_keys (fetch.facebook.Facebook attribute), 92
api_query_keys (fetch.foursquare.Foursquare attribute), 102
api_query_keys (fetch.linkedin.Linkedin attribute), 111
api_query_keys (fetch.panoramio.Panoramio attribute), 119
api_query_keys (fetch.twitter.Twitter attribute), 122
API_SECRET (in module media.facebook), 78
API_SECRET (in module media.flickr), 82
API_SECRET (in module media.linkedin), 85
API_SECRET (in module media.twitter), 88
API_TOKEN (in module media.facebook), 78
API_TOKEN (in module media.linkedin), 85
API_TOKEN (in module media.twitter), 88
API_TOKEN_SECRET (in module media.linkedin), 85
API_TOKEN_SECRET (in module media.twitter), 88
ApiError, 146
apis (fetch.facebook.Facebook attribute), 92
apis (fetch.foursquare.Foursquare attribute), 102
apis (fetch.linkedin.Linkedin attribute), 111
apis (fetch.panoramio.Panoramio attribute), 119
apis (fetch.twitter.Twitter attribute), 122
ApiWarning, 146
auth (fetch.facebook.Facebook attribute), 92
auth (fetch.foursquare.Foursquare attribute), 102
auth (fetch.linkedin.Linkedin attribute), 111
auth (fetch.twitter.Twitter attribute), 122
axis2() (in module misc.utils), 176

B

basename() (in module misc.utils), 176
bbox (sketch.location.Location attribute), 12
bbox2latlon() (sketch.location.Geographer class method), 6
bbox2polygon() (sketch.location.Geographer class method), 7
bboxintersection() (sketch.location.Geographer class method), 8
bboxintersects() (sketch.location.Geographer class method), 8
bboxunion() (sketch.location.Geographer class method), 8
bboxwithin() (sketch.location.Geographer class method), 8
bounding_locations() (sketch.location._geoLocatorAPI method), 22
bounding_locations_from() (sketch.location._geoLocatorAPI class method), 23
build_map() (geoprocs.maps.KMLmap class method), 150

C

cashtag() (sketch.quotation._extractor method), 56
centroid() (sketch.location._geoLocatorAPI class method), 23
changed() (misc.utils.DictDiffer method), 175
check() (sketch.quotation.Interpreter class method), 44
choropleth() (geoprocs.maps.Folmap method), 149
Chronometer (class in sketch.timing), 27
class_decorator() (in module misc.decorators), 163
classes (misc.imports.Module attribute), 168
clean() (sketch.location.Location method), 12
clean() (sketch.quotation.Quotation method), 47
clean() (sketch.status.Status method), 73
clean() (sketch.timing.Timing method), 37
clean() (sketch.user.User method), 69
clean() (sketch.weather.Weather method), 64
clear() (fetch.facebook.Facebook method), 92
clear() (fetch.foursquare.Foursquare method), 102
clear() (fetch.linkedin.Linkedin method), 111
clear() (fetch.twitter.Twitter method), 122
clear() (query.api.API method), 143
clear() (query.dbs.DBs method), 138
clear() (sketch.location.Location method), 12
clear() (sketch.quotation.Quotation method), 48
clear() (sketch.status.Status method), 74
clear() (sketch.timing.Timing method), 37
clear() (sketch.user.User method), 69
clear() (sketch.weather.Weather method), 64
clearAPI() (fetch.facebook.Facebook method), 92
clearAPI() (fetch.foursquare.Foursquare method), 102
clearAPI() (fetch.linkedin.Linkedin method), 111
clearAPI() (fetch.panoramio.Panoramio method), 119
clearAPI() (fetch.twitter.Twitter method), 123
clearAuthentication() (fetch.facebook.Facebook method), 92
clearAuthentication() (fetch.foursquare.Foursquare method), 102
clearAuthentication() (fetch.linkedin.Linkedin method), 111
clearAuthentication() (fetch.twitter.Twitter method), 123
clearBBox() (sketch.location.Location method), 12
clearCondition() (sketch.weather.Weather method), 64
clearCredential() (fetch.facebook.Facebook method), 92
clearCredential() (fetch.foursquare.Foursquare method), 102
clearCredential() (fetch.linkedin.Linkedin method), 111
clearCredential() (fetch.twitter.Twitter method), 123
clearForecast() (sketch.weather.Weather method), 64
clearIdKey() (query.api.API method), 143
clearIdKey() (query.dbs.DBs method), 138
clearKey() (fetch.facebook.Facebook method), 92

