pyowm Documentation

Claudio Sparpaglione

Jan 03, 2019

Contents

Welcome to PyOWM's documentation!

CHAPTER 1

What is PyOWM?

PyOWM is a client Python wrapper library for OpenWeatherMap web APIs. It allows quick and easy consumption of OWM data from Python applications via a simple object model and in a human-friendly fashion.

CHAPTER 2

What APIs can I access with PyOWM?

With PyOWM you can interact programmatically with the following OpenWeatherMap web APIs:

- **Weather API v2.5, offering**
    - current weather data
    - weather forecasts
    - weather history
- **Agro API v1.0**, offering polygon editing, soil data, satellite imagery search and download
- **Air Pollution API v3.0**, offering data about CO, O3, NO2 and SO2
- **UV Index API v3.0**, offering data about Ultraviolet exposition
- **Stations API v3.0**, allowing to create and manage meteostation and publish local weather measurements
- **Weather Alerts API v3.0**, allowing to set triggers on weather conditions and areas and poll for spawned alerts

And You can also get **image tiles** for several map layers provided by OWM

The documentation of OWM APIs can be found on the OWM Website

CHAPTER 3

Supported environments and Python versions

PyOWM runs on Windows, Linux and MacOS.

PyOWM runs on:

- Python 2.7
- Python 3.4+

Please notice that **support for Python 2.x will eventually be dropped** - check details

PyOWM also integrates with Django 1.10+ models, but that integration might have issues (contributions are welcome)

CHAPTER 4

Installation

4.1 pip

The easiest method of all:

```
$ pip install pyowm
```

4.2 Get the lastest development version

You can install the development trunk with _pip_:

```
$ pip install git+https://github.com/csparpa/pyowm.git@develop
```

but be aware that it might not be stable!

4.3 setuptools

You can install from source using _setuptools_: either download a release from GitHub or just take the latest main branch), then:

```
$ unzip <zip archive>   # or tar -xzf <tar.gz archive>
$ cd pyowm-x.y.z
$ python setup.py install
```

The .egg will be installed into the system-dependent Python libraries folder

4.4 Distribution packages

- On Windows you have installers
- On ArchLinux you can install PyOWM with the Yaourt package manager, run:

```
Yaourt -S python2-owm  # Python 2.7 (https://aur.archlinux.org/packages/
↪python-owm)
Yaourt -S python-owm   # Python 3.x (https://aur.archlinux.org/packages/
↪python2-owm)
```

CHAPTER 5

PyOWM v2 usage documentation

Here are some usage examples for the different OWM APIs: these examples and guides refer to PyOWM library versions belonging to the 2.x stream

5.1 Weather API examples

5.1.1 Import the PyOWM library

As simple as:

```
>>> from pyowm import OWM
```

5.1.2 Create global OWM object

Use your OWM API key if you have one (read here on how to obtain an API key). By default, if you don't specify which API subscription type you want to use, a free-subscription OWM global object is instantiated:

```
>>> API_key = 'G097IueS-9xN712E'
>>> owm = OWM(API_key)
```

Of course you can change your API key at a later time if you need:

```
>>> owm.get_API_key()
'G09_7IueS-9xN712E'
>>> owm.set_API_key('6Lp$0UY220_HaSB45')
```

The same happens with the language: you can speficy in which language the OWM Weather API will return textual data of weather queries. Language is specified by passing its corresponding two-characters string, eg: `es`, `sk`, etc. The default language is English (`en`):

```
>>> owm_en = OWM()                # default language is English
>>> owm_ru = OWM(language='ru')  # Russian
```

You can obtain the OWM global object related to a specific OWM Weather API version, just specify it after the API key parameter(check before that the version is supported!):

```
>>> owm = OWM(API_key='abcdef', version='2.5')
```

If you don't specify an API version number, you'll be provided with the OWM object that represents the latest available OWM Weather API version.

Advanced users might want to inject into the library a specific configuration: this can be done by injecting the Python path of your personal configuration module as a string into the library instantiation call like this:

```
>>> owm = OWM(API_key='abcdef', version='2.5', config_module='mypackage.mysubpackage.
↪myconfigmodule')
```

Be careful! You must provide a well-formatted configuration module for the library to work properly and your module must be in your PYTHONPATH. More on configuration modules formatting can be found here.

5.1.3 Using a paid (pro) API key subscription

If you purchased a pro subscription on the OWM Weather API, you can instantiate the global OWM like this:

```
>>> owm = pyowm.OWM('abcdef', subscription_type='pro')
```

When instantiating paid subscription OWM objects, you must provide an API key.

5.1.4 OWM Weather API version 2.5 usage examples

Setting a local cache provider

The PyOWM library comes with a built-in support for local caches: OWM Weather API reponses can be cached in order to save time and bandwidth. The default configuration uses no cache, however the library contains a built-in simple LRU cache implementation that can be plugged in by changing the `configuration25.py` module and specifying a `LRUCache` class instance:

```
...
# Cache provider to be used
from pyowm.caches.lrucache import LRUCache
cache = LRUCache()
...
```

By using the `configuration25.py` module, it is also possible to leverage external cache providers module, provided that they implement the interface that is expected by the library code.

Getting currently observed weather for a specific location.

Querying for current weather is simple: provide an `OWM` object with the location you want the current weather be looked up for and the job is done. You can specify the location either by passing its toponym (eg: "London"), the city ID (eg: 2643741) or its geographic coordinates (lon/lat):

```
obs = owm.weather_at_place('London,GB')                    # Toponym
obs = owm.weather_at_id(2643741)                           # City ID
obs = owm.weather_at_coords(-0.107331,51.503614)           # lat/lon
```

An `Observation` object will be returned, containing weather info about the location matching the toponym/ID/coordinates you provided. Be precise when specifying locations!

Retrieving city ID for a location

City IDs can be retrieved using a registry:

```
reg = owm.city_id_registry()
reg.ids_for('London')          # [ (123, 'London', 'GB'), (456, 'London', 'MA'), (789,
↪'London', 'WY')]
reg.locations_for("London")  # gives a list of Location instances
reg.geopoints_for("London")  # gives a list of pyowm.utils.geo.Point objects
```

You can pass the retrieved IDs with `owm.weather_at_id` method.

As multiple locations with the same name exist in different states, the registry comes with support for narrowing down queries on specific countries…

```
london = reg.ids_for('London', country='GB')                   # [ (123, 'London, GB') ]
london_loc = reg.locations_for('London', country='GB')         # [ <Location obj> ]
london_geopoint = reg.geopoints_for('London', country='GB')  # [ <Point obj> ]
```

… as well as for changing the type of matches between the provided string and the locations' toponyms:

```
reg.ids_for("london", matching='exact')   # literal matching
reg.ids_for("london", matching='nocase') # case-insensitive
reg.ids_for("london", matching='like')    # substring search
```

Also remember that the registry can provide the geopoints (instances of `pyowm.utils.geo.Point`) corresponding to the searched toponyms:

```
list_of_geopoints = reg.geopoints_for('London', country='GB')
```

Please refer to the SW API docs for further detail.

Currently observed weather extended search

You can query for currently observed weather:

- for all the places whose name equals the toponym you provide (use `search='accurate'`)
- for all the places whose name contains the toponym you provide (use `search='like'`)
- for all the places whose lon/lat coordinates are in the surroundings of the lon/lat couple you provide

In all cases, a list of `Observation` objects is returned, each one describing the weather currently observed in one of the places matching the search. You can control how many items the returned list will contain by using the `limit` parameter.

Examples:

```
# Find observed weather in all the "London"s in the world
obs_list = owm.weather_at_places('London', 'accurate')
# As above but limit result items to 3
obs_list = owm.weather_at_places('London',searchtype='accurate',limit=3)

# Find observed weather for all the places whose name contains the word "London"
obs_list = owm.weather_at_places('London', 'like')
# As above but limit result items to 5
obs_list = owm.weather_at_places('London',searchtype='like', 5)

# Find observed weather for all the places in the surroundings of lon=-2.15,lat=57
obs_list = owm.weather_around_coords(-2.15, 57)
# As above but limit result items to 8
obs_list = owm.weather_around_coords(-2.15, 57, limit=8)
```

Getting data from Observation objects

`Observation` objects store two useful objects: a `Weather` object that contains the weather-related data and a `Location` object that describes the location the weather data is provided for.

If you want to know when the weather observation data have been received, just call:

```
>>> obs.get_reception_time()                          # UNIX GMT time
1379091600L
>>> obs.get_reception_time(timeformat='iso')          # ISO8601
'2013-09-13 17:00:00+00'
>>> obs.get_reception_time(timeformat='date')         # datetime.datetime instance
datetime.datetime(2013, 09, 13, 17, 0, 0, 0)
```

You can retrieve the `Weather` object like this:

```
>>> w = obs.get_weather()
```

and then access weather data using the following methods:

```
>>> w.get_reference_time()                            # get time of observation in␣
↪GMT UNIXtime
1377872206L
>>> w.get_reference_time(timeformat='iso')            # ...or in ISO8601
'2013-08-30 14:16:46+00'
>>> w.get_reference_time(timeformat='date')           # ...or as a datetime.datetime␣
↪object
datetime.datetime(2013, 08, 30, 14, 16, 46, 0)

>>> w.get_clouds()                                    # Get cloud coverage
65

>>> w.get_rain()                                      # Get rain volume
{'3h': 0}

>>> w.get_snow()                                      # Get snow volume
{}

>>> w.get_wind()                                      # Get wind degree and speed
{'deg': 59, 'speed': 2.660}
```

(continues on next page)

(continued from previous page)

```
>>> w.get_humidity()                                        # Get humidity percentage
67

>>> w.get_pressure()                                        # Get atmospheric pressure
{'press': 1009, 'sea_level': 1038.381}

>>> w.get_temperature()                                     # Get temperature in Kelvin
{'temp': 293.4, 'temp_kf': None, 'temp_max': 297.5, 'temp_min': 290.9}
>>> w.get_temperature(unit='celsius')                       # ... or in Celsius degs
>>> w.get_temperature('fahrenheit')                         # ... or in Fahrenheit degs

>>> w.get_status()                                          # Get weather short status
'clouds'
>>> w.get_detailed_status()                                # Get detailed weather status
'Broken clouds'

>>> w.get_weather_code()                                    # Get OWM weather condition␣
↪code
803

>>> w.get_weather_icon_name()                               # Get weather-related icon name
'02d'

>>> w.get_weather_icon_url()                               # Get weather-related icon URL
'http://openweathermap.org/img/w/02d.png'

>>> w.get_sunrise_time()                                    # Sunrise time (GMT UNIXtime␣
↪or ISO 8601)
1377862896L
>>> w.get_sunset_time('iso')                                # Sunset time (GMT UNIXtime or␣
↪ISO 8601)
'2013-08-30 20:07:57+00'
```

Support to weather data interpretation and lists of OWM weather conditions, codes and icons can be found here.

As said, `Observation` objects also contain a `Location` object with info about the weather location:

```
>>> l = obs.get_location()
>>> l.get_name()
'London'
>>> l.get_lon()
-0.12574
>>> l.get_lat()
51.50863
>>> l.get_ID()
2643743
```

The last call returns the OWM city ID of the location - refer to the OWM API documentation for details.

Getting weather forecasts

The OWM Weather API currently provides weather forecasts that are sampled :

- every 3 hours
- every day (24 hours)

The 3h forecasts are provided for a streak of 5 days since the request time and daily forecasts are provided for a maximum streak of 14 days since the request time (but also shorter streaks can be obtained).

You can query for 3h forecasts for a location using:

```
# Query for 3 hours weather forecast for the next 5 days over London
>>> fc = owm.three_hours_forecast('London,uk')
```

You can query for daily forecasts using:

```
# Query for daily weather forecast for the next 14 days over London
>>> fc = owm.daily_forecast('London,uk')
```

and in this case you can limit the amount of days the weather forecast streak will contain by using the `limit` parameter:

```
# Daily weather forecast just for the next 6 days over London
>>> fc = owm.daily_forecast('London,uk', limit=6)
```

Both of the above calls return a `Forecaster` object. `Forecaster` objects contain a `Forecast` object, which has all the information about your weather forecast. If you need to manipulate the latter, just go with:

```
>>> f = fc.get_forecast()
```

A `Forecast` object encapsulates the `Location` object relative to the forecast and a list of `Weather` objects:

```
# When has the forecast been received?
>>> f.get_reception_time()                           # UNIX GMT time
1379091600L
>>> f.get_reception_time('iso')                      # ISO8601
'2013-09-13 17:00:00+00'
>>> f.get_reception_time('date')                     # datetime.datetime instance
datetime.datetime(2013, 09, 13, 17, 0, 0, 0)

# Which time interval for the forecast?
>>> f.get_interval()
'daily'

# How many weather items are in the forecast?
>>> len(f)
20

# Get Location
>>> f.get_location()
<pyowm.location.Location object at 0x01921DF0>
```

Once you obtain a `Forecast` object, reading the forecast data is easy - you can get the whole list of `Weather` objects or you can use the built-in iterator:

```
# Get the list of Weather objects...
>>> lst = f.get_weathers()

# ...or iterate directly over the Forecast object
>>> for weather in f:
      print (weather.get_reference_time('iso'),weather.get_status())
('2013-09-14 14:00:00+0','Clear')
('2013-09-14 17:00:00+0','Clear')
('2013-09-14 20:00:00+0','Clouds')
```

The `Forecaster` class provides a few convenience methods to inspect the weather forecasts in a human-friendly fashion. You can - for example - ask for the GMT time boundaries of the weather forecast data:

```
# When in time does the forecast begin?
>>> fc.when_starts()                                    # UNIX GMT time
1379090800L
>>> fc.when_starts('iso')                               # ISO8601
'2013-09-13 16:46:40+00'
>>> fc.when_starts('date')
datetime.datetime(2013, 09, 13, 16, 46, 40, 0)          # datetime.datetime instance

# ...and when will it end?
>>> fc.when_ends()                                      # UNIX GMT time
1379902600L
>>> fc.when_ends('iso')                                 # ISO8601
'2013-09-23 02:16:40+00'
>>> fc.when_ends('date')                                # datetime.datetime instance
datetime.datetime(2013, 09, 13, 16, 46, 40, 0)
```

In example, you can ask the `Forecaster` instance to tell which is the weather forecast for a specific point in time. You can specify this time using a UNIX timestamp, an ISO8601-formatted string or a Python `datetime.datetime` object (all times must will be handled as GMT):

```
# Tell me the weather for tomorrow at this hour
>>> from datetime import datetime
>>> date_tomorrow = datetime(2013, 9, 19, 12, 0)
>>> str_tomorrow = "2013-09-19 12:00+00"
>>> unix_tomorrow = 1379592000L
>>> fc.get_weather_at(date_tomorrow)
<weather.Weather at 0x00DF75F7>
>>> fc.get_weather_at(str_tomorrow)
<weather.Weather at 0x00DF75F7>
>>> fc.get_weather_at(unix_tomorrow)
<weather.Weather at 0x00DF75F7>
```

You will be provided with the `Weather` sample that lies closest to the time that you specified. Of course this will work only if the specified time is covered by the forecast! Otherwise, you will be prompted with an error:

```
>>> fc.get_weather_at("1492-10-12 12:00:00+00")
pyowm.exceptions.not_found_error.NotFoundError: The searched item was not found.
Reason: Error: the specified time is not included in the weather coverage range
```

Keep in mind that you can leverage the convenience `timeutils` module's functions to quickly build datetime objects:

```
>>> from pyowm import timeutils
>>> timeutils.tomorrow()                                # Tomorrow at this hour
datetime.datetime(2013, 9, 19, 12, 0)
>>> timeutils.yesterday(23, 27)                         # Yesterday at 23:27
datetime.datetime(2013, 9, 19, 12, 0)
>>> timeutils.next_three_hours()
datetime.datetime(2013, 9, 18, 15, 0)                   # 3 hours from now
>>> t = datetime.datetime(2013, 19, 27, 8, 47, 0)
>>> timeutils.next_three_hours(t)
datetime.datetime(2013, 19, 27, 11, 47, 0)              # 3 hours from a specific␣
↪datetime
```

Other useful convenicence methods in class `Forecaster` are:

```
# Will it rain, be sunny, foggy or snow during the covered period?
>>> fc.will_have_rain()
True
>>> fc.will_have_sun()
True
>>> fc.will_have_fog()
False
>>> fc.will_have_clouds()
False
>>> fc.will_have_snow()
False

# Will it be rainy, sunny, foggy or snowy at the specified GMT time?
time = "2013-09-19 12:00+00"
>>> fc.will_be_rainy_at(time)
False
>>> fc.will_be_sunny_at(time)
True
>>> fc.will_be_foggy_at(time)
False
>>> fc.will_be_cloudy_at(time)
False
>>> fc.will_be_snowy_at(time)
False
>>> fc.will_be_sunny_at(0L)           # Out of weather forecast coverage
pyowm.exceptions.not_found_error.NotFoundError: The searched item was not found.
Reason: Error: the specified time is not included in the weather coverage range

# List the weather elements for which the condition will be:
# rain, sun, fog and snow
>>> fc.when_rain()
[<weather.Weather at 0x00DB22F7>,<weather.Weather at 0x00DB2317>]
>>> fc.when_sun()
[<weather.Weather at 0x00DB62F7>]
>> fc.when_clouds()
[<weather.Weather at 0x00DE22F7>]
>>> fc.when_fog()
[<weather.Weather at 0x00DC22F7>.]
>>> fc.when_snow()
[]                                   # It won't snow: empty list

# Get weather for the hottest, coldest, most humid, most rainy, most snowy
# and most windy days in the forecast
>>> fc.most_hot()
<weather.Weather at 0x00DB67D9>
>>> fc.most_cold()
<weather.Weather at 0x00DB62F7>
>>> fc.most_humid()
<weather.Weather at 0x00DB62F7>
>>> fc.most_rainy()
<weather.Weather at 0x00DB62F7>
>>> fc.most_snowy()
None                                 # No snow in the forecast
>>> fc.most_windy()
<weather.Weather at 0x00DB62F7>
```

When calling the `will_be_*_at()` methods you can specify either a UNIX timestamp, a `datetime.datetime` object or an ISO8601-formatted string (format: “YYYY-MM-DD HH:MM:SS+00”). A boolean value will be returned, telling if the queried weather condition will apply to the time you specify (the check will be performed on the *Weather* object of the forecast which is closest in time to the time value that you provided).

When calling the `when_*()` methods you will be provided with a sublist of the `Weather` objects list in into the `Forecaster` instance, with items having as weather condition the one the method queries for.

Note on weather forecast items reference timestamps

Sometimes - due to caching on the OWM API side - the weather objects returned inside a *Forecast* object may refer to timestamps in the recent past. In order to remove those outdated weather items from the forecast, you can do:

```
>>> fcst.actualize()
```

Note on weather history

Weather history retrieval is a *paid OWM API feature*.

Getting weather history on a location

Weather history on a specific toponym can be retrieved using:

```
>>> owm.weather_history_at_place('London,uk')
[ <weather.Weather at 0x00BF81A2>, <weather.Weather at 0x00BF81C8>, ... ]
```

A list of `Weather` objects is returned. You can can specify a time window in which you want the results to be filtered:

```
>>> owm.weather_history_at_place('London,uk', start=1379090800L, end=1379099800L)
>>> owm.weather_history_at_place('London,uk', '2013-09-13 16:46:40+00', '2013-09-13␣
↪19:16:40+00')
>>> from datetime import datetime
>>> owm.weather_history_at_place('London,uk', datetime(2013, 9, 13, 16, 46, 40),␣
↪datetime(2013, 9, 13, 19, 16, 40))
```

The time boundaries can be expressed either as a UNIX timestamp, a *datetime.datetime* object or an ISO8601-formatted string (format: “YYYY-MM-DD HH:MM:SS+00”).

