pyowm Documentation

Claudio Sparpaglione

Jun 22, 2020

Contents

Welcome to PyOWM v3 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 + OneCall API, 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

Used to work with PyOWM v2?

PyOWM v3 is a brand new branch of the library and therefore differs from PyOWM v2 branch. This means that **v3 offers no retrocompatibility with v2: this might result in your code breaking** if it uses PyOWM v2 and you uncarefully update! Moreover, PyOWM v3 runs on Python 3 only.

PyOWM v2 will follow this Timeline

It is highly recommended that you upgrade your PyOWM v2 dependency to PyOWM v3: follow this guide for Migrating

CHAPTER 4

Supported environments and Python versions

PyOWM runs on Windows, Linux and MacOS. PyOWM runs on Python 3.7+

CHAPTER 5

Usage and Technical Documentation

5.1 PyOWM v3 documentation

5.1.1 Quick code recipes (work in progress)

Code recipes

This section provides code snippets you can use to quickly get started with PyOWM when performing common enquiries related to weather data.

Table of contents:

- *Library initialization*
- *Identifying cities and places via city IDs*
- *Weather data*
- *Weather forecasts*
- *Meteostation historic measurements*
- *OneCall data*

Library initialization

Initialize PyOWM with default configuration and a free API key

```
from pyowm.owm import OWM
owm = OWM('your-free-api-key')
```

Initialize PyOWM with configuration loaded from an external JSON file

You can setup a configuration file and then have PyOWM read it. The file must contain a valid JSON document with the following format:

```
{
    "subscription_type": free|startup|developer|professional|enterprise
    "language": en|ru|ar|zh_cn|ja|es|it|fr|de|pt|... (check https://openweathermap.
↪org/current)
    "connection": {
        "use_ssl": true|false,
        "verify_ssl_certs": true|false,
        "use_proxy": true|false,
        "timeout_secs": N
    },
    "proxies": {
        "http": HTTP_URL,
        "https": SOCKS5_URL
    }
}
```

```
from pyowm.owm import OWM
from pyowm.utils.config import get_config_from
config_dict = get_config_from('/path/to/configfile.json')
owm = OWM('your-free-api-key', config_dict)
```

Initialize PyOWM with a paid subscription - eg: professional

If you bought a paid subscription then you need to provide PyOWM both your paid API key and the subscription type that you've bought

```
from pyowm.owm import OWM
from pyowm.utils.config import get_default_config_for_subscription_type
config_dict = get_default_config_for_subscription_type('professional')
owm = OWM('your-paid-api-key', config_dict)
```

Use PyOWM behind a proxy server

If you have an HTTP or SOCKS5 proxy server you need to provide PyOWM two URLs; one for each HTTP and HTTPS protocols. URLs are in the form: 'protocol://username:password@proxy_hostname:proxy_port'

```
from pyowm.owm import OWM
from pyowm.utils.config import get_default_config_for_proxy
config_dict = get_default_config_for_proxy(
    'http://user:pass@192.168.1.77:8464',
    'https://user:pass@192.168.1.77:8934'
)
owm = OWM('your-api-key', config_dict)
```

Language setting

English is the default - but you can change it Check out https://openweathermap.org/current for the complete list of supported languages

```
from pyowm.owm import OWM
from pyowm.utils.config import get_default_config
config_dict = get_default_config()
config_dict['language'] = 'pt'  # your language here
owm = OWM('your-api-key', config_dict)
```

Get PyOWM configuration

Configuration can be changed: just get it, it's a plain Python dict

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
config_dict = owm.configuration
```

Get the version of PyOWM library

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
version_tuple = (major, minor, patch) = owm.version
```

Identifying cities and places via city IDs

Obtain the city ID registry

Use the city ID registry to lookup the ID of a city given its name

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
city_id_registry = owm.city_id_registry()
```

Get the ID of a city given its name

Don't forget that there is a high probabilty that your city is not unique in the world, and multiple cities with the same name exist in other countries Therefore specify toponyms and country 2-letter names separated by comma. Eg: if you search for the British `London` you'll likely multiple results: you then should also specify the country (`GB`) to narrow the search only to Great Britain.

Let's search for it:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
reg = owm.city_id_registry()
list_of_tuples = london = reg.ids_for('London')                        # lots␣
↪of results
```