clearKey() (fetch.foursquare.Foursquare method), 102
clearKey() (fetch.linkedin.Linkedin method), 111
clearKey() (fetch.twitter.Twitter method), 123
clearLanguage() (sketch.quotation.Quotation method), 48
clearLatLonRad() (sketch.location.Location method), 12
clearLocale() (sketch.quotation.Quotation method), 48
clearLocation() (sketch.weather.Weather method), 64
clearMapFilter() (query.api.API method), 143
clearMapFilter() (query.dbs.DBs method), 138
clearMapSearch() (query.api.API method), 143
clearMapSearch() (query.dbs.DBs method), 138
clearMapStream() (query.api.API method), 143
clearMapStream() (query.dbs.DBs method), 138
clearName() (sketch.user.User method), 69
clearPlace() (sketch.location.Location method), 12
clearQuery() (query.api.API method), 143
clearQuery() (query.dbs.DBs method), 138
clearQuote() (sketch.quotation.Quotation method), 48
clearSince() (sketch.timing.Timing method), 37
clearSpan() (sketch.timing.Timing method), 37
clearStatus() (sketch.status.Status method), 74
clearSWNE() (sketch.location.Location method), 12
clearTiming() (sketch.weather.Weather method), 64
clearToken() (fetch.facebook.Facebook method), 92
clearToken() (fetch.foursquare.Foursquare method), 102
clearToken() (fetch.linkedin.Linkedin method), 111
clearToken() (fetch.twitter.Twitter method), 123
clearTranslation() (sketch.quotation.Quotation method), 48
clearUntil() (sketch.timing.Timing method), 38
clearURL() (fetch.facebook.Facebook method), 92
clearURL() (fetch.foursquare.Foursquare method), 102
clearURL() (fetch.linkedin.Linkedin method), 111
clearURL() (fetch.twitter.Twitter method), 123
client_id (fetch.facebook.Facebook attribute), 92
client_id (fetch.foursquare.Foursquare attribute), 102
client_id (fetch.linkedin.Linkedin attribute), 111
client_id (fetch.panoramio.Panoramio attribute), 119
client_id (fetch.twitter.Twitter attribute), 123
CLIENT_ID (in module media.foursquare), 82
CLIENT_ID (in module media.google), 83
client_key (fetch.facebook.Facebook attribute), 93
client_key (fetch.foursquare.Foursquare attribute), 102
client_key (fetch.linkedin.Linkedin attribute), 111
client_key (fetch.twitter.Twitter attribute), 123
client_secret (fetch.facebook.Facebook attribute), 93
client_secret (fetch.foursquare.Foursquare attribute), 102
client_secret (fetch.linkedin.Linkedin attribute), 111
client_secret (fetch.twitter.Twitter attribute), 123
Cliques (class in networks.cliques), 154
cliques (networks.cliques.Cliques attribute), 154
CliquesError, 155
CliquesWarning, 155
code() (sketch.location.Geographer class method), 8
CODER (sketch.location.Geographer attribute), 6
compute() (networks.cliques.Cliques method), 154
condition (sketch.weather.Weather attribute), 64
Conditioning (class in misc.decorators), 162
connections() (fetch.facebook.Facebook method), 93
connections() (fetch.twitter._Twitter class method), 130
content (sketch.quotation.Quotation attribute), 48
convert_angle_units() (sketch.location._geoLocatorAPI class method), 23
convert_distance_units() (sketch.location._geoLocatorAPI class method), 24
convert_time_units() (sketch.timing.Chronometer class method), 28
coordinates (sketch.location.Location attribute), 12
copy() (sketch.location.Location method), 12
copy() (sketch.quotation.Quotation method), 48
copy() (sketch.status.Status method), 74
copy() (sketch.timing.Timing method), 38
copy() (sketch.user.User method), 69
copy() (sketch.weather.Weather method), 64
country (sketch.location.Location attribute), 13
country_id (sketch.location.Location attribute), 13
create() (networks.cliques.Cliques method), 155
create_image() (geoprocs.maps.Netmap method), 149
create_map() (geoprocs.maps.KMLmap class method), 150
create_map() (geoprocs.maps.Netmap method), 150
create_map_dir() (geoprocs.maps.Netmap method), 150
create_placemark() (geoprocs.maps.KMLmap class method), 150
credentials (fetch.facebook.Facebook attribute), 93
credentials (fetch.foursquare.Foursquare attribute), 102
credentials (fetch.linkedin.Linkedin attribute), 111
credentials (fetch.twitter.Twitter attribute), 123
cursor() (fetch.twitter._Tweepy class method), 134

D

dance() (fetch.facebook.Facebook method), 93
dance() (fetch.foursquare.Foursquare method), 102
dance() (fetch.linkedin.Linkedin method), 111
dance() (fetch.twitter.Twitter method), 123
datetime() (sketch.timing.Chronometer class method), 28
DATETIME_UNITS (sketch.timing.Chronometer attribute), 28
datetimeformat() (in module misc.utils), 176
DBs (class in query.dbs), 137
dbs (module), 135
dbs.mongodb (module), 136
DECIMAL_PRECISION (sketch.location.Geographer attribute), 6
decorate() (misc.decorators.Docstring class method), 157
decorator_postcondition() (in module misc.decorators), 166

decorator_precondition() (in module misc.decorators), 165
DEF_APP_DESC (in module media), 77
DEF_APP_NAME (in module media), 77
DEF_DB_HOST (in module dbs), 135
DEF_DB_PORT (in module dbs), 135
DEF_OWNER_EMAIL (in module media), 77
DEF_OWNER_NAME (in module dbs), 135
DEF_OWNER_NAME (in module media), 77
DEFAULT_MEDIA (in module media), 77
deg2dps() (sketch.location._geoLocatorAPI class method), 24
deg2rad() (sketch.location._geoLocatorAPI class method), 24
depth() (misc.imports.Module class method), 169
detect() (sketch.quotation.Interpreter class method), 44
detect() (sketch.quotation.Quotation method), 48
dict() (query.api.Query method), 146
DictDiffer (class in misc.utils), 174
display() (fetch.twitter.Twitter static method), 123
distance() (sketch.location.Geographer class method), 9
distance_matrix() (sketch.location._googleMapsAPI method), 18
distance_to() (sketch.location._geoLocatorAPI method), 24
distance_to_from() (sketch.location._geoLocatorAPI class method), 24
diversity() (fetch.twitter.Twitter static method), 123
Docstring (class in misc.decorators), 157
domain (fetch.facebook.Facebook attribute), 93
domain (fetch.foursquare.Foursquare attribute), 103
domain (fetch.linkedin.Linkedin attribute), 112
domain (fetch.panoramio.Panoramio attribute), 119
domain (fetch.twitter.Twitter attribute), 123
dps2deg() (sketch.location._geoLocatorAPI class method), 25
dps2rad() (sketch.location._geoLocatorAPI class method), 25
dst_unit (sketch.location.Location attribute), 13