What said before also applies for city ID-based queries:

```
>>> owm.weather_history_at_id(12345, start=1379090800L, end=1379099800L)
```

Getting meteostation measurements history

Weather data measurements history for a specific meteostation is available in three sampling intervals: `'tick'` (which stands for minutely), `'hour'` and `'day'`. The calls to be made are:

```
# Get tick historic data for station 39276, only 4 data items
>>> hist = owm.station_tick_history(39276, limit=4)
# Get hourly historic data for station 39276
>>> hist = owm.station_hour_history(39276)
# Get daily historic data for station 39276, only 10 data items
>>> hist = owm.station_day_history(39276, 10)
```

and all of them return a `Historian` object. As you can notice, the amount of data measurements returned can be limited usign the proper parameter: by default, all available data items are retrieved. Each data item is composed by a temperature sample, a humidity sample, a pressure sample, a rain volume sample and a wind speed sample.

Once you have a `Historian` instance, you can obtain its encapsulated `StationHistory` object, which is a databox containing the data:

```
>>> sh = his.get_station_history()
```

and query data directly on it:

```
>>> sh.get_station_ID()                   # Meteostation ID
39276
>>> sh.get_interval()                     # Data sampling interval
'tick'
>>> sh.get_reception_time()               # Timestamp when data was received (GMT␣
↪UNIXtime, ISO8601
                                          # or datetime.datetime)
1377862896L
>>> sh.get_reception_time("iso")
'2013-08-30 20:07:57+00'
>>> sh.get_measurements()                 # Get historic data as a dict
{
    1362933983: {
         "temperature": 266.25,
         "humidity": 27.3,
         "pressure": 1010.02,
         "rain": None,
         "wind": 4.7
     },
    [...]
}
```

The last call gives you back a dictionary containing the historic weather data: the keys of the dictionary are the UNIX timestamps of data sampling and the values are dictionaries having a fixed set of keys (*temperature*, *humidity*, *pressure*, *rain*, *wind*) along with their corresponding numeric values.

If you have no specific need to handle the raw data by yourself, you can leverage the convenience methods provided by the `Historian` class:

```
# Get the temperature time series (in different units of measure)
>>> his.temperature_series()
[(1381327200, 293.4), (1381327260, 293.6), (1381327320, 294.4), ...]
>>> his.temperature_series(unit="celsius")
[(1381327200, 20.25), (1381327260, 20.45), (1381327320, 21.25), ...]
>>> his.temperature_series("fahrenheit")
[(1381327200, 68.45), (1381327260, 68.81), (1381327320, 70.25), ...]

# Get the humidity time series
>>> his.humidity_series()
[(1381327200, 27.3), (1381327260, 27.2), (1381327320, 27.2), ...]

# Get the atmospheric pressure time series
>>> his.pressure_series()
[(1381327200, 1010.02), (1381327260, 1010.23), (1381327320, 1010.79), ...]

# Get the rain volume time series
>>> his.rain_series()
```

(continues on next page)

(continued from previous page)

```
[(1381327200, None), (1381327260, None), (1381327320, None), ...]

# Get the wind speed time series
>>> his.wind_series()
[(1381327200, 4.7), (1381327260, 4.7), (1381327320, 4.9), ...]
```

Each of the `*_series()` methods returns a list of tuples, each tuple being a couple in the form: (timestamp, measured value). When in the series values are not provided by the OWM Weather API, the numeric value is `None`. These convenience methods are especially useful if you need to chart the historic time series of the measured physical entities.

You can also get minimum, maximum and average values of each series:

```
# Get the minimum temperature value in the series
>>> his.min_temperature(unit="celsius")
(1381327200, 20.25)

# Get the maximum rain value in the series
>>> his.max_rain()
()

# Get the average wind value in the series
>>> his.average_wind()
4.816
```

Dumping objects' content to JSON and XML

The PyOWM object instances can be dumped to JSON or XML strings:

```
# Dump a Weather object to JSON...
>>> w.to_JSON()
{'referenceTime':1377851530,'Location':{'name':'Palermo',
'coordinates':{'lon':13.35976,'lat':38.115822}'ID':2523920},...}

#... and to XML
>>> w.to_XML()
<?xml version='1.0' encoding='utf8'?>
<weather xmlns:w="http://github.com/csparpa/pyowm/tree/master/pyowm/weatherapi25/xsd/
↪weather.xsd">
<w:status>Clouds</w:status>[...]</weather>
```

When you dump to XML you can decide wether or not to print the standard XML encoding declaration line and XML Name Schema prefixes using the relative switches:

```
>>> w.to_XML(xml_declaration=True, xmlns=False)
```

Checking if OWM Weather API is online

You can check out the OWM Weather API service availability:

```
>>> owm.is_API_online()
True
```

Printing objects

Most of PyOWM objects can be pretty-printed for a quick introspection:

```
>>> print w
<pyowm.weatherapi25.weather.Weather - reference time=2013-12-18 16:41:00,␣
↪status=Drizzle>
>>> print w.get_location()
<pyowm.weatherapi25.location.Location - ID=1234, name=Barcelona, lon=2.9, lat=41.23>d
```

In the following sections you will find a brief explanation of PyOWM's object model, with detail about the classes and datastructures of interest. For a detailed description of the classes, please refer to the SW API documentation.

5.1.5 Abstractions

A few abstract classes are provided in order to allow code reuse for supporting new OWM Weather API versions and to eventually patch the currently supported ones.

The OWM abstract class

The *OWM* class is an abstract entry-point to the library. Clients can obtain a concrete implementation of this class through a factory method that returns the *OWM* subclass instance corresponding to the OWM Weather API version that is specified (or to the latest OWM Weather API version available).

In order to leverage the library features, you need to import the OWM factory and then feed it with an API key, if you have one (read here on how to obtain an API key). Of course you can change your API Key after object instantiation, if you need.

Each kind of weather query you can issue against the OWM Weather API is done through a correspondent method invocation on your *OWM* object instance.

Each OWM Weather API version may have different features, and therefore the mapping *OWM* subclass may have different methods. The *OWM* common parent class provides methods that tells you the PyOWM library version, the supported OWM Weather API version and the availability of the web API service: these methods are inherited by all the *OWM* children classes.

The JSONParser abstract class

This abstract class states the interface for OWM Weather API responses' JSON parsing: every API endpoint returns a different JSON message that has to be parsed to a specific object from the PyOWM object model. Subclasses of *JSONParser* shall implement this contract: instances of these classes shall be used by subclasses of the *OWM* abstract class.

The OWMCache abstract class

This abstract class states the interface for OWM Weather API responses' cache. The target of subclasses is to implement the get/set methods so that the JSON payloads of OWM Weather API responses are cached and looked up using the correspondent HTTP full URL that originated them.

The LinkedList abstract class

This abstract class models a generic linked list data structure.

5.1.6 OWM Weather API 2.5 object model

The configuration25 module

This module contains configuration data for the OWM Weather API 2.5 object model. Specifically:

```
* OWM Weather API endpoint URLs
* parser objects for API JSON payloads parsing
* registry object for City ID lookup
* cache providers
* misc data
```

As regards cache providers:

- by default, the library doesn't use any cache (it uses a null-object cache instance)
- the library provides a basic LRU cache implementation (class `LRUCache` in module `caches.lrucache.py`)
- you can leverage 3rd-party caching systems (eg: Memcached, MongoDB, Redis, file-system caches, etc..): all you have to do is write/obtain a wrapper module for those systems which conforms to the interface stated into the `abstractions.owmcache` abstract class.

You can write down your own configuration module and inject it into the PyOWM when you create the OWM global object, provided that you strictly follow the format of the `config25` module - which can be seen from the source code - and you put your own module in a location visible by the PYTHONPATH.

The OWM25 class

The *OWM25* class extends the *OWM* abstract base class and provides a method for each of the OWM Weather API 2.5 endpoint:

```
# CURRENT WEATHER QUERYING
* find current weather at a specific location ---> eg: owm.weather_at_place('London,UK
↪')
* find current weather at a specific city ID  ---> eg: owm.weather_at_id(1812597)
* find current weather at specific lat/lon ------> eg: owm.weather_at_coords(-0.
↪107331,51.503614)
* find weather currently measured by station ----> eg: owm.weather_at_station(1000)
* find current weathers in all locations
  with name is equal/similar to a specific name -> eg: owm.weather_at_places(
↪'Springfield',search='accurate')
* find current weathers in all locations
  in the surroundings of specific lon/lat -------> eg: owm.weather_around_coords(-2.
↪15, 57.0)

# METEOSTATIONS QUERYING
* find stations close to specific lat/lon -------> eg: owm.stations_at_coords(-0.
↪107331,51.503614)

# WEATHER FORECAST QUERYING
* find 3 hours weather forecast at a specific
  location ---------------------------------------> eg: owm.three_hours_forecast(
↪'Venice,IT')
* find daily weather forecast at a specific
  location --------------------------------------> eg: owm.daily_forecast('San␣
↪Francisco,US')
```

(continues on next page)

(continued from previous page)

```
# WEATHER HISTORY QUERYING
* find weather history for a specific location --> eg: owm.weather_history_at_place(
↪'Kiev,UA')
* find weather history for a specific city id  --> eg: owm.weather_history_at_
↪id(12345)
* find historic minutely data measurements for a
  specific meteostation --------------------------> eg: owm.station_tick_history(39276)
* find historic hourly data measurements for a
  specific meteostation --------------------------> eg: owm.station_hour_history(39276)
* find historic daily data measurements for a
  specific meteostation --------------------------> eg: owm.station_day_history(39276)
```

The methods illustrated above return a single object instance (*Observation* or *Forecast* types) a list of instances. In all cases, it is up to the clients to handle the returned entities.

The *OWM25* class is injected with *jsonparser* subclasses instances: each one parses a JSON response coming from a specific API endpoint and creates the objects returned to the clients. These dependencies are configured into the *configuration25* module and injected into this class.

In order to interact with the web API, this class leverages an *OWMHTTPClient* instance.

The Location class

The *Location* class represents a location in the world. Each instance stores the geographic name of the location, the longitude/latitude couple and the country name. These data are retrieved from the OWM Weather API 2.5 responses' payloads.

Location instances can also be retrieved from City IDs using the *CityIDRegistry* module.

The Weather class

This class is a databox containing information about weather conditions in a place. Stored data include text information such as weather status (sunny/rainy/snowy/. . .) and numeric information such as the values of measured phyisical entities (mx/min/current temperatures, wind speed/orientation, humidity, pressure, cloud coverage, . . .).

Some types of data are grouped and stored into Python dictionaries, such as weather and temperature info.

This class also stores the reference timestamp for the weather data, that is to say the time when the data was measured.

When using *OWM25* class for the retrieval of weather history on a location, eg:

```
owm.weather_history_at_place('Kiev,UA')
```

a list of *Weather* objects is returned.

The Observation class

An instance of this class is returned whenever a query about currently observed weather in a location is issued (hence, its name).

The *Observation* class binds information about weather features that are currently being observed in a specific location in the world and that are stored as a *Weather* object instance and the details about the location, which is stored into the class as a *Location* object instance. Both current weather and location info are obtained via OWM Weather API responses parsing, which done by other classes in the PyOWM library: usually this data parsing stage ends with their storage into a newly created *Observation* instance.

When created, every *Observation* instance is fed with a timestamp that tells when the weather observation data have been received.

When using *OWM25* class for the retrieval of currently observed weather in multiple locations, eg:

```
owm.weather_at_places('Springfield',search='accurate')
owm.weather_around_coords(-2.15, 57.0)
```

a list of *Observation* instances is returned to the clients.

The Forecast class

This class represents a weather forecast for a specific location in the world. A weather forecast is made out by location data - encapsulated by a *Location* object - and a collection of weather conditions - a list of *Weather* objects.

The OWM Weather API 2.5 provides two types of forecast intervals: three hours and daily; each *Forecast* instance has a specific fields that tells the interval of the forecast.

Forecast instances can also tell the reception timestamp for the weather forecast, that is to say the time when the forecast has been recevied from the OWM Weather API.

This class also provides an iterator for easily iterating over the encapsulated *Weather* list:

```
>>> fcst = owm.daily_forecast('Tokyo')
>>> for weather in fcst:
...   print (weather.get_reference_time(format='iso'), weather.get_status())
('2013-09-14 14:00:00+0','Clear')
('2013-09-14 17:00:00+0','Clear')
('2013-09-14 20:00:00+0','Clouds')
```

Sometimes - due to caching on the OWM API side - the weather objects returned inside a *Forecast* object may refer to timestamps in the recent past. In order to remove those outdated weather items from the forecast, you can do:

```
>>> fcst.actualize()
```

The Forecaster class

Instances of this class are returned by weather forecast queries such as:

```
f = owm.three_hours_forecast('London')
f = owm.daily_forecast('Buenos Aires',limit=6)
```

A *Forecaster* object wraps a *Forecast* object and provides convenience methods that makes it possible to perform complex weather forecast data queries, which could not otherwise be possible using only the *Forecast* class interface. A central concept with this regard is the "time coverage" of the forecast, that is to say the temporal length of the forecast.

It is then possible to know when a weather forecast starts/ends, know which *Weather* items in the forecast carry sunny/cloudy/… weather conditions, determine wether the forecast contains sunny/cloudy/… *Weather* items or not and to obtain the closest *Weather* item of the forecast to the time provided by the clients.

The StationHistory class

Instances of this class are returned by historic weather data query for meteostations, such as:

```
sh = owm.station_tick_history(39276)
sh = owm.station_hour_history(2865, limit=3)
sh = owm.station_day_history(2865)
```

A *StationHistory* object contains information about the ID of the meteostation, the time granularity of the retrieved data ('tick','hour' or 'day' - where 'tick' represents data sampled every minute) and of course the raw data: temperature, humidity, pressure, rain and wind speed.

The Historian class

This convenience class is dual to Forecaster. Instances of this class are returned by meteostation weather history queries such as:

```
h = owm.station_hour_history(39276)
h = owm.station_tick_history(39276)
```

A _Historian _ object wraps a *StationHistory* object and provides convenience methods that make it possible, in example, to obtain the time series of each of the measured physical entities: this is particularly useful for example when creating cartesian charts.

The weatherutils module

This utility module provides functions for searching and filtering collections of *Weather* objects.

5.1.7 Caches

Collection of caches

The NullCache class

This is a null-object that does nothing and is used by default as the PyOWM library caching mechanism

The LRUCache class

This is a Least-Recently Used simple cache with configurable size and elements expiration time.

5.1.8 Commons

A few common classes are provided to be used by all codes supporting different OWM Weather API versions.

The OWMHTTPClient class

This class is used to issue HTTP requests to the OWM Weather API endpoints.

The FrontLinkedList class

This class realizes a linked list that performs insertions only at the front of the list (time: O(1)) and deletions at any of its places (time: O(n))

5.1.9 Utilities

A few packages are provided, containing utility functions that support the base PyOWM entity classes and the user:

- **geo utils**: geographic areas representations
- **string utils**: string manipulation
- **temp utils**: temperature conversions
- **time format utils**: formatting timestamp from/to any among: epoch, ISO8601 and Python's `datetime.datetime`
- **time utils**: human friendly timestamp generation
- **weather utils**: weather objets search and filtering based on status, etc.
- **xml utils**: XML manipulation

5.1.10 Exceptions

There is a tiny exception hierarchy: **pyowm.exceptions.OWMError** is the custom PyOWM exceptions base class.

This class derives into:

- **APICallError**, which bases all of the network/infrastructural issue-related errors
- **APIResponseError**, which bases all of 4xx HTTP errors
- **ParseResponseError**, which is raised upon impossibility to parse the JSON payload of API responses

5.2 Agro API examples

5.2.1 Agro API examples

OWM provides an API for Agricultural monitoring that provides soil data, satellite imagery, etc.

The first thing you need to do to get started with it is to create a *polygon* and store it on the OWM Agro API. PyOWM will give you this new polygon's ID and you will use it to invoke data queries upon that polygon.

Eg: you can look up satellite imagery, weather data, historical NDVI for that specific polygon.

Read further on to get more details.

OWM website technical reference

- https://agromonitoring.com/api

AgroAPI Manager object

In order to do any kind of operations against the OWM Agro API, you need to obtain a `pyowm.agro10.agro_manager.AgroManager` instance from the main OWM. You'll need your API Key for that:

```
import pyowm
owm = pyowm.OWM('your-API-key')
mgr = owm.agro_manager()
```

Read on to discover what you can do with it.

Polygon API operations

A polygon represents an area on a map upon which you can issue data queries. Each polygon has a unique ID, an optional name and links back to unique OWM ID of the User that owns that polygon. Each polygon has an area that is expressed in hacres, but you can also get it in squared kilometers:

```
pol                     # this is a pyowm.agro10.polygon.Polygon instance
pol.id                  # ID
pol.area                # in hacres
pol.area_km             # in sq kilometers
pol.user_id             # owner ID
```

Each polygon also carries along the `pyowm.utils.geo.Polygon` object that represents the geographic polygon and the `pyowm.utils.geo.Point` object that represents the baricentre of the polygon:

```
geopol = pol.geopolygon    # pyowm.utils.geo.Polygon object
point = pol.center         # pyowm.utils.geo.Point object
```

Reading Polygons

You can either get all of the Polygons you've created on the Agro API or easily get single polygons by specifying their IDs:

```
list_of_polygons = mgr.get_polygons()
a_polygon = mgr.get_polygon('5abb9fb82c8897000bde3e87')
```

Creating Polygons

Creating polygons is easy: you just need to create a `pyowm.utils.geo.Polygon` instance that describes the coordinates of the polygon you want to create on the Agro API. Then you just need to pass it (along with an optional name) to the Agro Manager object:

```

# first create the pyowm.utils.geo.Polygon instance that represents the area (here, a␣
↪triangle)
from pyowm.utils.geo import Polygon as GeoPolygon
gp = GeoPolygon([[
        [-121.1958, 37.6683],
        [-121.1779, 37.6687],
        [-121.1773, 37.6792],
        [-121.1958, 37.6683]]])
```

(continues on next page)

(continued from previous page)

```
# use the Agro Manager to create your polygon on the Agro API
the_new_polygon = mgr.create_polygon(gp, 'my new shiny polygon')

# the new polygon has an ID and a user_id
the_new_polygon.id
the_new_polygon.user_id
```

You get back a `pyowm.agro10.polygon.Polygon` instance and you can use its ID to operate this new polygon on all the other Agro API methods!

Updating a Polygon

Once you've created a polygon, you can only change its mnemonic name, as the rest of its parameters cannot be changed by the user. In order to do it:

```
my_polygon.name  # "my new shiny polygon"
my_polygon.name = "changed name"
mgr.update_polygon(my_polygon)
```

Deleting a Polygon

Delete a polygon with

```
mgr.delete_polygon(my_polygon)
```

Remember that when you delete a polygon, there is no going back!

Soil data API Operations

Once you've defined a polygon, you can easily get soil data upon it. Just go with:

```
soil = mgr.soil_data(polygon)
```

`Soil` is an entity of type `pyowm.agro10.soil.Soil` and is basically a wrapper around a Python dict reporting the basic soil information on that polygon:

```
soil.polygon_id                         # str
soil.reference_time(timeformat='unix')  # can be: int for UTC Unix time ('unix'),
                                        # ISO8601-formatted str for 'iso' or
                                        # datetime.datetime for 'date'
soil.surface_temp(unit='kelvin')        # float (unit can be: 'kelvin', 'celsius' or
↪'fahrenheit')
soil.ten_cm_temp(unit='kelvin')         # float (Kelvins, measured at 10 cm depth) -␣
↪unit same as for above
soil.moisture                           # float (m^3/m^3)
```

Soil data is updated twice a day.