(continues on next page)

(continued from previous page)

```
list_of_tuples = london = reg.ids_for('London', country='GB')                 # only␣
↪one: [ (2643743,, 'London, GB') ]
id_of_london_city = list_of_tuples[0][0]
```

The search here was by default case insensitive: you could have tried

```
list_of_tuples = london = reg.ids_for('london', country='GB')                 #␣
↪notice the lowercase
list_of_tuples = london = reg.ids_for('LoNdoN', country='GB')                 #␣
↪notice the camelcase
```

and would get the very same results as above.

Get the IDs of cities whose name cointain a specific string

In order yo find all cities with names having your string as a substring you need to use the optional parameter `matching='like'`

In example, let's find IDs for all British cities having the string `london` in their names:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
reg = owm.city_id_registry()
list_of_tuples = reg.ids_for('london', country='GB', matching='like')  # We'll get␣
↪[(2643741, 'City of London', 'GB'),
                                                                       #           ␣
↪(2648110, 'Greater London', 'GB'),
                                                                       #           ␣
↪(7535661, 'London Borough of Harrow', 'GB'),
                                                                       #           ␣
↪(2643743, 'London', 'GB'),
                                                                       #           ␣
↪(2643734, 'Londonderry County Borough', 'GB')]
```

Get geographic coordinates of a city given its name

Just use call `locations_for` on the registry: this will give you a `Location` object containing lat & lon

Let's find geocoords for Moscow (Russia):

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
reg = owm.city_id_registry()
list_of_locations = reg.locations_for('moscow', country='RU')
moscow = list_of_locations[0][0]
lat = moscow.lat   # 55.75222
lon = moscow.lon   # 37.615555
```

Get GeoJSON geometry (point) for a city given its name

PyOWM encapsulates GeoJSON geometry objects that are compliant with the GeoJSON specification.

This means, for example, that you can get a `Point` geometry using the registry. Let's find the geometries for all `Rome` cities in the world:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
reg = owm.city_id_registry()
list_of_geopoints = reg.geopoints_for('rome')
```

Observed weather

Obtain a Weather API manager object

The manager object is used to query weather data, including observations, forecasts, etc

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
weather_mgr = owm.weather_manager()
```

Get current weather status on a location

Queries work best by specifying toponyms and country 2-letter names separated by comma. Eg: instead of using `seattle` try using `seattle,WA`

Say now we want the currently observed weather in London (Great Britain):

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
observation = mgr.weather_at_place('London,GB')  # the observation object is a box␣
↪containing a weather object
weather = observation.weather
weather.status           # short version of status (eg. 'Rain')
weather.detailed_status  # detailed version of status (eg. 'light rain')
```

The weather object holds all weather-related info

Get current and today's min-max temperatures in a location

Temperature can be retrieved in Kelvin, Celsius and Fahrenheit units

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
weather = mgr.weather_at_place('Tokyo,JP').weather
temp_dict_kelvin = weather.temperature()   # a dict in Kelvin units (default when no␣
↪temperature units provided)
temp_dict_kelvin['temp_min']
temp_dict_kelvin['temp_max']
temp_dict_fahrenheit = weather.temperature('fahrenheit')  # a dict in Fahrenheit units
temp_dict_celsius = weather.temperature('celsius')  # guess?
```

Get current wind info on a location

Wind is a dict,with the following information: wind speed, degree (meteorological) and gusts. Available measurement units for speed and gusts are: meters/sec (default), miles/hour, knots and Beaufort scale.