E

EARTH_RADIUS_AVERAGE (sketch.location.Geographer attribute), 6
EARTH_RADIUS_EQUATOR (sketch.location.Geographer attribute), 6
EARTH_RADIUS_MEAN (sketch.location.Geographer attribute), 6
EARTH_RADIUS_POLAR (sketch.location.Geographer attribute), 6
email (sketch.user.User attribute), 70
encode() (in module misc.utils), 177
enquiries_map() (query.dbs.DBs method), 138
enquiry() (query.api.Query method), 146
entities() (fetch.twitter.Twitter static method), 123
entities() (sketch.quotation._extractor method), 56
entities_common() (fetch.twitter.Twitter static method), 123
entities_count (fetch.twitter.Twitter attribute), 123
Error, 166
estimate_radius_WGS84() (sketch.location._geoLocatorAPI class method), 25
exceptioninfo() (in module misc.utils), 177
ext (module), 184
ext._datasource (module), 187
ext.alchemyapi (module), 187
ext.browser (module), 187
ext.dal (module), 188
ext.dskey (module), 188
ext.dsquery (module), 189
ext.extractor (module), 189
ext.fbfql (module), 190
ext.geolocation (module), 190
ext.geoplanetapi (module), 190
ext.geosocial_mapper (module), 191
ext.geoyahoo (module), 191
ext.heatmap (module), 191
ext.panoramioapi (module), 192
ext.platform (module), 192
ext.ratings (module), 192
ext.regex (module), 193
ext.string_distance_measures (module), 193
ext.timeformatutils (module), 193
ext.timezones (module), 193
ext.weather (module), 194
ext.yahooGeoPlanet (module), 194
extname() (in module misc.utils), 177
extract() (sketch.quotation.Interpreter class method), 45
extract() (sketch.quotation.Quotation method), 49
EXTRACTOR (sketch.quotation.Interpreter attribute), 44

F

Facebook (class in fetch.facebook), 91
fetch (module), 91
fetch.facebook (module), 91
fetch.flickr (module), 100
fetch.foursquare (module), 101
fetch.linkedin (module), 110
fetch.panoramio (module), 118
fetch.twitter (module), 121
fetch_data() (geoprocs.lulc.ITagApi method), 153
fileinfo() (in module misc.utils), 177
filename() (in module misc.utils), 177
filexists() (in module misc.utils), 177
fillpath() (in module misc.utils), 177
filter() (networks.cliques.Cliques method), 155
filter() (query.api.API method), 143
filter() (query.dbs.DBs method), 138

filter() (sketch.location.Location method), 13
filter() (sketch.quotation.Quotation method), 49
filter() (sketch.status.Status method), 74
filter() (sketch.timing.Timing method), 38
filter() (sketch.user.User method), 70
filter() (sketch.weather.Weather method), 64
findfile() (in module misc.utils), 177
firstmatch() (in module misc.utils), 178
firstname (sketch.user.User attribute), 70
firstnonmatch() (in module misc.utils), 178
fit_bounds() (geoprocs.maps.Folmap method), 149
flat() (in module misc.utils), 178
flatten() (in module misc.utils), 178
Flickr (class in fetch.flickr), 100
followers() (fetch.twitter._Tweepy class method), 134
followers() (fetch.twitter._Twitter class method), 131
followers() (fetch.twitter._Twython class method), 133
Folmap (class in geoprocs.maps), 148
forecast (sketch.weather.Weather attribute), 65
format() (misc.decorators.Docstring class method), 157
format2() (in module misc.utils), 178
format_class() (misc.decorators.Docstring class method), 158
format_method() (misc.decorators.Docstring class method), 159
formatAndConvert() (in module misc.utils), 178
formatCheck() (in module misc.utils), 178
formatraceback() (in module misc.utils), 179
Foursquare (class in fetch.foursquare), 101
friends() (fetch.facebook.Facebook method), 93
friends() (fetch.twitter._Tweepy class method), 134
friends() (fetch.twitter._Twitter class method), 131
friends() (fetch.twitter._Twython class method), 133
from_degrees() (sketch.location._geoLocatorAPI class method), 25
from_dict() (query.api.Query class method), 146
from_dps() (sketch.location._geoLocatorAPI class method), 25
from_radians() (sketch.location._geoLocatorAPI class method), 25