Satellite Imagery API Operations

This is the real meat in Agro API: the possibility to obtain **satellite imagery** right upon your polygons!

Overwiew

Satellite Imagery comes in 3 formats:

- **PNG images**
- **PNG tiles** (variable zoom level)
- **GeoTIFF images**

Tiles can be retrieved by specifying a proper set of tile coordinates (x, y) and a zoom level: please refer to PyOWM's Map Tiles client documentation for further insights.

When we say that imagery is upon a polygon we mean that the polygon is fully contained in the scene that was acquired by the satellite.

Each image comes with a **preset**: a preset tells how the acquired scene has been post-processed, eg: image has been put in false colors or image contains values of the Enhanced Vegetation Index (EVI) calculated on the scene

Imagery is provided by the Agro API for different **satellites**

Images that you retrieve from the Agro API are `pyowm.agroapi10.imagery.SatelliteImage` instances, and they **contain both image's data and metadata**.

You can download NDVI images using several **color palettes** provided by the Agro API, for easier processing on your side.

Operations summary

Once you've defined a polygon, you can:

- **search for available images** upon the polygon and taken in a specific time frame. The search can be performed with multiple filters (including: satellite symbol, image type, image preset, min/max resolution, minx/max cloud coverage, . . .) and returns search results, each one being *metadata* for a specific image.
- from those metadata, **download an image**, be it a regular scene or a tile, optionally specifying a color palette for NDVI ones
- if your image has EVI or NDVI presets, you can **query for its statistics**: these include min/max/median/p25/p75 values for the corresponding index

A concrete example: we want to acquire all NDVI GeoTIFF images acquired by Landsat 8 from July 18, 2017 to October 26, 2017; then we want to get stats for one such image and to save it to a local file.

```
from pyowm.commons.enums import ImageTypeEnum
from pyowm.agroapi10.enums import SatelliteEnum, PresetEnum

pol_id = '5abb9fb82c8897000bde3e87'  # your polygon's ID
acq_from = 1500336000                # 18 July 2017
acq_to = 1508976000                  # 26 October 2017
img_type = ImageTypeEnum.GEOTIFF     # the image format type
preset = PresetEnum.NDVI     # the preset
sat = SatelliteEnum.LANDSAT_8.symbol # the satellite


# the search returns images metadata (in the form of `MetaImage` objects)
results = mgr.search_satellite_imagery(pol_id, acq_from, acq_to, img_type=img_type,␣
↪preset=preset, None, None, acquired_by=sat)

# download all of the images
```

(continues on next page)

(continued from previous page)

```
satellite_images = [mgr.download_satellite_image(result) for result in results]

# get stats for the first image
sat_img = satellite_images[0]
stats_dict = mgr.stats_for_satellite_image(sat_img)

# ...satellite images can be saved to disk
sat_img.persist('/path/to/my/folder/sat_img.tif')
```

Let's see in detail all of the imagery-based operations.

Searching images

Search available imagery upon your polygon by specifying at least a mandatory time window, with from and to timestamps expressed as UNIX UTC timestamps:

```
pol_id = '5abb9fb82c8897000bde3e87'  # your polygon's ID
acq_from = 1500336000                # 18 July 2017
acq_to = 1508976000                  # 26 October 2017

# the most basic search ever: search all available images upon the polygon in the␣
↪specified time frame
metaimages_list = mgr.search_satellite_imagery(pol_id, acq_from, acq_to)
```

What you will get back is actually metadata for the actual imagery, not data.

The function call will return **a list of `pyowm.agroapi10.imagery.MetaImage` instances, each one being a bunch of metadata relating to one single satellite image**.

Keep these objects, as you will need them in order to download the corresponding satellite images from the Agro API: think of them such as descriptors for the real images.

But let's get back to search! Search is a parametric affair… **you can specify many more filters**:

- the image format type (eg. PNG, GEOTIFF)
- the image preset (eg. false color, EVI)
- the satellite that acquired the image (you need to specify its symbol)
- the px/m resolution range for the image (you can specify a minimum value, a maximum value or both of them)
- the % of cloud coverage on the acquired scene (you can specify a minimum value, a maximum value or both of them)
- the % of valid data coverage on the acquired scene (you can specify a minimum value, a maximum value or both of them)

Sky is the limit…

As regards image type, image preset and satellite filters please refer to subsequent sections explaining the supported values.

Examples of search:

```
from pyowm.commons.enums import ImageTypeEnum
from pyowm.agroapi10.enums import SatelliteEnum, PresetEnum


# search all Landsat 8 images in the specified time frame
results = mgr.search_satellite_imagery(pol_id, acq_from, acq_to, acquired_
↪by=SatelliteEnum.LANDSAT_8.symbol)

# search all GeoTIFF images in the specified time frame                       ␣
↪
results = mgr.search_satellite_imagery(pol_id, acq_from, acq_to, img_
↪type=ImageTypeEnum.GEOTIFF)

# search all NDVI images acquired by Sentinel 2 in the specified time frame
results = mgr.search_satellite_imagery(pol_id, acq_from, acq_to, acquired_
↪by=SatelliteEnum.SENTINEL_2.symbol,
                                       preset=PresetEnum.NDVI)

# search all PNG images in the specified time frame with a max cloud coverage of 1%␣
↪and a min valid data coverage of 98%
results = mgr.search_satellite_imagery(pol_id, acq_from, acq_to, img_
↪type=ImageTypeEnum.PNG,
                                       max_cloud_coverage=1, min_valid_data_
↪coverage=98)

# search all true color PNG images in the specified time frame, acquired by Sentinel␣
↪2, with a range of metric resolution
# from 4 to 16 px/m, and with at least 90% of valid data coverage
results = mgr.search_satellite_imagery(pol_id, acq_from, acq_to, img_
↪type=ImageTypeEnum.PNG, preset=PresetEnum.TRUE_COLOR,
                                       min_resolution=4, max_resolution=16, min_valid_
↪data_coverage=90)
```

So, what metadata can be extracted by a `MetaImage` object? Here we go:

```
metaimage.polygon_id                 # the ID of the polygon upon which the image is␣
↪taken
metaimage.url                        # the URL the actual satellite image can be␣
↪fetched from
metaimage.preset                     # the satellite image preset
metaimage.image_type                 # the satellite image format type
metaimage.satellite_name             # the name of the satellite that acquired the␣
↪image
metaimage.acquisition_time('unix')   # the timestamp when the image was taken (can be␣
↪specified using: 'iso', 'unix' and 'date')
metaimage.valid_data_percentage      # the percentage of valid data coverage on the␣
↪image
metaimage.cloud_coverage_percentage  # the percentage of cloud coverage on the image
metaimage.sun_azimuth                # the sun azimuth angle at scene acquisition time
metaimage.sun_elevation              # the sun zenith angle at scene acquisition time
metaimage.stats_url                  # if the image is EVI or NDVI, this is the URL␣
↪where index statistics can be retrieved (see further on for details)
```

Download an image

Once you've got your metaimages ready, you can download the actual satellite images.

In order to download, you must specify to the Agro API manager object at least the desired metaimage to fetch. If you're downloading a tile, you must specify tile coordinates (x, y, and zoom level): these are mandatory, and if you forget to provide them you'll get an `AssertionError`.

Optionally, you can specify a color palette - but this will be significant only if you're downloading an image with NDVI preset (otherwise the palette parameter will be safely ignored) - please see further on for reference.

Once download is complete, you'll get back a `pyowm.agroapi10.imagery.SatelliteImage` object (more on this in a while).

Here are some examples:

```
from pyowm.agroapi10.enums import PaletteEnum

# Download a NDVI image
ndvi_metaimage   # metaimage for a NDVI image
bnw_sat_image = mgr.download_satellite_image(ndvi_metaimage, preset=PaletteEnum.BLACK_
↪AND_WHITE)
green_sat_image =  mgr.download_satellite_image(ndvi_metaimage, preset=PaletteEnum.
↪GREEN)

# Download a tile
tile_metaimage   # metaimage for a tile
tile_image = mgr.download_satellite_image(tile_metaimage, x=2, y=3, zoom=5)
tile_image = mgr.download_satellite_image(tile_metaimage)   # AssertionError (x, y␣
↪and zoom are missing!)
```

Downloaded satellite images contain both binary image data and and embed *the original* `MetaImage` *object describing image metadata*. Furthermore, you can query for the download time of a satellite image, and for its related color palette:

```

# Get satellite image download time - you can as usual specify: 'iso', 'date' and
↪'unix' time formats
bnw_sat_image.downloaded_on('iso')  #  '2017-07-18 14:08:23+00'

# Get its palette
bnw_sat_image.palette               #  '2'

# Get satellite image's data and metadata
bnw_sat_image.data                  # this returns a `pyowm.commons.image.Image`␣
↪object or a
                                    # `pyowm.commons.tile.Tile` object depending on␣
↪the satellite image
metaimage = bnw_sat_image.metadata  # this gives a `MetaImage` subtype object

# Use the Metaimage object as usual...
metaimage.polygon_id
metaimage.preset,
metaimage.satellite_name
metaimage.acquisition_time
```

You can also save satellite images to disk - it's as easy as:

```
bnw_sat_image.persist('C:\myfolder\myfile.png')
```

Querying for NDVI and EVI image stats

NDVI and EVI preset images have an extra blessing: you can query for statistics about the image index.

Once you've downloaded such satellite images, you can query for stats and get back a data dictionary for each of them:

```
ndvi_metaimage   # metaimage for a NDVI image

# download it
bnw_sat_image = mgr.download_satellite_image(ndvi_metaimage, preset=PaletteEnum.BLACK_
↪AND_WHITE)

# query for stats
stats_dict = mgr.stats_for_satellite_image(bnw_sat_image)
```

Stats dictionaries contain:

- `std`: the standard deviation of the index
- `p25`: the first quartile value of the index
- `num`: the number of pixels in the current polygon
- `min`: the minimum value of the index
- `max`: the maximum value of the index
- `median`: the median value of the index
- `p75`: the third quartile value of the index
- `mean`: the average value of the index

What if you try to get stats for a non-NDVI or non-EVI image? A `ValueError` will be raised!

Supported satellites

Supported satellites are provided by the `pyowm.agroapi10.enums.SatelliteEnum` enumerator which returns `pyowm.commons.databoxes.Satellite` objects:

```
from pyowm.agroapi10.enums import SatelliteEnum

sat = SatelliteEnum.SENTINEL_2
sat.name    # 'Sentinel-2'
sat.symbol # 's2'
```

Currently only Landsat 8 and Sentinel 2 satellite imagery is available

Supported presets

Supported presets are provided by the `pyowm.agroapi10.enums.PresetEnum` enumerator which returns strings, each one representing an image preset:

```
from pyowm.agroapi10.enums import PresetEnum

PresetEnum.TRUE_COLOR  # 'truecolor'
```

Currently these are the supported presets: true color, false color, NDVI and EVI

Supported image types

Supported image types are provided by the `pyowm.commons.databoxes.ImageTypeEnum` enumerator which returns `pyowm.commons.databoxes.ImageType` objects:

```
from pyowm.agroapi10.enums import ImageTypeEnum

png_type = ImageTypeEnum.PNG
geotiff_type = ImageTypeEnum.GEOTIFF
png_type.name        # 'PNG'
png_type.mime_type   # 'image/png'
```

Currently only PNG and GEOTIFF imagery is available

Supported color palettes

Supported color palettes are provided by the `pyowm.agroapi10.enums.PaletteEnum` enumerator which returns strings, each one representing a color palette for NDVI imges:

```
from pyowm.agroapi10.enums import PaletteEnum

PaletteEnum.CONTRAST_SHIFTED   # '3'
```

As said, palettes only apply to NDVI images: if you try to specify palettes when downloading images with different presets (eg. false color images), *nothing will happen*.

The default Agro API color palette is `PaletteEnum.GREEN` (which is `1`): if you don't specify any palette at all when downloading NDVI images, they will anyway be returned with this palette.

As of today green, black and white and two contrast palettes (one continuous and one continuous but shifted) are supported by the Agro API. Please check the documentation for palettes' details, including control points.

5.3 UV API examples

You can query the OWM API for current Ultra Violet (UV) intensity data in the surroundings of specific geocoordinates.

Please refer to the official API docs for UV

5.3.1 Querying UV index observations

Getting the data is easy:

```
uvi = owm.uvindex_around_coords(lat, lon)
```

The query returns an UV Index value entity instance

5.3.2 Querying UV index forecasts

As easy as:

```
uvi_list = owm.uvindex_forecast_around_coords(lat, lon)
```

5.3.3 Querying UV index history

As easy as:

```
uvi_history_list = owm.uvindex_history_around_coords(
    lat, lon,
    datetime.datetime(2017, 8, 1, 0, 0),
    end=datetime.datetime(2018, 2, 15, 0, 0))
```

`start` and `end` can be ISO-8601 date strings, unix timestamps or Python datetime objects.

In case `end` is not provided, then UV historical values will be retrieved dating back to `start` up to the current timestamp.

5.3.4 `UVIndex` entity

`UVIndex` is an entity representing a UV intensity measurement on a certain geopoint. Here are some of the methods:

```
uvi.get_value()
uvi.get_reference_time()
uvi.get_reception_time()
uvi.get_exposure_risk()
```

The `get_exposure_risk()` methods returns a string estimating the risk of harm from unprotected sun exposure if an average adult was exposed to a UV intensity such as the on in this measurement. This is the source mapping for the statement.

5.4 Air Pollution API examples

5.4.1 Carbon Monoxide (CO) Index

You can query the OWM API for Carbon Monoxide (CO) measurements in the surroundings of specific geocoordinates.

Please refer to the official API docs for CO data consumption for details about how the search radius is influenced by the decimal digits on the provided lat/lon values.

Queries return the latest CO Index values available since the specified `start` date and across the specified `interval` timespan. If you don't specify any value for `interval` this is defaulted to: `'year'`. Eg:

- `start='2016-07-01 15:00:00Z'` and `interval='hour'`: searches from 3 to 4 PM of day 2016-07-01
- `start='2016-07-01'` and `interval='day'`: searches on the day 2016-07-01
- `start='2016-07-01'` and `interval='month'`: searches on the month of July 2016
- `start='2016-07-01'` and `interval='year'`: searches from day 2016-07-01 up to the end of year 2016

Please be aware that also data forecasts can be returned, depending on the search query.

Querying CO index

Getting the data is easy:

```
# Get latest CO Index on geocoordinates
coi = owm.coindex_around_coords(lat, lon)

# Get available CO Index in the last 24 hours
coi = owm.coindex_around_coords(lat, lon,
    start=timeutils.yesterday(), interval='day')

# Get available CO Index in the last ...
coi = owm.coindex_around_coords(
    lat, lon,
    start=start_datetime,  # iso-8601, unix or datetime
    interval=span)         # can be: 'minute', 'hour', 'day', 'month', 'year'
```

`COIndex` entity

`COIndex` is an entity representing a set of CO measurements on a certain geopoint. Each CO measurement is taken at a certain air pressure value and has a VMR (Volume Mixing Ratio) value for CO. Here are some of the methods:

```
list_of_samples = coi.get_co_samples()
location = coi.get_location()
coi.get_reference_time()
coi.get_reception_time()

max_sample = coi.get_co_sample_with_highest_vmr()
min_sample = coi.get_co_sample_with_lowest_vmr()
```

If you want to know if a COIndex refers to the future - aka: is a forecast - wth respect to the current timestamp, then use the `is_forecast()` method

5.4.2 Ozone (O3)

You can query the OWM API for Ozone measurements in the surroundings of specific geocoordinates.

Please refer to the official API docs for O3 data consumption for details about how the search radius is influenced by the decimal digits on the provided lat/lon values.

Queries return the latest Ozone values available since the specified `start` date and across the specified `interval` timespan. If you don't specify any value for `interval` this is defaulted to: `'year'`. Eg:

- `start='2016-07-01 15:00:00Z'` and `interval='hour'`: searches from 3 to 4 PM of day 2016-07-01
- `start='2016-07-01'` and `interval='day'`: searches on the day 2016-07-01
- `start='2016-07-01'` and `interval='month'`: searches on the month of July 2016
- `start='2016-07-01'` and `interval='year'`: searches from day 2016-07-01 up to the end of year 2016

Please be aware that also data forecasts can be returned, depending on the search query.

Querying Ozone data

Getting the data is easy:

```
# Get latest O3 value on geocoordinates
o3 = owm.ozone_around_coords(lat, lon)

# Get available O3 value in the last 24 hours
oz = owm.ozone_around_coords(lat, lon,
         start=timeutils.yesterday(), interval='day')

# Get available O3 value in the last ...
oz = owm.ozone_around_coords(
       lat, lon,
        start=start_datetime,  # iso-8601, unix or datetime
        interval=span)         # can be: 'minute', 'hour', 'day', 'month', 'year'
```

`Ozone` entity

`Ozone` is an entity representing a set of CO measurements on a certain geopoint. Each ozone value is expressed in Dobson Units. Here are some of the methods:

```
location = oz.get_location()
oz = get_du_value()
oz.get_reference_time()
oz.get_reception_time()
```

If you want to know if an Ozone measurement refers to the future - aka: is a forecast - wth respect to the current timestamp, then use the `is_forecast()` method

Querying Nitrogen dioxide (NO2) and Sulfur Dioxide (SO2) data

This works exactly as for O2 adata - please refer to that bit of the docs

5.5 Stations API examples

5.5.1 Stations API 3.0 usage examples

Meteostations

Managing meteostations is easy!

Just get a reference to the `stationsapi30..stations_manager.StationsManager` object that proxies the OWM Stations API, and then work on it

You can issue CRUD (Create Read Update Delete) actions on the `StationsManager` and data is passed in/out in the form of `stationsapi30.stations.Station` objects

Here are some examples:

```
import pyowm
owm = pyowm.OWM('your-API-key')
mgr = owm.stations_manager()        # Obtain the Stations API client

# Create a new station
station = mgr.create_station("SF_TEST001", "San Francisco Test Station",
                                 37.76, -122.43, 150)
# Get all your stations
all_stations = mgr.get_stations()

# Get a station named by id
id = '583436dd9643a9000196b8d6'
retrieved_station = mgr.get_station(id)

# Modify a station by editing its "local" proxy object
retrieved_station.name = 'A different name'
mgr.modify_station(retrieved_station)

# Delete a station and all its related measurements
mgr.delete_station(retrieved_station)
```

Measurements

Each meteostation tracks datapoints, each one represented by an object. Datapoints that you submit to the OWM Stations API (also called "raw measurements") are of type: `stationsapi30.measurement.Measurement`, while datapoints that you query against the API come in the form of: `stationsapi30.measurement.AggregatedMeasurement` objects.

Each `stationsapi30.measurement.Measurement` cointains a reference to the `Station` it belongs to:

```
measurement.station_id
```

Create such objects with the class constructor or using the `stationsapi30.measurement.Measurement.from_dict()` utility method.

Once you have a raw measurement or a list of raw measurements (even belonging to mixed stations), you can submit them to the OWM Stations API via the `StationsManager` proxy:

```

# Send a new raw measurement for a station
mgr.send_measurement(raw_measurement_obj)

# Send a list of new raw measurements, belonging to multiple stations
mgr.send_measurements(list_of_raw_measurement_objs)
```

Reading measurements from the OWM Stations API can be easily done using the `StationsManager` as well. As sad, they come in the form of `stationsapi30.measurement.AggregatedMeasurement` instances. Each of such objects represents an *aggregation of measurements* for the station that you specified, with an aggregation time granularity of *day*, *hour* or *minute* - you tell what. You can query aggregated measurements in any time window.

So when querying for measurements, you need to specify:

- the reference station ID
- the aggregation granularity (as sai, among: `d`, `h` and `m`)
- the time window (start-end Unix timestamps)

- how many results you want

Example:

```
# Read aggregated measurements (on day, hour or minute) for a station in a given
# time interval
aggr_msmts = mgr.get_measurements(station_id, 'h', 1505424648, 1505425648, limit=5)
```

Buffers

As usually a meteostation tracks a lot of datapoints over time and it is expensive (eg. in terms of battery and bandwidth usage) to submit them one by one to the OWM Stations API, a good abstraction tool to work with with measurements is `stationsapi30.buffer.Buffer` objects.

A buffer is basically a “box” that collects multiple measurements for a station. You can use the buffer to store measurements over time and to send all of the measurements to the API at once.

Examples:

```
from pyowm.stationsapi30.buffer import Buffer

# Create a buffer for a station...
buf = Buffer(station_id)

# ...and append measurement objects to it
buf.append(msmt_1)
buf.append(msmt_2)
buf.append(msmt_3)

# ... or read data from other formats
# -- a dict
# (as you would pass to Measurement.from_dict method)
buf.append_from_dict(msmt_dict)
# -- a JSON string
# that string must be parsable as a dict that you can feed to
# Measurement.from_dict method
with open('my-msmts.json') as j:
    buf.append_from_json(j.read())

# buffers are nice objects
# -- they are iterable
print(len(buf))
for measurement in buf:
    print(measurement)

# -- they can be joined
new_buf = buf + another_buffer

# -- they can be emptied
buf.empty()

# -- you can order measurements in a buffer by their creation time
buf.sort_chronologically()
buf.sort_reverse_chronologically()
```

(continues on next page)

(continued from previous page)

```
# Send measurements stored in a buffer to the API using the StationManager object
mgr.send_buffer(buf)
```

You can load/save measurements into/from Buffers from/tom any persistence backend:

- *Saving*: persist data to the filesystem or to custom data persistence backends that you can provide (eg. databases)
- *Loading*: You can also pre-load a buffer with (or append to it) measurements stored on the file system or read from custom data persistence backends

The default persistence backend is: `stationsapi30.persistence_backend.JSONPersistenceBackend` and allows to read/write buffer data from/to JSON files

As said, you can use your own *custom data backends*: they must be subclasses of `stationsapi30.persistence_backend.PersistenceBackend`

Examples:

```

from pyowm.stationsapi30 import persistence_backend

# instantiate the default JSON-based backend: you need to provide the ID of the
# stations related to measurements...
json_be = persistence_backend.JSONPersistenceBackend('/home/myfile.json', station_id)


# ... and use it to load a buffer
buf = json_be.load_to_buffer()


# ... and to save buffers
json_be.persist_buffer(buf)


# You can use your own persistence backends
my_custom_be = MyCustomPersistenceBackend()
buf = my_custom_be.load_to_buffer()
my_custom_be.persist_buffer(buf)
```

5.6 Alerts API examples

5.6.1 Weather Alert API

You can use the OWM API to create triggers.

Each trigger represents the check if a set of conditions on certain weather parameter values are met over certain geographic areas.

Whenever a condition is met, an alert is fired and stored, and can be retrieved by polling the API.

OWM website technical reference

- https://openweathermap.org/triggers
- http://openweathermap.org/triggers-struct

A full example first

Hands-on! This is a full example of how to use the Alert API. Check further for details about the involved object types.

```python
from pyowm import OWM
from pyowm.utils import geo
from pyowm.alertapi30.enums import WeatherParametersEnum, OperatorsEnum,␣
↪AlertChannelsEnum
from pyowm.alertapi30.condition import Condition

# obtain an AlertManager instance
owm = OWM(API_Key='blablabla')
am = owm.alert_manager()

# -- areas --
geom_1 = geo.Point(lon, lat)  # available types: Point, MultiPoint, Polygon,␣
↪MultiPolygon
geom_1.geojson()
'''
{
  "type": "Point",
  "coordinates":[ lon, lat ]
}
'''
geom_2 = geo.MultiPolygon([[lon1, lat1], [lon2, lat2], [lon3, lat3], [lon1, lat1]]
                          [[lon7, lat7], [lon8, lat8], [lon9, lat9], [lon7, lat7]])


# -- conditions --
condition_1 = Condition(WeatherParametersEnum.TEMPERATURE,
                        OperatorsEnum.GREATER_THAN,
                        313.15)  # kelvin
condition_2 = Condition(WeatherParametersEnum.CLOUDS,
                        OperatorsEnum.EQUAL,
                        80) # clouds % coverage

# -- triggers --

# create a trigger
trigger = am.create_trigger(start_after_millis_=355000, end_after_millis=487000,
                            conditions=[condition_1, condition_2],
                            area=[geom_1, geom_2],
                            alert_channel=AlertChannelsEnum.OWM_API)

# read all triggers
triggers_list = am.get_triggers()

# read one named trigger
trigger_2 = am.get_trigger('trigger_id')

# update a trigger
am.update_trigger(trigger_2)

# delete a trigger
am.delete_trigger(trigger_2)

# -- alerts --
```

(continues on next page)

(continued from previous page)

```
# retrieved from the local parent Trigger obj...
alerts_list = trigger.get_alerts()
alerts_list = trigger.get_alerts_since('2018-01-09T23:07:24Z')  # useful for polling␣
↪alerts
alerts_list = trigger.get_alerts_on(WeatherParametersEnum.TEMPERATURE)
alert = trigger.get_alert('alert_id')

# ...or retrieved from the remote API
alerts_list_alternate = am.get_alerts_for(trigger)
alert_alternate = am.get_alert('alert_id')


# delete all or one alert
am.delete_all_alerts_for(trigger)
am.delete_alert_for(trigger, alert)
```

Alert API object model

This is the Alert API object model:

- *Trigger*: collection of alerts to be met over specified areas and within a specified time frame according to specified weather params conditions
- *Condition*: rule for matching a weather measuerment with a specified threshold
- *Alert*: whenever a condition is met, an alert is created (or updated) and can be polled to verify when it has been met and what the actual weather param value was.
- *Area*: geographic area over which the trigger is checked
- *AlertChannel*: as OWM plans to add push-oriented alert channels (eg. push notifications), we need to encapsulate this into a specific class

and then you have an *AlertManager* class that you will need to instantiate to operatively interact with the Alert API

Area

The area describes the geographic boundaries over which a trigger is evaluated. Don't be mislead by the term "area": this can refer also to a specific geopoint or a set of them, besides - of course - polygons and set of polygons.

Any of the geometry subtypes found in `pyowm.utils.geo` module (point, multipoint, polygon, multipolygon) are fine to use.

Example:

```
from pyowm.utils import geo
point = geo.Point(20.8, 30.9)  # available geometry types: Point, MultiPoint, Polygon,
↪ MultiPolygon
point.geojson()
'''
{
  "type": "Point",
  "coordinates":[ 20.8, 30.9 ]
}
'''
```

Defining complex geometries is sometimes difficult, but in most cases you just need to set triggers upon cities: that's why we've added a method to the `pyowm.weatherapi25.cityidregistry.CityIDRegistry` registry that returns the geopoints that correspond to one or more named cities:

```
import pyowm
owm = pyowm.OWM('your-API-key')
reg = owm.city_id_registry()
geopoints = reg.geopoints_for('London', country='GB')
```

But still some very spread cities (think of London,GB or Los Angeles,CA) exist and therefore approximating a city to a single point is not accurate at all: that's why we've added a nice method to get a *squared polygon that is circumscribed to the circle having a specified geopoint as its centre*. This makes it possible to easily get polygons to cover large squared areas and you would only need to specify the radius of the circle. Let's do it for London,GB in example:

```
geopoints = reg.geopoints_for('London', country='GB')
centre = geopoints[0] # the list has only 1 geopoint
square_polygon = centre.bounding_square_polygon(inscribed_circle_radius_km=12) #␣
↪radius of the inscribed circle in kms (defaults to: 10)
```

Please, notice that if you specify big values for the radius you need to take care about the projection of geographic coordinates on a proper geoid: this means that if you don't, the polygon will only *approximate* a square.

Topology is set out as stated by GeoJSON

Moreover, there is a useful factory for Areas: `pyowm.utils.geo.GeometryBuilder.build()`, that you can use to turn a geoJSON standard dictionary into the corresponding topology type:

```
from pyowm.utils.geo import GeometryBuilder
the_dict = {
    "type": "Point",
    "coordinates": [53, 37]
}
geom = GeometryBuilder.build(the_dict)
type(geom)  # <pyowm.utils.geo.Point>
```

You can bind multiple `pyowm.utils.geo` geometry types to a Trigger: a list of such geometries is considered to be the area on which conditions of a Trigger are checked.

Condition

A condition is a numeric rule to be checked on a named weather variable. Something like:

```
  - VARIABLE X IS GREATER THAN AMOUNT_1
  - VARIABLE Y IS EQUAL TO AMOUNT_2
  - VARIABLE Z IS DIFFERENT FROM AMOUNT_3
```

`GREATER, EQUAL TO, DIFFERENT FROM` are called comparison expressions or operators; `VARIABLE X, Y, Z` are called target parameters.

Each condition is then specified by:

- target_param: weather parameter to be checked. Can be: `temp, pressure, humidity, wind_speed, wind_direction, clouds`.
- expression: str, operator for the comparison. Can be: `$gt`, $gte, $lt, $lte, $eq, $ne'
- amount: number, the comparison value

Conditions are bound to Triggers, as they are set on Trigger instantiation.

As Conditions can be only set on a limited number of weather variables and can be expressed only through a closed set of comparison operators, convenient **enumerators** are offered in module `pyowm.alertapi30.enums`:

- `WeatherParametersEnum` –> what weather variable to set this condition on
- `OperatorsEnum` –> what comparison operator to use on the weather parameter

Use enums so that you don't have to remember the syntax of operators and weather params that is specific to the OWM Alert API. Here is how you use them:

```
from pyowm.alertapi30 import enums
enums.WeatherParametersEnum.items()       # [('TEMPERATURE', 'temp'), ('WIND_SPEED',
↪'wind_speed'), ... ]
enums.WeatherParametersEnum.TEMPERATURE  # 'temp'
enums.WeatherParametersEnum.WIND_SPEED   # 'wind_speed'

enums.OperatorsEnum.items()              # [('GREATER_THAN', '$gt'), ('NOT_EQUAL', '
↪$ne'), ... ]
enums.OperatorsEnum.GREATER_THAN         # '$gt'
enums.OperatorsEnum.NOT_EQUAL            # '$ne'
```

Here is an example of conditions:

```
from pyowm.alertapi30.condition import Condition
from pyowm.alertapi30 import enums

# this condition checks if the temperature is bigger than 313.15 Kelvin degrees
condition = Condition(enums.WeatherParametersEnum.TEMPERATURE,
                      enums.OperatorsEnum.GREATER_THAN,
                      313.15)
```

Remember that each Condition is checked by the OWM Alert API on the geographic area that you need to specify!

You can bind multiple `pyowm.alertapi30.condition.Condition` objects to a Trigger: each Alert will be fired when a specific Condition is met on the area.

Alert

As said, whenever one or more conditions are met on a certain area, an alert is fired (this means that "the trigger triggers")

If the condition then keeps on being met, more and more alerts will be spawned by the OWM Alert API. You can retrieve such alerts by polling the OWM API (see below about how to do it).

Each alert is represented by PyOWM as a `pyowm.alertapi30.alert.Alert` instance, having:

- a unique identifier
- timestamp of firing
- a link back to the unique identifier of the parent `pyowm.alertapi30.trigger.Trigger` object instance
- the list of met conditions (each one being a dict containing the `Condition` object and the weather parameter value that actually made the condition true)
- the geocoordinates where the condition has been met (they belong to the area that had been specified for the Trigger)

Example:

```
from pyowm.alertapi30.condition import Condition
from pyowm.alertapi30 import enums
from pyowm.alertapi30.alert import Alert

condition = Condition(enums.WeatherParametersEnum.TEMPERATURE,
                      enums.OperatorsEnum.GREATER_THAN,
                      356.15)

alert = Alert('alert-id',                     # alert id
              'parent-trigger-id',            # parent trigger's id
              [{                              # list of met conditions
                    "current_value": 326.4,
                    "condition": condition
               }],
               {"lon": 37, "lat": 53},        # coordinates
               1481802100000                  # fired on
)
```

As you see, you're not meant to create alerts, but PyOWM is supposed to create them for you as they are fired by the OWM API.

AlertChannel

Something that OWM envisions, but still does not offer. Possibly, when you will setup a trigger you shall also specify the channels you want to be notified on: that's why we've added a reference to a list of `AlertChannel` instances directly on the Trigger objects (the list now only points to the default channel)

A useful enumerator is offered in module `pyowm.alertapi30.enums`: `AlertChannelsEnum` (says what channels should the alerts delivered to)

As of today, the default `AlertChannel` is: `AlertChannelsEnum.OWM_API_POLLING`, and is the only one available.

Trigger

As said, each trigger represents the check if a set of conditions on certain weather parameter values are met over certain geographic areas.

A Trigger is the local proxy for the corresponding entry on the OWM API: Triggers can be operated through `pyowm.alertapi30.alertmanager.AlertManager` instances.

Each Trigger has these attributes:

- start_after_millis: *with resepect to the time when the trigger will be crated on the Alert API*, how many milliseconds after should it begin to be checked for conditions matching
- end_after_millis: *with resepect to the time when the trigger will be crated on the Alert API*, how many milliseconds after should it end to be checked for conditions matching
- alerts: a list of `pyowm.alertapi30.alert.Alert` instances, which are the alerts that the trigger has fired so far
- conditions: a list of `pyowm.alertapi30.condition.Condition` instances
- area: a list of `pyowm.utils.geo.Geometry` instances, representing the geographic area on which the trigger's conditions need to be checked

- alertChannels: list of `pyowm.alertapi30.alert.AlertChannel` objects, representing which channels this trigger is notifying to

Notes on trigger's time period By design, PyOWM will only use the `after` operator to communicate time periods for Triggers to the Alert API. will send them to the API using the `after` operator.

The millisecond start/end deltas will be calculated with respect to the time when the Trigger record is created on the Alert API using `pyowm.alertapi30.alertmanager.AlertManager.create_trigger`

AlertManager

The OWM main entry point object allows you to get an instance of an `pyowm.alertapi30.alert_manager.AlertManager` object: use it to interact with the Alert API and create/read/update/delete triggers and read/delete the related alerts.

Here is how to instantiate an `AlertManager`:

```python
from pyowm import OWM

owm = OWM(API_Key='my-API-key')
am = owm.alert_manager()
```

Then you can do some nice things with it:

- create a trigger
- read all of your triggers
- read a named trigger
- modify a named trigger
- delete a named trigger
- read all the alerts fired by a named trigger
- read a named alert
- delete a named alert
- delete all of the alerts for a named trigger

5.7 Map tiles client examples

5.7.1 Tiles client

OWM provides tiles for a few map layers displaying world-wide features such as global temperature, pressure, wind speed, and precipitation amount.

Each tile is a PNG image that is referenced by a triplet: the (x, y) coordinates and a zoom level

The zoom level might depend on the type of layers: 0 means no zoom (full globe covered), while usually you can get up to a zoom level of 18.

Available map layers are specified by the `pyowm.tiles.enums.MapLayerEnum` values.

OWM website technical reference

- http://openweathermap.org/api/weathermaps

Usage examples

Tiles can be fetched this way:

```
from pyowm import OWM
from pyowm.tiles.enums import MapLayerEnum

owm = OWM('my-API-key')

# Choose the map layer you want tiles for (eg. temeperature
layer_name = MapLayerEnum.TEMPERATURE

# Obtain an instance to a tile manager object
tm = owm.tile_manager(layer_name)

# Now say you want tile at coordinate x=5 y=2 at a zoom level of 6
tile = tm.get_tile(5, 2, 6)

# You can now save the tile to disk
tile.persist('/path/to/file.png')

# Wait! but now I need the pressure layer tile at the very same coordinates and zoom␣
↪level! No worries...
# Just change the map layer name on the TileManager and off you go!
tm.map_layer = MapLayerEnum.PRESSURE
tile = tm.get_tile(5, 2, 6)
```

Tile object

A `pyowm.commons.tile.Tile` object is a wrapper for the tile coordinates and the image data, which is a `pyowm.commons.image.Image` object instance.

You can save a tile to disk by specifying a target file:

```
tile.persist('/path/to/file.png')
```

Use cases

I have the lon/lat of a point and I want to get the tile that contains that point at a given zoom level

Turn the lon/lat couple to a `pyowm.utils.geo.Point` object and pass it

```
from pyowm.utils.geo import Point
from pyowm.commons.tile import Tile

geopoint = Point(lon, lat)
x_tile, y_tile = Tile.tile_coords_for_point(geopoint, zoom_level):
```

I have a tile and I want to know its bounding box in lon/lat coordinates

Easy! You'll get back a `pyowm.utils.geo.Polygon` object, from which you can extract lon/lat coordinates this way

```
polygon = tile.bounding_polygon()
geopoints = polygon.points
geocoordinates = [(p.lon, p.lat) for p in geopoints]  # this gives you tuples with␣
↪lon/lat
```

CHAPTER 6

PyOWM v3 usage documentation

Coming soon!

CHAPTER 7

PyOWM software API documentation

This is the Python API documentation of PyOWM:

7.1 pyowm package

7.1.1 Subpackages

pyowm.abstractions package

Submodules

pyowm.abstractions.jsonparser module

Module containing an abstract base class for JSON OWM Weather API responses parsing

class `pyowm.abstractions.jsonparser.`**`JSONParser`**
Bases: `object`

A global abstract class representing a JSON to object parser.

`parse_JSON`(*JSON_string*)
Returns a proper object parsed from the input JSON_string. Subclasses know from their specific type which object is to be parsed and returned

Parameters **`JSON_string`** (`str`) – a JSON text string

Returns an object

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the resulting object

pyowm.abstractions.linkedlist module

Module containing abstractions for defining a linked list data structure

class `pyowm.abstractions.linkedlist.`**`LinkedList`**

Bases: `object`

An abstract class representing a Linked List data structure. Each element in the list should contain data and a reference to the next element in the list.

`add`(*data*)

Adds a new node to the list. Implementations should decide where to put this new element (at the top, in the middle or at the end of the list) and should therefore update pointers to next elements and the list's size.

Parameters **`data`** (*`object`*) – the data to be inserted in the new list node

`contains`(*data*)

Checks if the provided data is stored in at least one node of the list.

Parameters **`data`** (*`object`*) – data of the seeked node

Returns a boolean

`index_of`(*data*)

Finds the position of a node in the list. The index of the first occurrence of the data is returned (indexes start at 0)

Parameters **`data`** – data of the seeked node

Type object

Returns the int index or -1 if the node is not in the list

`pop`()

Removes the last node from the list

Returns the object data that was stored in the last node

`remove`(*data*)

Removes a node from the list. Implementations should decide the policy to be followed when list items having the same data are to be removed, and should therefore update pointers to next elements and the list's size.

Parameters **`data`** (*`object`*) – the data to be removed in the new list node

`size`()

Returns the number of elements in the list

Returns an int

pyowm.abstractions.owm module

Module containing the abstract PyOWM library main entry point interface

class `pyowm.abstractions.owm.`**`OWM`**

Bases: `object`

A global abstract class representing the OWM Weather API. Every query to the API is done programmatically via a concrete instance of this class. Subclasses should provide a method for every OWM Weather API endpoint.

`get_API_key`()

Returns the OWM API key

Returns the OWM API key string

get_API_version()
Returns the currently supported OWM Weather API version

Returns the OWM Weather API version string

get_version()
Returns the current version of the PyOWM library

Returns the current PyOWM library version string

is_API_online()
Returns `True` if the OWM Weather API is currently online. A short timeout is used to determine API service availability.