G

geocode() (fetch.twitter.Twitter static method), 123
geocode() (sketch.location._googleMapsAPI method), 19
geocode() (sketch.location._googlePlacesAPI class method), 20
geocode() (sketch.location.Location method), 13
Geographer (class in sketch.location), 5
geoprocs (module), 147
geoprocs.lulc (module), 151
geoprocs.maps (module), 148
get() (fetch.facebook.Facebook method), 93
get() (fetch.foursquare.Foursquare method), 103
get() (fetch.linkedin.Linkedin method), 112
get() (fetch.twitter.Twitter method), 123
get() (query.api.API method), 143
get() (query.dbs.DBs method), 138
get() (sketch.location.Location method), 13
get() (sketch.quotation.Quotation method), 50
get() (sketch.status.Status method), 74
get() (sketch.timing.Timing method), 38
get() (sketch.user.User method), 70
get() (sketch.weather.Weather method), 65
get_url() (geoprocs.lulc.ITagApi method), 153
getAPI() (fetch.facebook.Facebook method), 93
getAPI() (fetch.foursquare.Foursquare method), 103
getAPI() (fetch.linkedin.Linkedin method), 112
getAPI() (fetch.panoramio.Panoramio method), 119
getAPI() (fetch.twitter.Twitter method), 124
getAuthentication() (fetch.facebook.Facebook method), 93
getAuthentication() (fetch.foursquare.Foursquare method), 103
getAuthentication() (fetch.linkedin.Linkedin method), 112
getAuthentication() (fetch.twitter.Twitter method), 124
getBBox() (sketch.location.Location method), 13
getCondition() (sketch.weather.Weather method), 65
getCredential() (fetch.facebook.Facebook method), 94
getCredential() (fetch.foursquare.Foursquare method), 103
getCredential() (fetch.linkedin.Linkedin method), 112
getCredential() (fetch.twitter.Twitter method), 124
getDefaultAPI() (fetch.facebook.Facebook method), 94
getDefaultAPI() (fetch.foursquare.Foursquare method), 103
getDefaultAPI() (fetch.linkedin.Linkedin method), 112
getDefaultAPI() (fetch.panoramio.Panoramio method), 119
getDefaultAPI() (fetch.twitter.Twitter method), 124
getDefaultAuthentication() (fetch.facebook.Facebook method), 94
getDefaultAuthentication() (fetch.foursquare.Foursquare method), 104
getDefaultAuthentication() (fetch.linkedin.Linkedin method), 113
getDefaultAuthentication() (fetch.twitter.Twitter method), 124
getDefaultCredential() (fetch.facebook.Facebook method), 94
getDefaultCredential() (fetch.foursquare.Foursquare method), 104
getDefaultCredential() (fetch.linkedin.Linkedin method), 113
getDefaultCredential() (fetch.twitter.Twitter method), 125
getDefaultKey() (fetch.facebook.Facebook method), 94
getDefaultKey() (fetch.foursquare.Foursquare method), 104

getDefaultKey() (fetch.linkedin.Linkedin method), 113
getDefaultKey() (fetch.twitter.Twitter method), 125
getDefaultMapEnquiry() (query.dbs.DBs method), 139
getDefaultMapFilter() (query.api.API method), 144
getDefaultMapFilter() (query.dbs.DBs method), 139
getDefaultMapSearch() (query.api.API method), 144
getDefaultMapSearch() (query.dbs.DBs method), 139
getDefaultMapStream() (query.api.API method), 144
getDefaultMapStream() (query.dbs.DBs method), 139
getDefaultToken() (fetch.facebook.Facebook method), 95
getDefaultToken() (fetch.foursquare.Foursquare method), 104
getDefaultToken() (fetch.linkedin.Linkedin method), 113
getDefaultToken() (fetch.twitter.Twitter method), 125
getDefaultURL() (fetch.facebook.Facebook method), 95
getDefaultURL() (fetch.foursquare.Foursquare method), 104
getDefaultURL() (fetch.linkedin.Linkedin method), 113
getDefaultURL() (fetch.twitter.Twitter method), 125
getForecast() (sketch.weather.Weather method), 65
getIdKey() (query.api.API method), 144
getIdKey() (query.dbs.DBs method), 139
getKey() (fetch.facebook.Facebook method), 95
getKey() (fetch.foursquare.Foursquare method), 105
getKey() (fetch.linkedin.Linkedin method), 114
getKey() (fetch.twitter.Twitter method), 126
getLanguage() (sketch.quotation.Quotation method), 50
getLatLonRad() (sketch.location.Location method), 14
getLocale() (sketch.quotation.Quotation method), 50
getLocation() (sketch.weather.Weather method), 65
getLocationsFromPhotos() (fetch.flickr.Flickr method), 100
getMapEnquiry() (query.dbs.DBs method), 139
getMapFilter() (query.api.API method), 144
getMapFilter() (query.dbs.DBs method), 139
getMapSearch() (query.api.API method), 144
getMapSearch() (query.dbs.DBs method), 139
getMapStream() (query.api.API method), 145
getMapStream() (query.dbs.DBs method), 140
getName() (sketch.user.User method), 70
getPhotosByPage() (fetch.flickr.Flickr method), 100
getPlace() (sketch.location.Location method), 14
getQuery() (query.api.API method), 145
getQuery() (query.dbs.DBs method), 140
getQuote() (sketch.quotation.Quotation method), 51
getSince() (sketch.timing.Timing method), 38
getSpan() (sketch.timing.Timing method), 39
getStatus() (sketch.status.Status method), 75
getSWNE() (sketch.location.Location method), 14
getTiming() (sketch.weather.Weather method), 65
getToken() (fetch.facebook.Facebook method), 95
getToken() (fetch.foursquare.Foursquare method), 105
getToken() (fetch.linkedin.Linkedin method), 114
getToken() (fetch.twitter.Twitter method), 126
getTranslation() (sketch.quotation.Quotation method), 51
getUntil() (sketch.timing.Timing method), 39
getURL() (fetch.facebook.Facebook method), 96
getURL() (fetch.foursquare.Foursquare method), 105
getURL() (fetch.linkedin.Linkedin method), 114
getURL() (fetch.twitter.Twitter method), 126
getUserInfo() (fetch.flickr.Flickr method), 100
GMAPS (sketch.location.Geographer attribute), 6
GOSLATE (sketch.quotation.Interpreter attribute), 44
GOTRANS (sketch.quotation.Interpreter attribute), 44
GPLACES (sketch.location.Geographer attribute), 6
grain (sketch.location.Location attribute), 15
graph (networks.cliques.Cliques attribute), 155
graphics (module), 155
graphics.base (module), 155
gt() (sketch.timing.Chronometer class method), 31
gte() (sketch.timing.Chronometer class method), 31
guesstype() (in module misc.utils), 179