Returns bool

set_API_key(*API_key*)
Updates the OWM API key

Parameters **API_key** (*str*) – the new value for the OWM API key

pyowm.abstractions.owmcache module

Module containing the abstract PyOWM cache provider

class pyowm.abstractions.owmcache.**OWMCache**
Bases: `object`

A global abstract class representing a caching provider which can be used to lookup the JSON responses to the most recently or most frequently issued OWM Weather API requests. The purpose of the caching mechanism is to avoid OWM Weather API requests and therefore network traffic: the implementations should be adapted to the time/memory requirements of the OWM data clients (i.e: a "slimmer" cache with lower lookup times but higher miss rates or a "fatter" cache with higher memory consumption and higher hit rates?). Subclasses should implement a proper caching algorithms bearing in mind that different weather data types may have different change rates: in example, observed weather can change very frequently while long-period weather forecasts change less frequently. External caching mechanisms (eg: memcached, redis, etc..) can be used by extending this class into a proper decorator for the correspondent Python bindings.

get(*request_url*)
In case of a hit, returns the JSON string which represents the OWM web API response to the request being identified by a specific string URL.

Parameters **request_url** (*str*) – an URL that uniquely identifies the request whose response is to be looked up

Returns a JSON str in case of cache hit or `None` otherwise

set(*request_url*, *response_json*)
Adds the specified response_json value to the cache using as a lookup key the request_url of the request that generated the value.

Parameters

- **request_url** (*str*) – the request URL
- **response_json** (*str*) – the response JSON

Module contents

pyowm.agroapi10 package

Submodules

pyowm.agroapi10.agro_manager module

pyowm.agroapi10.enums module

class pyowm.agroapi10.enums.**PaletteEnum**

Bases: `object`

Allowed color palettes for satellite images on Agro API 1.0

BLACK_AND_WHITE = '2'

CONTRAST_CONTINUOUS = '4'

CONTRAST_SHIFTED = '3'

GREEN = '1'

classmethod items()

All values for this enum :return: list of str

class pyowm.agroapi10.enums.**PresetEnum**

Bases: `object`

Allowed presets for satellite images on Agro API 1.0

EVI = 'evi'

FALSE_COLOR = 'falsecolor'

NDVI = 'ndvi'

TRUE_COLOR = 'truecolor'

classmethod items()

All values for this enum :return: list of str

class pyowm.agroapi10.enums.**SatelliteEnum**

Bases: `object`

Allowed presets for satellite names on Agro API 1.0

LANDSAT_8 = <pyowm.commons.databoxes.Satellite - name=Landsat 8 symbol=l8>

SENTINEL_2 = <pyowm.commons.databoxes.Satellite - name=Sentinel-2 symbol=s2>

classmethod items()

All values for this enum :return: list of str

pyowm.agroapi10.imagery module

pyowm.agroapi10.polygon module

pyowm.agroapi10.search module

pyowm.agroapi10.soil module

class `pyowm.agroapi10.soil.`**`Soil`**(*reference_time*, *surface_temp*, *ten_cm_temp*, *moisture*, *polygon_id=None*)

Bases: `object`

Soil data over a specific Polygon

Parameters

- **`reference_time`** (*`int`*) – UTC UNIX time of soil data measurement
- **`surface_temp`** (*`float`*) – soil surface temperature in Kelvin degrees
- **`ten_cm_temp`** (*`float`*) – soil temperature at 10 cm depth in Kelvin degrees
- **`moisture`** (*`float`*) – soil moisture in m^3/m^3
- **`polygon_id`** (*`str`*) – ID of the polygon this soil data was measured upon

Returns a *Soil* instance

Raises *AssertionError* when any of the mandatory fields is *None* or has wrong type

classmethod from_dict(*the_dict*)

`reference_time`(*timeformat='unix'*)

Returns the UTC time telling when the soil data was measured

Parameters **`timeformat`** (*`str`*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

`surface_temp`(*unit='kelvin'*)

Returns the soil surface temperature

Parameters **`unit`** (*`str`*) – the unit of measure for the temperature value. May be: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a float

Raises ValueError when unknown temperature units are provided

`ten_cm_temp`(*unit='kelvin'*)

Returns the soil temperature measured 10 cm below surface

Parameters **`unit`** (*`str`*) – the unit of measure for the temperature value. May be: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a float

Raises ValueError when unknown temperature units are provided

pyowm.agroapi10.uris module

URIs templates for resources exposed by the Agro API 1.0

Module contents

pyowm.alertapi30 package

Submodules

pyowm.alertapi30.alert_manager module

pyowm.alertapi30.alert module

class pyowm.alertapi30.alert.**Alert**(*id*, *trigger_id*, *met_conditions*, *coordinates*, *last_update=None*)
Bases: `object`

Represents the situation happening when any of the conditions bound to a *Trigger* is met. Whenever this happens, an *Alert* object is created (or updated) and is bound to its parent *Trigger*. The trigger can then be polled to check what alerts have been fired on it. :param id: unique alert identifier :type name: str :param trigger_id: link back to parent *Trigger* :type trigger_id: str :param met_conditions: list of dict, each one referring to a *Condition* obj bound to the parent *Trigger* and reporting the actual measured values that made this *Alert* fire :type met_conditions: list of dict :param coordinates: dict representing the geocoordinates where the *Condition* triggering the *Alert* was met :type coordinates: dict :param last_update: epoch of the last time when this *Alert* has been fired :type last_update: int

class pyowm.alertapi30.alert.**AlertChannel**(*name*)
Bases: `object`

Base class representing a channel through which one can acknowledge that a weather alert has been issued. Examples: OWM API polling, push notifications, email notifications, etc. This feature is yet to be implemented by the OWM API. :param name: name of the channel :type name: str :returns: an *AlertChannel* instance

pyowm.alertapi30.condition module

class pyowm.alertapi30.condition.**Condition**(*weather_param*, *operator*, *amount*, *id=None*)
Bases: `object`

Object representing a condition to be checked on a specific weather parameter. A condition is given when comparing the weather parameter against a numerical value with respect to an operator. Allowed weather params and operators are specified by the *pyowm.utils.alertapi30.WeatherParametersEnum* and *pyowm.utils.alertapi30.OperatorsEnum* enumerator classes. :param weather_param: the weather variable to be checked (eg. TEMPERATURE, CLOUDS, …) :type weather_param: str :param operator: the comparison operator to be applied to the weather variable (eg. GREATER_THAN, EQUAL, …) :type operator: str :param amount: comparison value :type amount: int or float :param id: optional unique ID for this Condition instance :type id: str :returns: a *Condition* instance :raises: *AssertionError* when either the weather param has wrong type or the operator has wrong type or the amount has wrong type

classmethod from_dict(*the_dict*)

pyowm.alertapi30.enums module

class pyowm.alertapi30.enums.**AlertChannelsEnum**

Bases: `object`

Allowed alert channels

OWM_API_POLLING = <pyowm.alertapi30.alert.AlertChannel object>

classmethod items()

All values for this enum :return: list of str

class pyowm.alertapi30.enums.**OperatorsEnum**

Bases: `object`

Allowed comparison operators for condition checking upon weather parameters

EQUAL = '$eq'

GREATER_THAN = '$gt'

GREATER_THAN_EQUAL = '$gte'

LESS_THAN = '$lt'

LESS_THAN_EQUAL = '$lte'

NOT_EQUAL = '$ne'

classmethod items()

All values for this enum :return: list of str

class pyowm.alertapi30.enums.**WeatherParametersEnum**

Bases: `object`

Allowed weather parameters for condition checking

CLOUDS = 'clouds'

HUMIDITY = 'humidity'

PRESSURE = 'pressure'

TEMPERATURE = 'temp'

WIND_DIRECTION = 'wind_direction'

WIND_SPEED = 'wind_speed'

classmethod items()

All values for this enum :return: list of str

pyowm.alertapi30.trigger module

class pyowm.alertapi30.trigger.**Trigger**(*start_after_millis*, *end_after_millis*, *conditions*, *area*, *alerts=None*, *alert_channels=None*, *id=None*)

Bases: `object`

Object representing a the check if a set of weather conditions are met on a given geographical area: each condition is a rule on the value of a given weather parameter (eg. humidity, temperature, etc). Whenever a condition from a *Trigger* is met, the OWM API crates an alert and binds it to the the *Trigger*. A *Trigger* is the local proxy for the corresponding entry on the OWM API, therefore it can get ouf of sync as time goes by and conditions are met: it's up to you to "refresh" the local trigger by using a *pyowm.utils.alertapi30.AlertManager* instance. :param start_after_millis: how many milliseconds after the trigger creation the trigger begins to be checked :type start_after_millis: int :param end_after_millis: how many milliseconds after the trigger creation the trigger ends to be checked :type end_after_millis: int :param alerts: the *Alert* objects representing the alerts that have been fired for this *Trigger* so far. Defaults to *None* :type alerts: list of *pyowm.utils.alertapi30.Alert* instances :param conditions: the *Condition* objects representing the set of checks to be done on weather variables :type conditions: list of *pyowm.utils.alertapi30.Condition* instances :param area: the geographic are over which conditions are checked: it can be composed by multiple geoJSON types :type area: list of geoJSON types :param alert_channels: the alert channels through which alerts originating from this *Trigger* can be consumed. Defaults to OWM API polling :type alert_channels: list of *pyowm.utils.alertapi30.AlertChannel* instances :param id: optional unique ID for this *Trigger* instance :type id: str :returns: a *Trigger* instance :raises: *ValueError* when start or end epochs are *None* or when end precedes start or when conditions or area are empty collections

get_alert(*alert_id*)
Returns the *Alert* of this *Trigger* having the specified ID :param alert_id: str, the ID of the alert :return: *Alert* instance

get_alerts()
Returns all of the alerts for this *Trigger* :return: a list of *Alert* objects

get_alerts_on(*weather_param*)
Returns all the *Alert* objects of this *Trigger* that refer to the specified weather parameter (eg. 'temp', 'pressure', etc.). The allowed weather params are the ones enumerated by class *pyowm.alertapi30.enums.WeatherParametersEnum* :param weather_param: str, values in *pyowm.alertapi30.enums.WeatherParametersEnum* :return: list of *Alert* instances

get_alerts_since(*timestamp*)
Returns all the *Alert* objects of this *Trigger* that were fired since the specified timestamp. :param timestamp: time object representing the point in time since when alerts have to be fetched :type timestamp: int, `datetime.datetime` or ISO8601-formatted string :return: list of *Alert* instances

Module contents

pyowm.caches package

Submodules

pyowm.caches.lrucache module

Module containing LRU cache related class

class `pyowm.caches.lrucache.`**LRUCache**(*cache_max_size=20*, *item_lifetime_millis=600000*)
Bases: `pyowm.abstractions.owmcache.OWMCache`

This cache is made out of a 'table' dict and the 'usage_recency' linked list.'table' maps uses requests' URLs as keys and stores JSON raw responses as values. 'usage_recency' tracks down the "recency" of the OWM Weather API requests: the more recent a request, the more the element will be far from the "death" point of the recency list. Items in 'usage_recency' are the requests' URLs themselves. The implemented LRU caching mechanism is the following:

- cached elements must expire after a certain time passed into the cache. So when an element is looked up and found in the cache, its insertion timestamp is compared to the current one: if the difference is higher than a prefixed value, then the lookup is considered a MISS: the element is removed either from 'table' and from 'usage_recency' and must be requested again to the OWM Weather API. If the time difference is ok, then the lookup is considered a HIT.
- when a GET results in a HIT, promote the element to the front of the recency list updating its cache insertion timestamp and return the data to the cache clients

- when a GET results in a MISS, return `None`
- when a SET is issued, check if the maximum size of the cache has been reached: if so, discard the least recently used item from the recency list and the dict; then add the element to 'table' recording its timestamp and finally add it at the front of the recency list.

Parameters

- **`cache_max_size`** (*int*) – the maximum size of the cache in terms of cached OWM Weather API responses. A reasonable default value is provided.
- **`item_lifetime_millis`** (*int*) – the maximum lifetime allowed for a cache item in milliseconds. A reasonable default value is provided.

Returns a new *LRUCache* instance

`clean`()

Empties the cache

`get`(*request_url*)

In case of a hit, returns the JSON string which represents the OWM web API response to the request being identified by a specific string URL and updates the recency of this request.

Parameters **`request_url`** (*str*) – an URL that uniquely identifies the request whose response is to be looked up

Returns a JSON str in case of cache hit or `None` otherwise

`set`(*request_url*, *response_json*)

Checks if the maximum size of the cache has been reached and in case discards the least recently used item from 'usage_recency' and 'table'; then adds the response_json to be cached to the 'table' dict using as a lookup key the request_url of the request that generated the value; finally adds it at the front of 'usage_recency'

Parameters

- **`request_url`** (*str*) – the request URL that uniquely identifies the request whose response is to be cached
- **`response_json`** (*str*) – the response JSON to be cached

`size`()

Returns the number of elements that are currently stored into the cache

Returns an int

pyowm.caches.nullcache module

Module containing a null-object cache for OWM Weather API responses

class `pyowm.caches.nullcache.`**`NullCache`**

Bases: `pyowm.abstractions.owmcache.OWMCache`

A null-object implementation of the *OWMCache* abstract class

`get`(*request_url*)

Always returns `None` (nothing will ever be cached or looked up!)

Parameters **`request_url`** (*str*) – the request URL

Returns `None`

set (*request_url*, *response_json*)

Does nothing.

Parameters

- **request_url** (*str*) – the request URL
- **response_json** (*str*) – the response JSON

Module contents

pyowm.commons package

Submodules

pyowm.commons.databoxes module

class pyowm.commons.databoxes.**ImageType** (*name*, *mime_type*)

Bases: `object`

Databox class representing an image type

Parameters

- **name** (*str*) – the image type name
- **mime_type** (*str*) – the image type MIME type

class pyowm.commons.databoxes.**Satellite** (*name*, *symbol*)

Bases: `object`

Databox class representing a satellite

Parameters

- **name** (*str*) – the satellite
- **symbol** (*str*) – the short name of the satellite

pyowm.commons.enums module

class pyowm.commons.enums.**ImageTypeEnum**

Bases: `object`

Allowed image types on OWM APIs

GEOTIFF = <pyowm.commons.databoxes.ImageType - name=GEOTIFF mime=image/tiff>

PNG = <pyowm.commons.databoxes.ImageType - name=PNG mime=image/png>

classmethod items ()

All values for this enum :return: list of *pyowm.commons.enums.ImageType*

classmethod lookup_by_mime_type (*mime_type*)

classmethod lookup_by_name (*name*)

pyowm.commons.frontlinkedlist module

Module containing class related to the implementation of linked-list data structure

class pyowm.commons.frontlinkedlist.**FrontLinkedList**

Bases: *pyowm.abstractions.linkedlist.LinkedList*

Implementation of a linked-list data structure. Insertions are performed at the front of the list and so are O(1) while deletions take O(n) because they can be performed against any of the linked list's elements. Each element in the list is a LinkedListNode instance; after instantiation, the list contains no elements.

Parameters

- **first_node** (LinkedListNode) – reference to the first LinkedListNode element in the list
- **last_node** (LinkedListNode) – reference to the last LinkedListNode element in the list

add(*data*)

Adds a new data node to the front list. The provided data will be encapsulated into a new instance of LinkedListNode class and linked list pointers will be updated, as well as list's size.

Parameters **data** (*object*) – the data to be inserted in the new list node

contains(*data*)

Checks if the provided data is stored in at least one node of the list.

Parameters **data** (*object*) – the seeked data

Returns a boolean

first_node()

index_of(*data*)

Finds the position of a node in the list. The index of the first occurrence of the data is returned (indexes start at 0)

Parameters **data** – data of the seeked node

Type object

Returns the int index or -1 if the node is not in the list

pop()

Removes the last node from the list

remove(*data*)

Removes a data node from the list. If the list contains more than one node having the same data that shall be removed, then the node having the first occurrency of the data is removed.

Parameters **data** (*object*) – the data to be removed in the new list node

size()

Returns the number of elements in the list

Returns an int

class pyowm.commons.frontlinkedlist.**FrontLinkedListIterator**(*obj*)

Bases: object

Iterator over the LinkedListNode elements of a LinkedList class instance. The implementation keeps a copy of the iterated list so avoid concurrency problems when iterating over it. This can nevertheless be memory-consuming when big lists have to be iterated over.

Parameters **obj** (*object*) – the iterable object (LinkedList)

Returns a FrontLinkedListIterator instance

next()

Compatibility for Python 2.x, delegates to function: *__next__()* Returns the next *Weather* item

Returns the next *Weather* item

class pyowm.commons.frontlinkedlist.**LinkedListNode**(*data*, *next_node*)

Bases: object

Class representing an element of the LinkedList

Parameters

- **data** (*object*) – the actual data that this node holds
- **next** (LinkedListNode) – reference to the next LinkedListNode instance in the list

data()

Returns the data in this node

Returns an object

next()

Returns the next LinkedListNode in the list

Returns a LinkedListNode instance

update_next(*linked_list_node*)

Parameters **linked_list_node** (LinkedListNode) – the new reference to the next LinkedListNode element

pyowm.commons.http_client module

pyowm.commons.image module

class pyowm.commons.image.**Image**(*data*, *image_type=None*)

Bases: object

Wrapper class for a generic image

Parameters

- **data** (*bytes*) – raw image data
- **image_type** (*pyowm.commons.databoxes.ImageType* or *None*) – the type of the image, if known

classmethod load(*path_to_file*)

Loads the image data from a file on disk and tries to guess the image MIME type

Parameters **path_to_file** (*str*) – path to the source file

Returns a *pyowm.image.Image* instance

persist(*path_to_file*)

Saves the image to disk on a file

Parameters **path_to_file** (*str*) – path to the target file

Returns *None*

pyowm.commons.tile module

Module contents

pyowm.exceptions package

Submodules

pyowm.exceptions.api_call_error module

Module containing APICallError class

exception pyowm.exceptions.api_call_error.**APICallError**(*message*, *triggering_error=None*)

Bases: *pyowm.exceptions.OWMError*

Error class that represents network/infrastructural failures when invoking OWM Weather API, in example due to network errors.

Parameters

- **message** (*str*) – the message of the error
- **triggering_error** (an *Exception* subtype) – optional *Exception* object that triggered this error (defaults to None)

exception pyowm.exceptions.api_call_error.**APICallTimeoutError**(*message*, *triggering_error=None*)

Bases: *pyowm.exceptions.api_call_error.APICallError*

Error class that represents response timeout conditions

Parameters

- **message** (*str*) – the message of the error
- **triggering_error** (an *Exception* subtype) – optional *Exception* object that triggered this error (defaults to None)

exception pyowm.exceptions.api_call_error.**APIInvalidSSLCertificateError**(*message*, *triggering_error=None*)

Bases: *pyowm.exceptions.api_call_error.APICallError*

Error class that represents failure in verifying the SSL certificate provided by the OWM API

Parameters

- **message** (*str*) – the message of the error
- **triggering_error** (an *Exception* subtype) – optional *Exception* object that triggered this error (defaults to None)

exception pyowm.exceptions.api_call_error.**BadGatewayError**(*message*, *triggering_error=None*)

Bases: *pyowm.exceptions.api_call_error.APICallError*

Error class that represents 502 errors - i.e when upstream backend cannot communicate with API gateways.

Parameters

- **message** (*str*) – the message of the error

- **triggering_error** (an *Exception* subtype) – optional *Exception* object that triggered this error (defaults to `None`)

pyowm.exceptions.api_response_error module

Module containing APIResponseError class

exception `pyowm.exceptions.api_response_error.`**APIResponseError**(*cause*, *status_code*)

Bases: *pyowm.exceptions.OWMError*

Error class that represents HTTP error status codes in OWM Weather API responses.

Parameters

- **cause** (*str*) – the message of the error
- **status_code** (*int*) – the HTTP error status code

Returns a *APIResponseError* instance

exception `pyowm.exceptions.api_response_error.`**NotFoundError**(*cause*, *status_code=404*)

Bases: *pyowm.exceptions.api_response_error.APIResponseError*

Error class that represents the situation when an entity is not found into a collection of entities.

Parameters

- **cause** (*str*) – the message of the error
- **status_code** (*int*) – the HTTP error status code

Returns a *NotFoundError* instance

exception `pyowm.exceptions.api_response_error.`**UnauthorizedError**(*cause*, *status_code=403*)

Bases: *pyowm.exceptions.api_response_error.APIResponseError*

Error class that represents the situation when an entity cannot be retrieved due to user subscription unsufficient capabilities.

Parameters

- **cause** (*str*) – the message of the error
- **status_code** (*int*) – the HTTP error status code

Returns a *UnauthorizedError* instance

pyowm.exceptions.parse_response_error module

Module containing ParseResponseError class

exception `pyowm.exceptions.parse_response_error.`**ParseResponseError**(*cause*)

Bases: *pyowm.exceptions.OWMError*

Error class that represents failures when parsing payload data in HTTP responses sent by the OWM Weather API.

Parameters **cause** (*str*) – the message of the error

Returns a *ParseResponseError* instance

Module contents

Module containing the OWMError class as base for all other OWM errors

exception `pyowm.exceptions.`**`OWMError`**
Bases: `Exception`

pyowm.pollutionapi30 package

Subpackages

Submodules

pyowm.pollutionapi30.airpollution_client module

pyowm.pollutionapi30.coindex module

Carbon Monoxide classes and data structures.

class `pyowm.pollutionapi30.coindex.`**`COIndex`**(*reference_time*, *location*, *interval*, *co_samples*, *reception_time*)
Bases: `object`

A class representing the Carbon monOxide Index observed in a certain location in the world. The index is made up of several measurements, each one at a different atmospheric pressure. The location is represented by the encapsulated *Location* object.

Parameters

- **reference_time** (*int*) – GMT UNIXtime telling when the CO data has been measured
- **location** (*Location*) – the *Location* relative to this CO observation
- **interval** (*str*) – the time granularity of the CO observation
- **co_samples** (*list of dicts*) – the CO samples
- **reception_time** (*int*) – GMT UNIXtime telling when the CO observation has been received from the OWM Weather API

Returns an *COIndex* instance

Raises *ValueError* when negative values are provided as reception time, CO samples are not provided in a list

get_co_sample_with_highest_vmr()
Returns the CO sample with the highest Volume Mixing Ratio value :return: dict

get_co_sample_with_lowest_vmr()
Returns the CO sample with the lowest Volume Mixing Ratio value :return: dict

get_co_samples()
Returns the CO samples for this index

Returns list of dicts

get_interval()
Returns the time granularity interval for this CO index measurement

Returns str

get_location()
Returns the *Location* object for this CO index measurement

Returns the *Location* object

get_reception_time(*timeformat='unix'*)
Returns the GMT time telling when the CO observation has been received from the OWM Weather API

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

get_reference_time(*timeformat='unix'*)
Returns the GMT time telling when the CO samples have been measured

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

is_forecast()
Tells if the current CO observation refers to the future with respect to the current date :return: bool

to_JSON()
Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.pollutionapi30.ozone module

class pyowm.pollutionapi30.ozone.**Ozone**(*reference_time*, *location*, *interval*, *du_value*, *reception_time*)
Bases: `object`

A class representing the Ozone (O3) data observed in a certain location in the world. The location is represented by the encapsulated *Location* object.

Parameters

- **reference_time** (*int*) – GMT UNIXtime telling when the O3 data have been measured
- **location** (*Location*) – the *Location* relative to this O3 observation

- **`du_value`** (*`float`*) – the observed O3 Dobson Units value (reference: http://www.theozonehole.com/dobsonunit.htm)
- **`interval`** (*`str`*) – the time granularity of the O3 observation
- **`reception_time`** (*`int`*) – GMT UNIXtime telling when the observation has been received from the OWM Weather API

Returns an *Ozone* instance

Raises *ValueError* when negative values are provided as reception time or du_value

`get_du_value`()

Returns the O3 Dobson Unit of this observation

Returns float

`get_interval`()

Returns the time granularity interval for this O3 observation

Returns str

`get_location`()

Returns the *Location* object for this O3 observation

Returns the *Location* object

`get_reception_time`(*timeformat='unix'*)

Returns the GMT time telling when the O3 observation has been received from the OWM Weather API

Parameters **`timeformat`** (*`str`*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

`get_reference_time`(*timeformat='unix'*)

Returns the GMT time telling when the O3 data have been measured

Parameters **`timeformat`** (*`str`*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

`is_forecast`()

Tells if the current O3 observation refers to the future with respect to the current date :return: bool

`to_JSON`()

Dumps object fields into a JSON formatted string

Returns the JSON string

`to_XML`(*xml_declaration=True*, *xmlns=True*)

Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **`XML_declaration`** (*`bool`*) – if `True` (default) prints a leading XML declaration line

- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.pollutionapi30.no2index module

Nitrogen Dioxide classes and data structures.

class pyowm.pollutionapi30.no2index.**NO2Index**(*reference_time*, *location*, *interval*, *no2_samples*, *reception_time*)

Bases: `object`

A class representing the Nitrogen DiOxide Index observed in a certain location in the world. The index is made up of several measurements, each one at a different atmospheric levels. The location is represented by the encapsulated *Location* object.

Parameters

- **reference_time** (*int*) – GMT UNIXtime telling when the NO2 data has been measured
- **location** (*Location*) – the *Location* relative to this NO2 observation
- **interval** (*str*) – the time granularity of the NO2 observation
- **no2_samples** (*list of dicts*) – the NO2 samples
- **reception_time** (*int*) – GMT UNIXtime telling when the NO2 observation has been received from the OWM Weather API

Returns a *NO2Index* instance

Raises *ValueError* when negative values are provided as reception time, NO2 samples are not provided in a list

get_interval()

Returns the time granularity interval for this NO2 index measurement

Returns str

get_location()

Returns the *Location* object for this NO2 index measurement

Returns the *Location* object

get_no2_samples()

Returns the NO2 samples for this index

Returns list of dicts

get_reception_time(*timeformat='unix'*)

Returns the GMT time telling when the NO2 observation has been received from the OWM Weather API

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

get_reference_time(*timeformat='unix'*)

Returns the GMT time telling when the NO2 samples have been measured

Parameters **`timeformat`** (*`str`*) – the format for the time value. May be: ‘*unix*’ (default) for UNIX time ‘*iso*’ for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` ‘*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

`get_sample_by_label`(*label*)
Returns the NO2 sample having the specified label or *None* if none is found

Parameters **`label`** – the label for the seeked NO2 sample

Returns dict or *None*

`is_forecast`()
Tells if the current NO2 observation refers to the future with respect to the current date :return: bool

`to_JSON`()
Dumps object fields into a JSON formatted string

Returns the JSON string

`to_XML`(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The ‘xml_declaration’ switch enables printing of a leading standard XML line containing XML version and encoding. The ‘xmlns’ switch enables printing of qualified XMLNS prefixes.

Parameters

- **`XML_declaration`** (*`bool`*) – if `True` (default) prints a leading XML declaration line
- **`xmlns`** (*`bool`*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.pollutionapi30.so2index module

Sulphur Dioxide classes and data structures.

class `pyowm.pollutionapi30.so2index.`**`SO2Index`**(*reference_time*, *location*, *interval*, *so2_samples*, *reception_time*)
Bases: `object`

A class representing the Sulphur Dioxide Index observed in a certain location in the world. The index is made up of several measurements, each one at a different atmospheric pressure. The location is represented by the encapsulated *Location* object.

Parameters

- **`reference_time`** (*`int`*) – GMT UNIXtime telling when the SO2 data has been measured
- **`location`** (*Location*) – the *Location* relative to this SO2 observation
- **`interval`** (*`str`*) – the time granularity of the SO2 observation
- **`so2_samples`** (*`list of dicts`*) – the SO2 samples
- **`reception_time`** (*`int`*) – GMT UNIXtime telling when the SO2 observation has been received from the OWM Weather API

Returns an *SOIndex* instance

Raises *ValueError* when negative values are provided as reception time, SO2 samples are not provided in a list

get_interval()
Returns the time granularity interval for this SO2 index measurement

Returns str

get_location()
Returns the *Location* object for this SO2 index measurement

Returns the *Location* object

get_reception_time(*timeformat='unix'*)
Returns the GMT time telling when the SO2 observation has been received from the OWM Weather API

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

get_reference_time(*timeformat='unix'*)
Returns the GMT time telling when the SO2 samples have been measured

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

get_so2_samples()
Returns the SO2 samples for this index

Returns list of dicts

is_forecast()
Tells if the current SO2 observation refers to the future with respect to the current date :return: bool

to_JSON()
Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.pollutionapi30.parsers module

Module contents

pyowm.stationsapi30 package

Subpackages

pyowm.stationsapi30.parsers package

Submodules

pyowm.stationsapi30.parsers.aggregated_measurement_parser module

Module containing a concrete implementation for JSONParser abstract class, returning an AggregatedMeasurement instance

class `pyowm.stationsapi30.parsers.aggregated_measurement_parser.`**`AggregatedMeasurementParse`**

Bases: *`pyowm.abstractions.jsonparser.JSONParser`*

Concrete *JSONParser* implementation building a *pyowm.stationsapi30.measurement.AggregatedMeasurement* instance out of raw JSON data

`parse_JSON`(*JSON_string*)

Parses a *pyowm.stationsapi30.measurement.AggregatedMeasurement* instance out of raw JSON data.

Parameters **`JSON_string`** (*`str`*) – a raw JSON string

Returns a *pyowm.stationsapi30.measurement.AggregatedMeasurement* instance or `None` if no data is available

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the result

`parse_dict`(*data_dict*)

Parses a dictionary representing the attributes of a *pyowm.stationsapi30.smeasurement.AggregatedMeasurement* entity :param data_dict: dict :return: *pyowm.stationsapi30.measurement.AggregatedMeasurement*

pyowm.stationsapi30.parsers.station_parser module

Module containing a concrete implementation for JSONParser abstract class, returning a Station instance

class `pyowm.stationsapi30.parsers.station_parser.`**`StationParser`**

Bases: *`pyowm.abstractions.jsonparser.JSONParser`*

Concrete *JSONParser* implementation building a *pyowm.stationsapi30.station.Station* instance out of raw JSON data

`parse_JSON`(*JSON_string*)

Parses a *pyowm.stationsapi30.station.Station* instance out of raw JSON data.

Parameters **`JSON_string`** (*`str`*) – a raw JSON string

Returns a *pyowm.stationsapi30.station.Station* instance or `None` if no data is available

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the result

parse_dict(*data_dict*)
Parses a dictionary representing the attributes of a *pyowm.stationsapi30.station.Station* entity :param data_dict: dict :return: *pyowm.stationsapi30.station.Station*

Module contents

pyowm.stationsapi30.xsd package

Submodules

pyowm.stationsapi30.xsd.xmlnsconfig module

Module contents

Submodules

pyowm.stationsapi30.buffer module

class `pyowm.stationsapi30.buffer.`**Buffer**(*station_id*)
Bases: `object`

append(*measurement*)
Appends the specified `Measurement` object to the buffer :param measurement: a `measurement.Measurement` instance

append_from_dict(*the_dict*)
Creates a `measurement.Measurement` object from the supplied dict and then appends it to the buffer :param the_dict: dict

append_from_json(*json_string*)
Creates a `measurement.Measurement` object from the supplied JSON string and then appends it to the buffer :param json_string: the JSON formatted string

created_at = None

creation_time(*timeformat='unix'*)
Returns the UTC time of creation of this aggregated measurement

Parameters timeformat (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time, '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` or *date* for a `datetime.datetime` object

Returns an int or a str or a `datetime.datetime` object or None

Raises ValueError

empty()
Drops all measurements of this buffer instance

measurements = None

sort_chronologically()
Sorts the measurements of this buffer in chronological order

sort_reverse_chronologically()
Sorts the measurements of this buffer in reverse chronological order

station_id = None

pyowm.stationsapi30.measurement module

class pyowm.stationsapi30.measurement.**AggregatedMeasurement**(*station_id*, *timestamp*, *aggregated_on*, *temp=None*, *humidity=None*, *wind=None*, *pressure=None*, *precipitation=None*)

Bases: `object`

A class representing an aggregation of measurements done by the Stations API on a specific time-frame. Values for the aggregation time-frame can be: 'm' (minute), 'h' (hour) or 'd' (day)

Parameters

- **station_id** (*str*) – unique station identifier
- **timestamp** (*int*) – reference UNIX timestamp for this measurement
- **aggregated_on** (*string between 'm','h' and 'd'*) – aggregation time-frame for this measurement
- **temp** (dict or *None*) – optional dict containing temperature data
- **humidity** (dict or *None*) – optional dict containing humidity data
- **wind** (dict or *None*) – optional dict containing wind data
- **pressure** (dict or *None*) – optional dict containing pressure data
- **precipitation** (dict or *None*) – optional dict containing precipitation data

ALLOWED_AGGREGATION_TIME_FRAMES = ['m', 'h', 'd']

creation_time(*timeformat='unix'*)

Returns the UTC time of creation of this aggregated measurement

Parameters timeformat (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time, '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` or *date* for a `datetime.datetime` object

Returns an int or a str or a `datetime.datetime` object or None

Raises ValueError

to_JSON()

Dumps object fields into a JSON formatted string

Returns the JSON string

to_dict()

Dumps object fields into a dict

Returns a dict

class `pyowm.stationsapi30.measurement.`**`Measurement`**(*station_id*, *timestamp*, *temperature=None*, *wind_speed=None*, *wind_gust=None*, *wind_deg=None*, *pressure=None*, *humidity=None*, *rain_1h=None*, *rain_6h=None*, *rain_24h=None*, *snow_1h=None*, *snow_6h=None*, *snow_24h=None*, *dew_point=None*, *humidex=None*, *heat_index=None*, *visibility_distance=None*, *visibility_prefix=None*, *clouds_distance=None*, *clouds_condition=None*, *clouds_cumulus=None*, *weather_precipitation=None*, *weather_descriptor=None*, *weather_intensity=None*, *weather_proximity=None*, *weather_obscuration=None*, *weather_other=None*)

Bases: `object`

`creation_time`(*timeformat='unix'*)

Returns the UTC time of creation of this raw measurement

Parameters **`timeformat`** (`str`) – the format for the time value. May be: '*unix*' (default) for UNIX time, '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` or *date* for a `datetime.datetime` object

Returns an int or a str or a `datetime.datetime` object or None

Raises ValueError

classmethod **`from_dict`**(*the_dict*)

`to_JSON`()

Dumps object fields into a JSON formatted string

Returns the JSON string

`to_dict`()

Dumps object fields into a dictionary

Returns a dict

pyowm.stationsapi30.persistence_backend module

Module containing asbtract classes and a few convenience implementations of raw measurements I/O

class `pyowm.stationsapi30.persistence_backend.`**`JSONPersistenceBackend`**(*json_file_path*, *station_id*)

Bases: `pyowm.stationsapi30.persistence_backend.PersistenceBackend`

A *PersistenceBackend* loading/saving data to a JSON file. Data will be saved as a JSON list, each element being representing data of a *pyowm.stationsapi30.measurement.Measurement* object.

Parameters

- **`json_file_path`** (`str`) – path to the JSON file

- **station_id** (*str*) – unique OWM-provided ID of the station whose data is read/saved

load_to_buffer()
Reads meteostation measurement data into a *pyowm.stationsapi30.buffer.Buffer* object.

Returns a *pyowm.stationsapi30.buffer.Buffer* instance

persist_buffer(*buffer*)
Saves data contained into a *pyowm.stationsapi30.buffer.Buffer* object in a durable form.

Parameters **buffer** (*pyowm.stationsapi30.buffer.Buffer* instance) – the Buffer object to be persisted

class pyowm.stationsapi30.persistence_backend.**PersistenceBackend**
Bases: object

A global abstract class representing an I/O manager for buffer objects containing raw measurements.

load_to_buffer()
Reads meteostation measurement data into a *pyowm.stationsapi30.buffer.Buffer* object.

Returns a *pyowm.stationsapi30.buffer.Buffer* instance

persist_buffer(*buffer*)
Saves data contained into a *pyowm.stationsapi30.buffer.Buffer* object in a durable form.

Parameters **buffer** (*pyowm.stationsapi30.buffer.Buffer* instance) – the Buffer object to be persisted

pyowm.stationsapi30.station module

class pyowm.stationsapi30.station.**Station**(*id*, *created_at*, *updated_at*, *external_id*, *name*, *lon*, *lat*, *alt*, *rank*)
Bases: object

A class representing a meteostation in Stations API. A reference about OWM stations can be found at: http://openweathermap.org/stations

Parameters

- **id** (*str*) – unique OWM identifier for the station
- **created_at** (*str in format %Y-%m-%dT%H:%M:%S.%fZ*) – UTC timestamp marking the station registration.
- **updated_at** (*str in format %Y-%m-%dT%H:%M:%S.%fZ*) – UTC timestamp marking the last update to this station
- **external_id** (*str*) – user-given identifier for the station
- **name** (*str*) – user-given name for the station
- **lon** (*float*) – longitude of the station
- **lat** (*float*) – latitude of the station
- **alt** (*float*) – altitude of the station
- **rank** (*int*) – station rank

creation_time(*timeformat='unix'*)
Returns the UTC time of creation of this station

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time, '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` or *date* for a `datetime.datetime` object

Returns an int or a str or a `datetime.datetime` object or None

Raises ValueError

last_update_time(*timeformat='unix'*)
Returns the UTC time of the last update on this station's metadata

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time, '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` or *date* for a `datetime.datetime` object

Returns an int or a str or a `datetime.datetime` object or None

Raises ValueError

to_JSON()
Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.stationsapi30.stations_manager module

Module contents

pyowm.tiles package

Submodules

pyowm.tiles.enums module

class `pyowm.tiles.enums.`**MapLayerEnum**
Bases: `object`

Allowed map layer values for tiles retrieval

PRECIPITATION = 'precipitation_new'

PRESSURE = 'pressure_new'

TEMPERATURE = 'temp_new'

WIND = 'wind_new'

classmethod items()
All values for this enum :return: list of tuples

pyowm.tiles.tile_manager module

Module contents

pyowm.utils package

Submodules

pyowm.utils.geo module

pyowm.utils.temputils module

Module containing utility functions for temperature and wind units conversion

`pyowm.utils.temputils.`**`kelvin_dict_to`**(*d*, *target_temperature_unit*)
Converts all the values in a dict from Kelvin temperatures to the specified temperature format.

Parameters

- **d** (*dict*) – the dictionary containing Kelvin temperature values
- **target_temperature_unit** (*str*) – the target temperature unit, may be: 'celsius' or 'fahrenheit'

Returns a dict with the same keys as the input dict and converted temperature values as values

Raises *ValueError* when unknown target temperature units are provided

`pyowm.utils.temputils.`**`kelvin_to_celsius`**(*kelvintemp*)
Converts a numeric temperature from Kelvin degrees to Celsius degrees

Parameters **kelvintemp** (*int/long/float*) – the Kelvin temperature

Returns the float Celsius temperature

Raises *TypeError* when bad argument types are provided

`pyowm.utils.temputils.`**`kelvin_to_fahrenheit`**(*kelvintemp*)
Converts a numeric temperature from Kelvin degrees to Fahrenheit degrees

Parameters **kelvintemp** (*int/long/float*) – the Kelvin temperature

Returns the float Fahrenheit temperature

Raises *TypeError* when bad argument types are provided

`pyowm.utils.temputils.`**`metric_wind_dict_to_imperial`**(*d*)
Converts all the wind values in a dict from meters/sec (metric measurement system) to miles/hour (imperial measurement system) .