H

hashtag() (sketch.quotation._extractor method), 56
Heatmap (class in geoprocs.maps), 148

I

id (sketch.location.Location attribute), 15
id (sketch.quotation.Quotation attribute), 52
id (sketch.status.Status attribute), 75
id (sketch.timing.Timing attribute), 39
id (sketch.user.User attribute), 70
id (sketch.weather.Weather attribute), 66
idkey (query.api.API attribute), 145
idkey (query.dbs.DBs attribute), 140
imports() (in module misc.imports), 169
imports() (misc.imports.Module method), 169
info_by_id() (fetch.twitter._Twitter class method), 131
infoMessage() (in module misc.imports), 167
instanceUpdate() (in module misc.utils), 179
Interpreter (class in sketch.quotation), 44
iso6391tolanguage() (sketch.quotation.Interpreter static method), 46
isonlyNone() (in module misc.utils), 180
isonlyScalar() (in module misc.utils), 180
isrunning() (in module misc.utils), 180
istype() (in module misc.utils), 180
ITagApi (class in geoprocs.lulc), 151
items() (sketch.location.Location method), 15
items() (sketch.quotation.Quotation method), 52
items() (sketch.status.Status method), 75
items() (sketch.timing.Timing method), 39
items() (sketch.user.User method), 70
items() (sketch.weather.Weather method), 66
iterSubclasses() (in module misc.utils), 180

K

keep_in_front() (geoprocs.maps.Folmap method), 149
keys (fetch.facebook.Facebook attribute), 96
keys (fetch.foursquare.Foursquare attribute), 106
keys (fetch.linkedin.Linkedin attribute), 114
keys (fetch.twitter.Twitter attribute), 126
keys() (sketch.location.Location method), 15
keys() (sketch.quotation.Quotation method), 52
keys() (sketch.status.Status method), 75
keys() (sketch.timing.Timing method), 40
keys() (sketch.user.User method), 71
keys() (sketch.weather.Weather method), 66
kimports() (in module misc.imports), 171
KMLmap (class in geoprocs.maps), 150
KWARGS_API_FILTER (in module media.facebook), 78
KWARGS_API_FILTER (in module media.flickr), 78
KWARGS_API_FILTER (in module media.foursquare), 82
KWARGS_API_FILTER (in module media.linkedin), 85
KWARGS_API_FILTER (in module media.panoramio), 88
KWARGS_API_FILTER (in module media.twitter), 88
KWARGS_API_SEARCH (in module media.facebook), 78
KWARGS_API_SEARCH (in module media.flickr), 78
KWARGS_API_SEARCH (in module media.foursquare), 82
KWARGS_API_SEARCH (in module media.linkedin), 85
KWARGS_API_SEARCH (in module media.panoramio), 88
KWARGS_API_SEARCH (in module media.twitter), 88

L

lang (sketch.quotation.Quotation attribute), 52
LANGS (sketch.quotation.Interpreter attribute), 44
language (sketch.quotation.Quotation attribute), 52
LANGUAGES (sketch.quotation.Interpreter attribute), 44
languagetoiso6391() (sketch.quotation.Interpreter static method), 46
lapply() (misc.decorators.Slicing static method), 160
lastname (sketch.user.User attribute), 71
latdeg2m() (sketch.location._geoLocatorAPI class method), 26
latitude (sketch.location.Location attribute), 15
latlon2bbox() (sketch.location.Geographer class method), 10
latm2deg() (sketch.location._geoLocatorAPI class method), 26
lclean() (misc.decorators.Slicing static method), 161
Linkedin (class in fetch.linkedin), 110
list2dict() (in module misc.utils), 181
lngdeg2m() (sketch.location._geoLocatorAPI class method), 26
lngm2deg() (sketch.location._geoLocatorAPI class method), 26
loadToken() (fetch.facebook.Facebook method), 96
loadToken() (fetch.foursquare.Foursquare method), 106
loadToken() (fetch.linkedin.Linkedin method), 115
loadToken() (fetch.twitter.Twitter method), 126
locale (sketch.quotation.Quotation attribute), 52
locality (sketch.location.Location attribute), 15
locate() (sketch.location._geoPlanetAPI method), 20
Location (class in sketch.location), 11
location (sketch.status.Status attribute), 73
LOCATOR (sketch.location.Geographer attribute), 6
Logger (class in misc.imports), 166
longitude (sketch.location.Location attribute), 15
lt() (sketch.timing.Chronometer class method), 32
lte() (sketch.timing.Chronometer class method), 32
LULCError, 154
LULCWarning, 154