Parameters **d** (*dict*) – the dictionary containing metric values

Returns a dict with the same keys as the input dict and values converted to miles/hour

pyowm.utils.timeformatutils module

Module containing utility functions for time formats conversion

class `pyowm.utils.timeformatutils.`**`UTC`**

Bases: `datetime.tzinfo`

`dst`(*dt*)
datetime -> DST offset in minutes east of UTC.

`tzname`(*dt*)
datetime -> string name of time zone.

`utcoffset`(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

`pyowm.utils.timeformatutils.`**`timeformat`**(*timeobject*, *timeformat*)
Formats the specified time object to the target format type.

Parameters

- **`timeobject`** (int, `datetime.datetime` or ISO8601-formatted string with pattern `YYYY-MM-DD HH:MM:SS+00`) – the object conveying the time value
- **`timeformat`** (*`str`*) – the target format for the time conversion. May be: '*unix*' (outputs an int UNIXtime), '*date*' (outputs a `datetime.datetime` object) or '*iso*' (outputs an ISO8601-formatted string with pattern `YYYY-MM-DD HH:MM:SS+00`)

Returns the formatted time

Raises ValueError when unknown timeformat switches are provided or when negative time values are provided

`pyowm.utils.timeformatutils.`**`to_ISO8601`**(*timeobject*)
Returns the ISO8601-formatted string corresponding to the time value conveyed by the specified object, which can be either a UNIXtime, a `datetime.datetime` object or an ISO8601-formatted string in the format *YYYY-MM-DD HH:MM:SS+00'*.

Parameters **`timeobject`** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the time value

Returns an ISO8601-formatted string with pattern *YYYY-MM-DD HH:MM:SS+00'*

Raises *TypeError* when bad argument types are provided, *ValueError* when negative UNIXtimes are provided

`pyowm.utils.timeformatutils.`**`to_UNIXtime`**(*timeobject*)
Returns the UNIXtime corresponding to the time value conveyed by the specified object, which can be either a UNIXtime, a `datetime.datetime` object or an ISO8601-formatted string in the format *YYYY-MM-DD HH:MM:SS+00'*.

Parameters **`timeobject`** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the time value

Returns an int UNIXtime

Raises *TypeError* when bad argument types are provided, *ValueError* when negative UNIXtimes are provided

`pyowm.utils.timeformatutils.`**`to_date`**(*timeobject*)
Returns the `datetime.datetime` object corresponding to the time value conveyed by the specified object, which can be either a UNIXtime, a `datetime.datetime` object or an ISO8601-formatted string in the format *YYYY-MM-DD HH:MM:SS+00'*.

Parameters **timeobject** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the time value

Returns a `datetime.datetime` object

Raises *TypeError* when bad argument types are provided, *ValueError* when negative UNIXtimes are provided

pyowm.utils.timeutils module

Module containing utility functions for time values generation/management

`pyowm.utils.timeutils.`**`last_hour`**`(`*date=None*`)`

Gives the `datetime.datetime` object corresponding to the last hour before now or before the specified `datetime.datetime` object.

Parameters **date** (`datetime.datetime` object) – the date you want an hour to be subtracted from (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`last_month`**`(`*date=None*`)`

Gives the `datetime.datetime` object corresponding to the last month before now or before the specified `datetime.datetime` object. A month corresponds to 30 days.

Parameters **date** (`datetime.datetime` object) – the date you want a month to be subtracted from (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`last_three_hours`**`(`*date=None*`)`

Gives the `datetime.datetime` object corresponding to last three hours before now or before the specified `datetime.datetime` object.

Parameters **date** (`datetime.datetime` object) – the date you want three hours to be subtracted from (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`last_week`**`(`*date=None*`)`

Gives the `datetime.datetime` object corresponding to the last week before now or before the specified `datetime.datetime` object. A week corresponds to 7 days.

Parameters **date** (`datetime.datetime` object) – the date you want a week to be subtracted from (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`last_year`**`(`*date=None*`)`

Gives the `datetime.datetime` object corresponding to the last year before now or before the specified `datetime.datetime` object. A year corresponds to 365 days.

Parameters **date** (`datetime.datetime` object) – the date you want a year to be subtracted from (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`millis_offset_between_epochs`**`(`*reference_epoch*, *target_epoch*`)`

Calculates the signed milliseconds delta between the reference unix epoch and the provided target unix epoch :param reference_epoch: the unix epoch that the millis offset has to be calculated with respect to :type reference_epoch: int :param target_epoch: the unix epoch for which the millis offset has to be calculated :type target_epoch: int :return: int `pyowm.utils.timeutils.`**`next_hour`**(*date=None*)

Gives the `datetime.datetime` object corresponding to the next hour from now or from the specified `datetime.datetime` object.

Parameters **`date`** (`datetime.datetime` object) – the date you want an hour to be added (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`next_month`**(*date=None*)

Gives the `datetime.datetime` object corresponding to the next month after now or after the specified `datetime.datetime` object. A month corresponds to 30 days.

Parameters **`date`** (`datetime.datetime` object) – the date you want a month to be added to (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`next_three_hours`**(*date=None*)

Gives the `datetime.datetime` object corresponding to the next three hours from now or from the specified `datetime.datetime` object.

Parameters **`date`** (`datetime.datetime` object) – the date you want three hours to be added (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`next_week`**(*date=None*)

Gives the `datetime.datetime` object corresponding to the next week from now or from the specified `datetime.datetime` object. A week corresponds to 7 days.

Parameters **`date`** (`datetime.datetime` object) – the date you want a week to be added (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`next_year`**(*date=None*)

Gives the `datetime.datetime` object corresponding to the next year after now or after the specified `datetime.datetime` object. A month corresponds to 30 days.

Parameters **`date`** (`datetime.datetime` object) – the date you want a year to be added to (if left `None`, the current date and time will be used)

Returns a `datetime.datetime` object

`pyowm.utils.timeutils.`**`now`**(*timeformat='date'*)

Returns the current time in the specified timeformat.

Parameters **`timeformat`** (*`str`*) – the target format for the time conversion. May be: '*date*' (default - outputs a `datetime.datetime` object), '*unix*' (outputs a long UNIXtime) or '*iso*' (outputs an ISO8601-formatted string with pattern `YYYY-MM-DD HH:MM:SS+00`)

Returns the current time value

Raises ValueError when unknown timeformat switches are provided or when negative time values are provided

`pyowm.utils.timeutils.`**`tomorrow`**(*hour=None*, *minute=None*)

Gives the `datetime.datetime` object corresponding to tomorrow. The default value for optional parameters is the current value of hour and minute. I.e: when called without specifying values for parameters, the resulting object will refer to the time = now + 24 hours; when called with only hour specified, the resulting object will refer to tomorrow at the specified hour and at the current minute.

Parameters

- **hour** (*int*) – the hour for tomorrow, in the format *0-23* (defaults to `None`)
- **minute** (*int*) – the minute for tomorrow, in the format *0-59* (defaults to `None`)

Returns a `datetime.datetime` object

Raises *ValueError* when hour or minute have bad values

`pyowm.utils.timeutils.`**`yesterday`**(*hour=None*, *minute=None*)

Gives the `datetime.datetime` object corresponding to yesterday. The default value for optional parameters is the current value of hour and minute. I.e: when called without specifying values for parameters, the resulting object will refer to the time = now - 24 hours; when called with only hour specified, the resulting object will refer to yesterday at the specified hour and at the current minute.

Parameters

- **hour** (*int*) – the hour for yesterday, in the format *0-23* (defaults to `None`)
- **minute** (*int*) – the minute for yesterday, in the format *0-59* (defaults to `None`)

Returns a `datetime.datetime` object

Raises *ValueError* when hour or minute have bad values

pyowm.utils.weatherutils module

Module containing search and filter utilities for *Weather* objects lists management

`pyowm.utils.weatherutils.`**`any_status_is`**(*weather_list*, *status*, *weather_code_registry*)

Checks if the weather status code of any of the *Weather* objects in the provided list corresponds to the detailed status indicated. The lookup is performed against the provided *WeatherCodeRegistry* object.

Parameters

- **weathers** (*list*) – a list of *Weather* objects
- **status** (*str*) – a string indicating a detailed weather status
- **weather_code_registry** (*WeatherCodeRegistry*) – a *WeatherCodeRegistry* object

Returns `True` if the check is positive, `False` otherwise

`pyowm.utils.weatherutils.`**`filter_by_status`**(*weather_list*, *status*, *weather_code_registry*)

Filters out from the provided list of *Weather* objects a sublist of items having a status corresponding to the provided one. The lookup is performed against the provided *WeatherCodeRegistry* object.

Parameters

- **weathers** (*list*) – a list of *Weather* objects
- **status** (*str*) – a string indicating a detailed weather status
- **weather_code_registry** (*WeatherCodeRegistry*) – a *WeatherCodeRegistry* object

Returns `True` if the check is positive, `False` otherwise

`pyowm.utils.weatherutils.`**`find_closest_weather`**(*weathers_list*, *unixtime*)

Extracts from the provided list of Weather objects the item which is closest in time to the provided UNIXtime.

Parameters

- **weathers_list** (*list*) – a list of *Weather* objects
- **unixtime** (*int*) – a UNIX time

Returns the *Weather* object which is closest in time or `None` if the list is empty `pyowm.utils.weatherutils.`**`is_in_coverage`**(*unixtime*, *weathers_list*)

Checks if the supplied UNIX time is contained into the time range (coverage) defined by the most ancient and most recent *Weather* objects in the supplied list

Parameters

- **`unixtime`** (*`int`*) – the UNIX time to be searched in the time range
- **`weathers_list`** (*`list`*) – the list of *Weather* objects to be scanned for global time coverage

Returns `True` if the UNIX time is contained into the time range, `False` otherwise

`pyowm.utils.weatherutils.`**`status_is`**(*weather*, *status*, *weather_code_registry*)

Checks if the weather status code of a *Weather* object corresponds to the detailed status indicated. The lookup is performed against the provided *WeatherCodeRegistry* object.

Parameters

- **`weather`** (*Weather*) – the *Weather* object whose status code is to be checked
- **`status`** (*`str`*) – a string indicating a detailed weather status
- **`weather_code_registry`** (*WeatherCodeRegistry*) – a *WeatherCodeRegistry* object

Returns `True` if the check is positive, `False` otherwise

pyowm.utils.xmlutils module

Module containing utility functions for generating XML strings

`pyowm.utils.xmlutils.`**`DOM_node_to_XML`**(*tree*, *xml_declaration=True*)

Prints a DOM tree to its Unicode representation.

Parameters

- **`tree`** (an `xml.etree.ElementTree.Element` object) – the input DOM tree
- **`xml_declaration`** (*`bool`*) – if `True` (default) prints a leading XML declaration line

Returns Unicode object

`pyowm.utils.xmlutils.`**`annotate_with_XMLNS`**(*tree*, *prefix*, *URI*)

Annotates the provided DOM tree with XMLNS attributes and adds XMLNS prefixes to the tags of the tree nodes.

Parameters

- **`tree`** (an `xml.etree.ElementTree.ElementTree` or `xml.etree.ElementTree.Element` object) – the input DOM tree
- **`prefix`** (*`str`*) – XMLNS prefix for tree nodes' tags
- **`URI`** (*`str`*) – the URI for the XMLNS definition file

`pyowm.utils.xmlutils.`**`create_DOM_node_from_dict`**(*d*, *name*, *parent_node*)

Dumps dict data to an `xml.etree.ElementTree.SubElement` DOM subtree object and attaches it to the specified DOM parent node. The created subtree object is named after the specified name. If the supplied dict is `None` no DOM node is created for it as well as no DOM subnodes are generated for eventual `None` values found inside the dict

Parameters

- **`d`** (*`dict`*) – the input dictionary

- **name** (*str*) – the name for the DOM subtree to be created
- **parent_node** (`xml.etree.ElementTree.Element` or derivative objects) – the parent DOM node the newly created subtree must be attached to

Returns `xml.etree.ElementTree.SubElementTree` object

Module contents

pyowm.uvindexapi30 package

Subpackages

Submodules

pyowm.uvindexapi30.uvindex module

class `pyowm.uvindexapi30.uvindex.`**`UVIndex`**(*reference_time*, *location*, *value*, *reception_time*)

Bases: `object`

A class representing the UltraViolet Index observed in a certain location in the world. The location is represented by the encapsulated *Location* object.

Parameters

- **reference_time** (*int*) – GMT UNIXtime telling when the UV data have been measured
- **location** (*Location*) – the *Location* relative to this UV observation
- **value** (*float*) – the observed UV intensity value
- **reception_time** (*int*) – GMT UNIXtime telling when the observation has been received from the OWM Weather API

Returns an *UVIndex* instance

Raises *ValueError* when negative values are provided as reception time or UV intensity value

get_exposure_risk()

Returns a string stating the risk of harm from unprotected sun exposure for the average adult on this UV observation :return: str

get_location()

Returns the *Location* object for this UV observation

Returns the *Location* object

get_reception_time(*timeformat='unix'*)

Returns the GMT time telling when the UV has been received from the API

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

get_reference_time(*timeformat='unix'*)

Returns the GMT time telling when the UV has been observed from the OWM Weather API

Parameters **`timeformat`** (*`str`*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

`get_value``()`
Returns the UV intensity for this observation

Returns float

`to_JSON``()`
Dumps object fields into a JSON formatted string

Returns the JSON string

`to_XML``(`*xml_declaration=True*, *xmlns=True*`)`
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **`XML_declaration`** (*`bool`*) – if `True` (default) prints a leading XML declaration line
- **`xmlns`** (*`bool`*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

`pyowm.uvindexapi30.uvindex.`**`uv_intensity_to_exposure_risk`**`(`*uv_intensity*`)`

pyowm.uvindexapi30.parsers module

pyowm.uvindexapi30.uv_client module

pyowm.uvindexapi30.uris module

URIs templates for resources exposed by the UVIndex API 3.0

Module contents

pyowm.weatherapi25 package

Subpackages

pyowm.weatherapi25.cityids package

Submodules

Module contents

pyowm.weatherapi25.parsers package

Submodules

pyowm.weatherapi25.parsers.forecastparser module

pyowm.weatherapi25.parsers.observationlistparser module

pyowm.weatherapi25.parsers.observationparser module

pyowm.weatherapi25.parsers.stationhistoryparser module

Module containing a concrete implementation for JSONParser abstract class, returning a StationHistory instance

class `pyowm.weatherapi25.parsers.stationhistoryparser.`**`StationHistoryParser`**

Bases: *`pyowm.abstractions.jsonparser.JSONParser`*

Concrete *JSONParser* implementation building a *StationHistory* instance out of raw JSON data coming from OWM Weather API responses.

`parse_JSON`(*JSON_string*)

Parses a *StationHistory* instance out of raw JSON data. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **`JSON_string`** (*`str`*) – a raw JSON string

Returns a *StationHistory* instance or `None` if no data is available

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the JSON string embeds an HTTP status error

pyowm.weatherapi25.parsers.stationlistparser module

Module containing a concrete implementation for JSONParser abstract class, returning a list of Station instances

class `pyowm.weatherapi25.parsers.stationlistparser.`**`StationListParser`**

Bases: *`pyowm.abstractions.jsonparser.JSONParser`*

Concrete *JSONParser* implementation building a list of *Station* instances out of raw JSON data coming from OWM Weather API responses.

parse_JSON(*JSON_string*)

Parses a list of *Station* instances out of raw JSON data. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **JSON_string** (`str`) – a raw JSON string

Returns a list of *Station* instances or `None` if no data is available

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the OWM API returns a HTTP status error

pyowm.weatherapi25.parsers.stationparser module

Module containing a concrete implementation for JSONParser abstract class, returning a Station instance

class `pyowm.weatherapi25.parsers.stationparser.`**StationParser**

Bases: *pyowm.abstractions.jsonparser.JSONParser*

Concrete *JSONParser* implementation building a *Station* instance out of raw JSON data coming from OWM Weather API responses.

parse_JSON(*JSON_string*)

Parses a *Station* instance out of raw JSON data. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **JSON_string** (`str`) – a raw JSON string

Returns a *Station* instance or `None` if no data is available

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the JSON string embeds an HTTP status error

pyowm.weatherapi25.parsers.weatherhistoryparser module

Module containing a concrete implementation for JSONParser abstract class, returning a list of Weather objects

class `pyowm.weatherapi25.parsers.weatherhistoryparser.`**WeatherHistoryParser**

Bases: *pyowm.abstractions.jsonparser.JSONParser*

Concrete *JSONParser* implementation building a list of *Weather* instances out of raw JSON data coming from OWM Weather API responses.

parse_JSON(*JSON_string*)

Parses a list of *Weather* instances out of raw JSON data. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **JSON_string** (`str`) – a raw JSON string

Returns a list of *Weather* instances or `None` if no data is available

Raises *ParseResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the JSON string embeds an HTTP status error

Module contents

pyowm.weatherapi25.xsd package

Submodules

pyowm.weatherapi25.xsd.xmlnsconfig module

XMLNS configuration

Module contents

Submodules

pyowm.weatherapi25.cityidregistry module

pyowm.weatherapi25.configuration25 module

pyowm.weatherapi25.forecast module

Module containing weather forecast classes and data structures.

class `pyowm.weatherapi25.forecast.`**`Forecast`**(*interval*, *reception_time*, *location*, *weathers*)

Bases: `object`

A class encapsulating weather forecast data for a certain location and relative to a specific time interval (forecast for every three hours or for every day)

Parameters

- **`interval`** (*`str`*) – the time granularity of the forecast. May be: *'3h'* for three hours forecast or *'daily'* for daily ones
- **`reception_time`** (*`int`*) – GMT UNIXtime of the forecast reception from the OWM web API
- **`location`** (*`Location`*) – the *Location* object relative to the forecast
- **`weathers`** (*`list`*) – the list of *Weather* objects composing the forecast

Returns a *Forecast* instance

Raises *ValueError* when negative values are provided

`actualize`()

Removes from this forecast all the *Weather* objects having a reference timestamp in the past with respect to the current timestamp

`count_weathers`()

Tells how many *Weather* items compose the forecast

Returns the *Weather* objects total

`get`(*index*)

Lookups up into the *Weather* items list for the item at the specified index

Parameters **`index`** (*`int`*) – the index of the *Weather* object in the list

Returns a *Weather* object

get_interval()
Returns the time granularity of the forecast

Returns str

get_location()
Returns the Location object relative to the forecast

Returns a *Location* object

get_reception_time(*timeformat='unix'*)

Returns the GMT time telling when the forecast was received from the OWM Weather API

Parameters **timeformat** (*str*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError

get_weathers()
Returns a copy of the *Weather* objects list composing the forecast

Returns a list of *Weather* objects

set_interval(*interval*)
Sets the time granularity of the forecast

Parameters **interval** (*str*) – the time granularity of the forecast, may be "3h" or "daily"

to_JSON()
Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

class `pyowm.weatherapi25.forecast.`**ForecastIterator**(*obj*)
Bases: `object`

Iterator over the list of *Weather* objects encapsulated in a *Forecast* class instance

Parameters **obj** (*object*) – the iterable object

Returns a *ForecastIterator* instance

next()
Compatibility for Python 2.x, delegates to function: *__next__()* Returns the next *Weather* item

Returns the next *Weather* item

pyowm.weatherapi25.forecaster module

pyowm.weatherapi25.historian module

Module containing weather history abstraction classes and data structures.

class pyowm.weatherapi25.historian.**Historian**(*station_history*)