M

M_TO (sketch.location.Geographer attribute), 6
makedir() (in module misc.utils), 181
map_tasks() (misc.pools.Parallel static method), 172
mappings (query.api.API attribute), 145
mappings (query.dbs.DBs attribute), 140
MapReduce (class in misc.pools), 173
max_id_query() (fetch.twitter.Twitter static method), 127
MAX_LATITUDE (sketch.location.Geographer attribute), 6
MAX_LONGITUDE (sketch.location.Geographer attribute), 6
media (fetch.facebook.Facebook attribute), 96
media (fetch.foursquare.Foursquare attribute), 106
media (fetch.linkedin.Linkedin attribute), 115
media (fetch.panoramio.Panoramio attribute), 120
media (fetch.twitter.Twitter attribute), 127
media (module), 77
media (query.api.API attribute), 145
media (query.dbs.DBs attribute), 140
media.alchemy (module), 77
media.facebook (module), 78
media.flickr (module), 78
media.foursquare (module), 82
media.geonames (module), 82
media.google (module), 83
media.linkedin (module), 85
media.owm (module), 85
media.panoramio (module), 87
media.twitter (module), 88
media.yahoo (module), 88
medianame (fetch.facebook.Facebook attribute), 96
medianame (fetch.foursquare.Foursquare attribute), 106
medianame (fetch.linkedin.Linkedin attribute), 115
medianame (fetch.panoramio.Panoramio attribute), 120

medianame (fetch.twitter.Twitter attribute), 127
mention() (sketch.quotation._extractor method), 57
mergedict() (in module misc.utils), 181
Message (class in misc.imports), 166
metaclass_decorator() (in module misc.decorators), 165
metaclass_maker() (in module misc.decorators), 165
MetaclassProxy (class in misc.decorators), 163
metadata (module), 3
Meteorologist (class in sketch.weather), 58
method_decorator() (in module misc.decorators), 164
mimports() (in module misc.imports), 171
mimports() (misc.imports.Module method), 169
MIN_LATITUDE (sketch.location.Geographer attribute), 5
MIN_LONGITUDE (sketch.location.Geographer attribute), 6
misc (module), 156
misc.decorators (module), 156
misc.imports (module), 166
misc.pools (module), 172
misc.utils (module), 174
MiscWarning, 166
Module (class in misc.imports), 168
modules (misc.imports.Module attribute), 168
multigraph (networks.cliques.Cliques attribute), 155
mutual_friends() (fetch.facebook.Facebook method), 96

N

name (sketch.user.User attribute), 71
ne (sketch.location.Location attribute), 15
Netmap (class in geoprocs.maps), 149
networks (module), 154
networks.cliques (module), 154
noaa() (sketch.weather.Meteorologist class method), 59
normcase() (in module misc.utils), 181
normpath() (in module misc.utils), 181
npy2ppt (in module misc.utils), 182
NumPyTypes (in module misc.utils), 175

O

objects() (fetch.facebook.Facebook method), 96
offsetCheck() (in module misc.utils), 182
openweather() (sketch.weather.Meteorologist class method), 59

P

Panoramio (class in fetch.panoramio), 119
Parallel (class in misc.pools), 172
PARAMETERS (geoprocs.lulc.ITagApi attribute), 153
partition() (misc.pools.MapReduce method), 174
path (geoprocs.maps.Netmap attribute), 149
permissions (fetch.facebook.Facebook attribute), 96
permissions (fetch.foursquare.Foursquare attribute), 106
permissions (fetch.linkedin.Linkedin attribute), 115
permissions (fetch.panoramio.Panoramio attribute), 120
permissions (fetch.twitter.Twitter attribute), 127
picklines() (in module misc.utils), 182
place (sketch.location.Location attribute), 15
place2locality() (sketch.location._geoPlanetAPI method), 20
place2woeid() (sketch.location._geoPlanetAPI method), 21
PLANET (sketch.location.Geographer attribute), 6
polygon2bbox() (sketch.location.Geographer class method), 10
post_new() (misc.decorators.MetaclassProxy class method), 163
postal_id (sketch.location.Location attribute), 15
ppt2npy (in module misc.utils), 182
PPTypes (in module misc.utils), 175
pre_new() (misc.decorators.MetaclassProxy class method), 163
PRETTY_PARAMETERS (geoprocs.lulc.ITagApi attribute), 153

Q

Query (class in query.api), 146
query (module), 136
query (query.api.API attribute), 145
query (query.dbs.DBs attribute), 140
query() (fetch.facebook.Facebook method), 97
query() (fetch.foursquare.Foursquare method), 106
query() (fetch.linkedin.Linkedin method), 115
query() (fetch.panoramio.Panoramio method), 120
query() (fetch.twitter.Twitter method), 127
query.api (module), 142
query.dbs (module), 137
Quotation (class in sketch.quotation), 47
quotation (sketch.status.Status attribute), 73
quote (sketch.quotation.Quotation attribute), 52

R

rad2deg() (sketch.location._geoLocatorAPI class method), 26
rad2dps() (sketch.location._geoLocatorAPI class method), 26
radius (sketch.location.Location attribute), 16
raiseMessage() (in module misc.imports), 167
rapply() (misc.decorators.Slicing static method), 161
readcomments() (in module misc.utils), 182
realpath() (in module misc.utils), 182
relnet (geoprocs.maps.Netmap attribute), 149
remove() (in module misc.utils), 182
removed() (misc.utils.DictDiffer method), 175
render() (geoprocs.maps.Folmap method), 149
reply() (sketch.quotation._extractor method), 57
request() (fetch.facebook.Facebook method), 97
requires (misc.imports.Module attribute), 168

returnLocations() (fetch.flickr.Flickr method), 100
retweet_attributions() (fetch.twitter.Twitter static method), 128
retweet_filter() (fetch.twitter.Twitter static method), 128
retweets() (fetch.twitter._Twython class method), 133
reverse() (sketch.location.Geographer class method), 10
round() (sketch.location.Geographer class method), 11
RT_PATTERNS (fetch.twitter.Twitter attribute), 122
run_tasks() (misc.pools.Parallel static method), 173
runcmd() (in module misc.utils), 182

S

saveToken() (fetch.facebook.Facebook method), 97
saveToken() (fetch.foursquare.Foursquare method), 107
saveToken() (fetch.linkedin.Linkedin method), 116
saveToken() (fetch.twitter.Twitter method), 128
search() (fetch.facebook.Facebook method), 98
search() (fetch.foursquare.Foursquare method), 107
search() (fetch.linkedin.Linkedin method), 116
search() (fetch.panoramio.Panoramio method), 120
search() (fetch.twitter.Twitter method), 128
search() (query.api.API method), 145
search() (query.dbs.DBs method), 140
searchForTargets() (fetch.flickr.Flickr method), 101
SEC_TO (sketch.timing.Chronometer attribute), 27
semantics (module), 147
set() (fetch.facebook.Facebook method), 98
set() (fetch.foursquare.Foursquare method), 107
set() (fetch.linkedin.Linkedin method), 116
set() (fetch.twitter.Twitter method), 128
set() (query.api.API method), 145
set() (query.dbs.DBs method), 140
set() (sketch.location.Location method), 16
set() (sketch.quotation.Quotation method), 52
set() (sketch.status.Status method), 75
set() (sketch.timing.Timing method), 40
set() (sketch.user.User method), 71
set() (sketch.weather.Weather method), 66
setAPI() (fetch.facebook.Facebook method), 98
setAPI() (fetch.foursquare.Foursquare method), 108
setAPI() (fetch.linkedin.Linkedin method), 117
setAPI() (fetch.panoramio.Panoramio method), 121
setAPI() (fetch.twitter.Twitter method), 129
setAuthentication() (fetch.facebook.Facebook method), 98
setAuthentication() (fetch.foursquare.Foursquare method), 108
setAuthentication() (fetch.linkedin.Linkedin method), 117
setAuthentication() (fetch.twitter.Twitter method), 129
setBBox() (sketch.location.Location method), 16
setCondition() (sketch.weather.Weather method), 67
setCredential() (fetch.facebook.Facebook method), 99
setCredential() (fetch.foursquare.Foursquare method), 108
setCredential() (fetch.linkedin.Linkedin method), 117
setCredential() (fetch.twitter.Twitter method), 129
setForecast() (sketch.weather.Weather method), 67
setIdKey() (query.api.API method), 145
setIdKey() (query.dbs.DBs method), 141
setKey() (fetch.facebook.Facebook method), 99
setKey() (fetch.foursquare.Foursquare method), 108
setKey() (fetch.linkedin.Linkedin method), 117
setKey() (fetch.twitter.Twitter method), 129
setLanguage() (sketch.quotation.Quotation method), 53
setLatLonRad() (sketch.location.Location method), 16
setLocale() (sketch.quotation.Quotation method), 53
setLocation() (sketch.weather.Weather method), 67
setMapEnquiry() (query.dbs.DBs method), 141
setMapFilter() (query.api.API method), 146
setMapFilter() (query.dbs.DBs method), 141
setMapSearch() (query.api.API method), 146
setMapSearch() (query.dbs.DBs method), 141
setMapStream() (query.api.API method), 146
setMapStream() (query.dbs.DBs method), 141
setName() (sketch.user.User method), 71
setPlace() (sketch.location.Location method), 17
setQuery() (query.api.API method), 146
setQuery() (query.dbs.DBs method), 141
setQuote() (sketch.quotation.Quotation method), 53
setSince() (sketch.timing.Timing method), 40
setSinceNow() (sketch.timing.Timing method), 41
setSpan() (sketch.timing.Timing method), 41
setStatus() (sketch.status.Status method), 75
setSWNE() (sketch.location.Location method), 17
setTiming() (sketch.weather.Weather method), 68
setToken() (fetch.facebook.Facebook method), 99
setToken() (fetch.foursquare.Foursquare method), 109
setToken() (fetch.linkedin.Linkedin method), 118
setToken() (fetch.twitter.Twitter method), 130
setTranslation() (sketch.quotation.Quotation method), 54
setUntil() (sketch.timing.Timing method), 41
setUntilNow() (sketch.timing.Timing method), 41
setURL() (fetch.facebook.Facebook method), 99
setURL() (fetch.foursquare.Foursquare method), 109
setURL() (fetch.linkedin.Linkedin method), 118
setURL() (fetch.twitter.Twitter method), 130
since (sketch.timing.Timing attribute), 41
since() (sketch.timing.Chronometer class method), 32
sketch (module), 5
sketch.location (module), 5
sketch.quotation (module), 43
sketch.status (module), 72
sketch.timing (module), 27
sketch.user (module), 69
sketch.weather (module), 58
Slicing (class in misc.decorators), 160