Bases: object

A class providing convenience methods for manipulating meteostation weather history data. The class encapsulates a *StationHistory* instance and provides abstractions on the top of it in order to let programmers exploit meteostation weather history data in a human-friendly fashion

Parameters **station_history** (*StationHistory*) – a *StationHistory* instance

Returns a *Historian* instance

average_humidity()

Returns the average value in the humidity series

Returns a float

Raises ValueError when the measurement series is empty

average_pressure()

Returns the average value in the pressure series

Returns a float

Raises ValueError when the measurement series is empty

average_rain()

Returns the average value in the rain series

Returns a float

Raises ValueError when the measurement series is empty

average_temperature(*unit='kelvin'*)

Returns the average value in the temperature series

Parameters **unit** (*str*) – the unit of measure for the temperature values. May be among: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a float

Raises ValueError when invalid values are provided for the unit of measure or the measurement series is empty

get_station_history()

Returns the *StationHistory* instance

Returns the *StationHistory* instance

humidity_series()

Returns the humidity time series relative to the meteostation, in the form of a list of tuples, each one containing the couple timestamp-value

Returns a list of tuples

max_humidity()

Returns a tuple containing the max value in the humidity series preceeded by its timestamp

Returns a tuple

Raises ValueError when the measurement series is empty

`max_pressure`()
Returns a tuple containing the max value in the pressure series preceeded by its timestamp

Returns a tuple

Raises ValueError when the measurement series is empty

`max_rain`()
Returns a tuple containing the max value in the rain series preceeded by its timestamp

Returns a tuple

Raises ValueError when the measurement series is empty

`max_temperature`(*unit='kelvin'*)
Returns a tuple containing the max value in the temperature series preceeded by its timestamp

Parameters **`unit`** (`str`) – the unit of measure for the temperature values. May be among: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a tuple

Raises ValueError when invalid values are provided for the unit of measure or the measurement series is empty

`min_humidity`()
Returns a tuple containing the min value in the humidity series preceeded by its timestamp

Returns a tuple

Raises ValueError when the measurement series is empty

`min_pressure`()
Returns a tuple containing the min value in the pressure series preceeded by its timestamp

Returns a tuple

Raises ValueError when the measurement series is empty

`min_rain`()
Returns a tuple containing the min value in the rain series preceeded by its timestamp

Returns a tuple

Raises ValueError when the measurement series is empty

`min_temperature`(*unit='kelvin'*)
Returns a tuple containing the min value in the temperature series preceeded by its timestamp

Parameters **`unit`** (`str`) – the unit of measure for the temperature values. May be among: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a tuple

Raises ValueError when invalid values are provided for the unit of measure or the measurement series is empty

`pressure_series`()
Returns the atmospheric pressure time series relative to the meteostation, in the form of a list of tuples, each one containing the couple timestamp-value

Returns a list of tuples

rain_series()

Returns the precipitation time series relative to the meteostation, in the form of a list of tuples, each one containing the couple timestamp-value

Returns a list of tuples

temperature_series(*unit='kelvin'*)

Returns the temperature time series relative to the meteostation, in the form of a list of tuples, each one containing the couple timestamp-value

Parameters **unit** (`str`) – the unit of measure for the temperature values. May be among: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a list of tuples

Raises ValueError when invalid values are provided for the unit of measure

wind_series()

Returns the wind speed time series relative to the meteostation, in the form of a list of tuples, each one containing the couple timestamp-value

Returns a list of tuples

pyowm.weatherapi25.location module

pyowm.weatherapi25.observation module

Weather observation classes and data structures.

class pyowm.weatherapi25.observation.**Observation**(*reception_time*, *location*, *weather*)

Bases: `object`

A class representing the weather which is currently being observed in a certain location in the world. The location is represented by the encapsulated *Location* object while the observed weather data are held by the encapsulated *Weather* object.

Parameters

- **reception_time** (`int`) – GMT UNIXtime telling when the weather obervation has been received from the OWM Weather API
- **location** (*Location*) – the *Location* relative to this observation
- **weather** (*Weather*) – the *Weather* relative to this observation

Returns an *Observation* instance

Raises *ValueError* when negative values are provided as reception time

get_location()

Returns the *Location* object for this observation

Returns the *Location* object

get_reception_time(*timeformat='unix'*)

Returns the GMT time telling when the observation has been received from the OWM Weather API

Parameters **timeformat** (`str`) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

get_weather()
Returns the *Weather* object for this observation

Returns the *Weather* object

to_JSON()
Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.weatherapi25.owm25 module

pyowm.weatherapi25.station module

Module containing classes and data structures related to meteostation data

class `pyowm.weatherapi25.station.`**Station**(*name*, *station_ID*, *station_type*, *status*, *lat*, *lon*, *distance=None*, *last_weather=None*)
Bases: `object`

A class representing meteostations which are reporting current weather conditions from geographical coordinates.

Parameters

- **name** (*string*) – meteostation name
- **station_ID** (*int*) – OWM station ID
- **station_type** (*int*) – meteostation type
- **status** (*int*) – station status
- **lat** (*float*) – latitude for station
- **lon** (*float*) – longitude for station
- **distance** (*float*) – distance of station from lat/lon of search criteria
- **last_weather** (*Weather* instance) – last reported weather conditions from station

Returns a *Station* instance

Raises *ValueError* if *lon* or *lat* values are provided out of bounds or *last_weather* is not an instance of *Weather* or *None*

get_distance()
Returns the distance of the station from the geo coordinates used in search

Returns the float distance from geo coordinates

get_last_weather()
Returns the last reported weather conditions reported by the station.

Returns the last *Weather* instance reported by station

get_lat()
Returns the latitude of the location

Returns the float latitude

get_lon()
Returns the longitude of the location

Returns the float longitude

get_name()
Returns the name of the station

Returns the Unicode station name

get_station_ID()
Returns the OWM station ID

Returns the int OWM station ID

get_station_type()
Returns the OWM station type

Returns the int OWM station type

get_status()
Returns the OWM station status

Returns the int OWM station status

to_JSON()
Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.weatherapi25.stationhistory module

Module containing classes and datastructures related to meteostation history data

class pyowm.weatherapi25.stationhistory.**StationHistory**(*station_ID*, *interval*, *reception_time*, *measurements*)
Bases: `object`

A class representing historic weather measurements collected by a meteostation. Three types of historic meteostation data can be obtained by the OWM Weather API: ticks (one data chunk per minute) data, hourly and daily data.

Parameters

- **`station_ID`** (*`int`*) – the numeric ID of the meteostation
- **`interval`** (*`str`*) – the time granularity of the meteostation data history
- **`reception_time`** (*`int`*) – GMT UNIXtime of the data reception from the OWM web API
- **`measurements`** (*`dict`*) – a dictionary containing raw weather measurements

Returns a *StationHistory* instance

Raises *ValueError* when the supplied value for reception time is negative

`get_interval``()`

Returns the interval of the meteostation history data

Returns the int interval

`get_measurements``()`

Returns the measurements of the meteostation as a dict. The dictionary keys are UNIX timestamps and for each one the value is a dict containing the keys 'temperature','humidity','pressure','rain','wind' along with their corresponding numeric values. Eg: `{1362933983: { "temperature": 266.25, "humidity": 27.3, "pressure": 1010.02, "rain": None, "wind": 4.7}, ... }`

Returns the dict containing the meteostation's measurements

`get_reception_time``(`*timeformat='unix'*`)`

Returns the GMT time telling when the meteostation history data was received from the OWM Weather API

Parameters **`timeformat`** (*`str`*) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError

`get_station_ID``()`

Returns the ID of the meteostation

Returns the int station ID

`set_interval``(`*interval*`)`

Sets the interval of the meteostation history data

Parameters **`interval`** (*`string`*) – the time granularity of the meteostation history data, may be among "tick","hour" and "day"

`set_station_ID``(`*station_ID*`)`

Sets the numeric ID of the meteostation

Parameters **`station_ID`** (*`int`*) – the numeric ID of the meteostation

`to_JSON``()`

Dumps object fields into a JSON formatted string

Returns the JSON string

to_XML(*xml_declaration=True*, *xmlns=True*)

Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **XML_declaration** (*bool*) – if `True` (default) prints a leading XML declaration line
- **xmlns** (*bool*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

pyowm.weatherapi25.weather module

Module containing weather data classes and data structures.

class `pyowm.weatherapi25.weather.`**Weather**(*reference_time*, *sunset_time*, *sunrise_time*, *clouds*, *rain*, *snow*, *wind*, *humidity*, *pressure*, *temperature*, *status*, *detailed_status*, *weather_code*, *weather_icon_name*, *visibility_distance*, *dewpoint*, *humidex*, *heat_index*)

Bases: `object`

A class encapsulating raw weather data. A reference about OWM weather codes and icons can be found at: http://bugs.openweathermap.org/projects/api/wiki/Weather_Condition_Codes

Parameters

- **reference_time** (*int*) – GMT UNIX time of weather measurement
- **sunset_time** (*int or None*) – GMT UNIX time of sunset or None on polar days
- **sunrise_time** (*int or None*) – GMT UNIX time of sunrise or None on polar nights
- **clouds** (*int*) – cloud coverage percentage
- **rain** (*dict*) – precipitation info
- **snow** (*dict*) – snow info
- **wind** (*dict*) – wind info
- **humidity** (*int*) – atmospheric humidity percentage
- **pressure** (*dict*) – atmospheric pressure info
- **temperature** (*dict*) – temperature info
- **status** (*Unicode*) – short weather status
- **detailed_status** (*Unicode*) – detailed weather status
- **weather_code** (*int*) – OWM weather condition code
- **weather_icon_name** (*Unicode*) – weather-related icon name
- **visibility_distance** (*float*) – visibility distance
- **dewpoint** (*float*) – dewpoint
- **humidex** (*float*) – Canadian humidex
- **heat_index** (*float*) – heat index

Returns a *Weather* instance

Raises *ValueError* when negative values are provided

`get_clouds`()
Returns the cloud coverage percentage as an int

Returns the cloud coverage percentage

`get_detailed_status`()
Returns the detailed weather status as a Unicode string

Returns the detailed weather status

`get_dewpoint`()
Returns the dew point as a float

Returns the dew point

`get_heat_index`()
Returns the heat index as a float

Returns the heat index

`get_humidex`()
Returns the Canadian humidex as a float

Returns the Canadian humidex

`get_humidity`()
Returns the atmospheric humidity as an int

Returns the humidity

`get_pressure`()
Returns a dict containing atmospheric pressure info

Returns a dict containing pressure info

`get_rain`()
Returns a dict containing precipitation info

Returns a dict containing rain info

`get_reference_time`(*timeformat='unix'*)
Returns the GMT time telling when the weather was measured

Parameters **`timeformat`** (`str`) – the format for the time value. May be: '*unix*' (default) for UNIX time '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` '*date* for `datetime.datetime` object instance

Returns an int or a str

Raises ValueError when negative values are provided

`get_snow`()
Returns a dict containing snow info

Returns a dict containing snow info

`get_status`()
Returns the short weather status as a Unicode string

Returns the short weather status

`get_sunrise_time`(*timeformat='unix'*)
Returns the GMT time of sunrise

Parameters **`timeformat`** (`str`) – the format for the time value. May be: '*unix*' (default) for UNIX time or '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns an int or a str or None

Raises ValueError

`get_sunset_time`(*timeformat='unix'*)
Returns the GMT time of sunset

Parameters **`timeformat`** (`str`) – the format for the time value. May be: '*unix*' (default) for UNIX time or '*iso*' for ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns an int or a str or None

Raises ValueError

`get_temperature`(*unit='kelvin'*)
Returns a dict with temperature info

Parameters **`unit`** (`str`) – the unit of measure for the temperature values. May be: '*kelvin*' (default), '*celsius*' or '*fahrenheit*'

Returns a dict containing temperature values.

Raises ValueError when unknown temperature units are provided

`get_visibility_distance`()
Returns the visibility distance as a float

Returns the visibility distance

`get_weather_code`()
Returns the OWM weather condition code as an int

Returns the OWM weather condition code

`get_weather_icon_name`()
Returns weather-related icon name as a Unicode string.

Returns the icon name.

`get_weather_icon_url`()
Returns weather-related icon URL as a Unicode string.

Returns the icon URL.

`get_wind`(*unit='meters_sec'*)
Returns a dict containing wind info

Parameters **`unit`** (`str`) – the unit of measure for the wind values. May be: '*meters_sec*' (default) or '*miles_hour*'

Returns a dict containing wind info

`to_JSON`()
Dumps object fields into a JSON formatted string

Returns the JSON string

`to_XML`(*xml_declaration=True*, *xmlns=True*)
Dumps object fields to an XML-formatted string. The 'xml_declaration' switch enables printing of a leading standard XML line containing XML version and encoding. The 'xmlns' switch enables printing of qualified XMLNS prefixes.

Parameters

- **`XML_declaration`** (*`bool`*) – if `True` (default) prints a leading XML declaration line
- **`xmlns`** (*`bool`*) – if `True` (default) prints full XMLNS prefixes

Returns an XML-formatted string

`pyowm.weatherapi25.weather.`**`weather_from_dictionary`**`(`*d*`)`

Builds a *Weather* object out of a data dictionary. Only certain properties of the dictionary are used: if these properties are not found or cannot be read, an error is issued.

Parameters **`d`** (*`dict`*) – a data dictionary

Returns a *Weather* instance

Raises *KeyError* if it is impossible to find or read the data needed to build the instance

pyowm.weatherapi25.weathercoderegistry module

Module containing weather code lookup and resolution classes

`class` `pyowm.weatherapi25.weathercoderegistry.`**`WeatherCodeRegistry`**`(`*code_ranges_dict*`)`

Bases: `object`

A registry class for looking up weather statuses from weather codes.

Parameters **`code_ranges_dict`** (*`dict`*) – a dict containing the mapping between weather statuses (eg: "sun","clouds",etc) and weather code ranges

Returns a *WeatherCodeRegistry* instance

`status_for``(`*code*`)`

Returns the weather status related to the specified weather status code, if any is stored, `None` otherwise.

Parameters **`code`** (*`int`*) – the weather status code whose status is to be looked up

Returns the weather status str or `None` if the code is not mapped

Module contents

7.1.2 Submodules

7.1.3 pyowm.constants module

Constants for the PyOWM library

7.1.4 Module contents

The PyOWM init file

Author: Claudio Sparpaglione, @csparpa <csparpa@gmail.com>

Platform: platform independent

`pyowm.`**`OWM`**`(`*API_key='b1b15e88fa797225412429c1c50c122a'*, *version='2.5'*, *config_module=None*, *language=None*, *subscription_type=None*, *use_ssl=None*`)`

A parametrized factory method returning a global OWM instance that represents the desired OWM Weather API version (or the currently supported one if no version number is specified)

Parameters

- **`API_key`** (*`str`*) – the OWM Weather API key (defaults to a test value)
- **`version`** (*`str`*) – the OWM Weather API version. Defaults to `None`, which means use the latest web API version
- **`config_module`** (*`str (eg: 'mypackage.mysubpackage.myconfigmodule')`*) – the Python path of the configuration module you want to provide for instantiating the library. Defaults to `None`, which means use the default configuration values for the web API version support you are currently requesting. Please be aware that malformed user-defined configuration modules can lead to unwanted behaviour!
- **`language`** (*`str`*) – the language in which you want text results to be returned. It's a two-characters string, eg: "en", "ru", "it". Defaults to: `None`, which means use the default language.
- **`subscription_type`** (*`str`*) – the type of OWM Weather API subscription to be wrapped. Can be 'free' (free subscription) or 'pro' (paid subscription), Defaults to: 'free'
- **`use_ssl`** (*`bool`*) – whether API calls should be made via SSL or not. Defaults to: False

Returns an instance of a proper *OWM* subclass

Raises *ValueError* when unsupported OWM API versions are provided

CHAPTER 8

How to contribute

There are multiple ways to contribute to the PyOWM project! Find the one that suits you best

8.1 Contributing

Contributing is easy anwd welcome"

You can contribute to PyOWM in a lot of ways:

- reporting a reproducible defect (eg. bug, installation crash, . . .)
- make a wish for a reasonable new feature
- increase the test coverage
- refactor the code
- improve PyOWM reach on platforms (eg. bundle it for Linux distros, managers, oding, testing, packaging, reporting issues) are welcome!_.

And last but not least. . . use it! Use PyOWM in your own projects, as lots of people already do

In order to get started, follow these simple steps

1. First, meet the community and wave hello! You can join the **PyOWM public Slack team** by signing up here
2. Depending on how you want to contribute, take a look at one of the following sections
3. Don't forget tell @csparpa or the community to add yourself to the `CONTRIBUTORS.md` file - or do it yourself if you're contributing on code

8.2 Reporting a PyOWM bug

That's simple: what you need to do is just open a new issue on GitHub.

8.2.1 Bug reports - general principles

In order to allow the community to understand what the bug is, *you should provide as much information as possible* on it. Vague or succint bug reports are not useful and will very likely result in follow ups needed.

Only bugs related to PyOWM will be addressed: it might be that you're using PyOWM in a broader context (eg. a webapplication) so bugs affecting the broader context are out of scope - unless they are caused in chain to PyOWM issues.

Also, please do understand that we can only act on *reproducible bugs*: this means that a bug does not exist if it is not possible to reproduce it by two different persons. So please provide facts, not "smells" of a potential bug"

8.2.2 What a good bug report should contain

These info are part of a good bug report

- brief description of the issue
- *how to reproduce the issue*
- what is the impacted PyOWM issue
- what Python version are you running PyOWM with
- what is your operating system
- stacktrace of the error/crash if available
- if you did a bit of research yourself and/or have a fix in mind, just suggest please :)
- (optional) transcripts from the shell/logfiles or screenshots

8.3 Requesting for a new PyOWM feature

That's simple as well!

1. Open an issue on GitHub (describe with as much detail as possible the feature you're proposing - and also
2. Depending on the entity of the request:
    - if it's going to be a breaking change, the feature will be scheduled for embedding into the next major release - so no code shall be provided by then
    - if it's only an enhancement, you might proceed with submitting the code yourself!

8.4 Contributing on code

This applies to all kind of works on code (fixing bugs, developing new features, refactoring code, adding tests…)

A few simple steps:

1. Fork the PyOWM repository on GitHub
2. Install the development dependencies on your local development setup
3. On your fork, work on the **development branch** (*not the master branch!!!*) or on a **ad-hoc feature branch**. Don't forget to insert your name in the `CONTRIBUTORS.md` file!
4. TEST YOUR CODE please!

5. DOCUMENT YOUR CODE - especially if new features/complex patches are introduced
6. Submit a pull request

8.4.1 Installing development dependencies

In order to develop code and run tests for PyOWM, you must have installed the dev dependencies. From the project root folder, just run:

```
pip install -r dev-requirements.txt
```

It is adviced that you do it on a virtualenv or, if you prefer to spin up the whole dev environment with just one command, you can run the PyOWM Docker image:

```
docker run -d --name pyowm csparpa/pyowm
```

8.4.2 Guidelines for code branching

The project adopts @nvie's branching model:

- the "develop" branch contains work-in-progress code
- new "feature"/"bugfix" branches can be opened by any contributor from the "develop" branch. Each feature branch adds a new feature/enhancement or bugfix
- the "master" branch will contain only stable code and the "develop" branch will be merged back into it only when a milestone is completed or hotfixes need to be applied. Merging of "develop" into "master" will be done by @csparpa when releasing - so please **never apply your modifications to the master branch!!!**

8.4.3 Guidelines for code testing

Main principles:

- Each software functionality should have a related unit test
- Each bug should have at least one regression test associated

Here is how to run tests

8.5 Contributing on PyOWM bundling/distributing

Please open a GitHub issue and get in touch to discuss your idea!

CHAPTER 9

PyOWM Community

Find us on Slack !

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index