span (sketch.timing.Timing attribute), 42
span() (sketch.timing.Chronometer class method), 34
statistics (module), 155
statistics.base (module), 156
Status (class in sketch.status), 73
status (query.api.API attribute), 146
status (query.dbs.DBs attribute), 141
status (sketch.location.Location attribute), 17
status (sketch.quotation.Quotation attribute), 54
status (sketch.status.Status attribute), 76
status (sketch.timing.Timing attribute), 42
status (sketch.user.User attribute), 72
status (sketch.weather.Weather attribute), 68
status() (fetch.linkedin.Linkedin static method), 118
status() (fetch.twitter.Twitter static method), 130
status_by_id() (fetch.twitter._Twitter class method), 132
status_show() (fetch.twitter._Twitter class method), 132
STATUS_UNIQIFY (query.api.API attribute), 142
STATUS_UNIQIFY (query.dbs.DBs attribute), 137
store (module), 136
stream() (fetch.facebook.Facebook method), 100
stream() (fetch.foursquare.Foursquare method), 109
stream() (fetch.linkedin.Linkedin method), 118
stream() (fetch.twitter._Tweepy class method), 134
stream() (fetch.twitter.Twitter method), 130
street (sketch.location.Location attribute), 17
stringify() (networks.cliques.Cliques static method), 155
subpatterns() (in module misc.utils), 182
sw (sketch.location.Location attribute), 17

T

tag() (sketch.quotation._extractor method), 57
text (sketch.quotation.Quotation attribute), 54
timedelta() (sketch.timing.Chronometer class method), 35
TIMEDELTA_UNITS (sketch.timing.Chronometer attribute), 28
timestamp() (in module misc.utils), 182
Timing (class in sketch.timing), 37
Timing (sketch.status.Status.timing attribute), 73
timing (sketch.timing.Timing attribute), 42
TIMING_UNITS (sketch.timing.Chronometer attribute), 28
today (sketch.weather.Weather attribute), 68
togeojson() (fetch.twitter.Twitter static method), 130
tomorrow (sketch.weather.Weather attribute), 68
topology() (sketch.location.Location method), 18
translate() (sketch.quotation.Interpreter class method), 46
translate() (sketch.quotation.Quotation method), 54
trends() (fetch.twitter._Tweepy class method), 134
trends() (fetch.twitter._Twitter class method), 132
trends() (fetch.twitter._Twython class method), 133
Twitter (class in fetch.twitter), 121
type2() (in module misc.utils), 183
typeCheck() (in module misc.utils), 183
typename() (in module misc.utils), 183
TYPES (in module misc.utils), 176

U

unchanged() (misc.utils.DictDiffer method), 175
uncpath() (in module misc.utils), 183
uniqify() (in module misc.utils), 183
units_to() (sketch.location._geoLocatorAPI class method), 27
units_to() (sketch.timing.Chronometer class method), 36
unix_since (sketch.timing.Timing attribute), 42
unix_until (sketch.timing.Timing attribute), 42
until (sketch.timing.Timing attribute), 43
until() (sketch.timing.Chronometer class method), 36
update() (query.api.API method), 146
update() (query.dbs.DBs method), 141
update() (sketch.location._geoCoderAPI method), 22
update() (sketch.location.Location method), 18
update() (sketch.quotation.Quotation method), 55
update() (sketch.status.Status method), 76
update() (sketch.timing.Timing method), 43
update() (sketch.user.User method), 72
update() (sketch.weather.Weather method), 68
updateMapEnquiry() (query.dbs.DBs method), 141
URL (geoprocs.lulc.ITagApi attribute), 153
url() (sketch.quotation._extractor method), 57
url_fql (fetch.facebook.Facebook attribute), 100
User (class in sketch.user), 69
User (sketch.status.Status.user attribute), 73
user (sketch.user.User attribute), 72
user() (sketch.quotation._extractor method), 58
user_id (sketch.user.User attribute), 72
user_info() (fetch.twitter.Twitter static method), 130
UtilsError, 176
UtilsWarning, 176
uuid() (in module misc.utils), 183

V

values() (sketch.location.Location method), 18
values() (sketch.quotation.Quotation method), 55
values() (sketch.status.Status method), 76
values() (sketch.timing.Timing method), 43
values() (sketch.user.User method), 72
values() (sketch.weather.Weather method), 69

W

WAPI (sketch.weather.Meteorologist attribute), 58
warnMessage() (in module misc.imports), 167
Weather (class in sketch.weather), 63
weather_at() (sketch.weather.Meteorologist class method), 60
weather_location() (sketch.weather.Meteorologist class method), 60

weather_timing() (sketch.weather.Meteorologist class method), 61
weathercom() (sketch.weather.Meteorologist class method), 62
WGS84_SEMIAXIS_a (sketch.location.Geographer attribute), 6
WGS84_SEMIAXIS_b (sketch.location.Geographer attribute), 6
wherematch() (in module misc.utils), 183
which() (in module misc.utils), 183
woe_id (sketch.location.Location attribute), 18
woeid2bbox() (sketch.location._geoPlanetAPI method), 21
woeid2centroid() (sketch.location._geoPlanetAPI method), 21
woeid2place() (sketch.location._geoPlanetAPI method), 21
worker() (misc.pools.Parallel static method), 173
world (sketch.location._geoPlanetAPI attribute), 21
WOWM (sketch.weather.Meteorologist attribute), 59
wrappers (fetch.facebook.Facebook attribute), 100
wrappers (fetch.foursquare.Foursquare attribute), 109
wrappers (fetch.linkedin.Linkedin attribute), 118
wrappers (fetch.panoramio.Panoramio attribute), 121
wrappers (fetch.twitter.Twitter attribute), 130
writable() (in module misc.utils), 184

Y

yahooweather() (sketch.weather.Meteorologist class method), 63
YCLIENT (sketch.weather.Meteorologist attribute), 59
yieldlines() (in module misc.utils), 184