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
observation = mgr.weather_at_place('Tokyo,JP')
wind_dict_in_meters_per_sec = observation.weather.wind()   # Default unit: 'meters_sec
↪'
wind_dict_in_meters_per_sec['speed']
wind_dict_in_meters_per_sec['deg']
wind_dict_in_meters_per_sec['gust']
wind_dict_in_miles_per_h = mgr.weather_at_place('Tokyo,JP').wind(unit='miles_hour')
wind_dict_in_knots = mgr.weather_at_place('Tokyo,JP').wind(unit='knots')
wind_dict_in_beaufort = mgr.weather_at_place('Tokyo,JP').wind(unit='beaufort')  #␣
↪Beaufort is 0-12 scale
```

Get current rain amount on a location

Also rain amount is a dict, with keys: `1h` an `3h`, containing the mms of rain fallen in the last 1 and 3 hours

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
rain_dict = mgr.weather_at_place('Berlin,DE').observation.rain
rain_dict['1h']
rain_dict['3h']
```

Get current pressure on a location

Pressure is similar to rain: you get a dict with keys: `press` (atmospheric pressure on the ground in hPa) and `sea_level` (on the sea level, if location is on the sea)

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
pressure_dict = mgr.weather_at_place('Berlin,DE').observation.pressure
pressure_dict['press']
pressure_dict['sea_level']
```

Get today's sunrise and sunset times for a location

You can get precise timestamps for sunrise and sunset times on a location. Sunrise can be `None` for locations in polar night, as well as sunset can be `None` in case of polar days Supported time units are: `unix` (default, UNIX time), `iso` (format `YYYY-MM-DD HH:MM:SS+00`) or `datetime` (gives a plain Python `datetime.datetime` object)

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
observation = mgr.weather_at_place('Berlin,DE')
```

(continues on next page)

(continued from previous page)

```
weather = observation.weather
sunrise_unix = weather.sunrise_time()  # default unit: 'unix'
sunrise_iso = weather.sunrise_time(timeformat='iso')
sunrise_date = weather.sunrise_time(timeformat='date')
sunrset_unix = weather.sunset_time()  # default unit: 'unix'
sunrset_iso = weather.sunset_time(timeformat='iso')
sunrset_date = weather.sunset_time(timeformat='date')
```

Get weather on geographic coordinates

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
my_city_id = 12345
moscow_lat = 55.75222
moscow_lon = 37.615555
weather_at_moscow = owm.weather_at_coords(moscow_lat, moscow_lon).weather
```

Get weather at city IDs

You can enquire the observed weather on a city ID:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
my_city_id = 12345
weather = owm.weather_at_id(my_city_id).weather
```

or on a list of city IDs:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
my_list_of_city_ids = [12345, 67890, 54321]
list_of_observations = owm.weather_at_ids(my_list_of_city_ids)
corresponding_weathers_list = [ obs.weather for obs in list_of_observations ]
```

Current weather search based on string similarity

In one shot, you can query for currently observed weather:

- for all the places whose name equals the string you provide (use `'accurate'`)
- for all the places whose name contains the string you provide (use `'like'`)

You can control how many items the returned list will contain by using the `limit` parameter

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
obs_list = mgr.weather_at_places('London', 'accurate')        # Find observed weather␣
↪in all the "London"s in the world
```

(continues on next page)

(continued from previous page)

```
obs_list = mgr.weather_at_places('London', 'like', limit=5)    # Find observed weather␣
↪for all the places whose name contains
                                                               # the word "London".␣
↪Limit the results to 5 only
```

Current weather radial search (circle)

In one shot, you can query for currently observed weather for all the cities whose lon/lat coordinates lie inside a circle whose center is the geocoords you provide. You can control how many cities you want to find by using the `limit` parameter.

The radius of the search circle is automatically determined to include the number of cities that you want to obtain (default is: 10)

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
obs_list = mgr.weather_around_coords(57, -2.15, limit=8)  # Find observed weather for␣
↪all the places in the
                                                           # surroundings of lat=57,
↪lon=-2.15, limit results to 8 only
```

Current weather search in bounding box

In one shot, you can query for currently observed weather for all the cities whose lon/lat coordinates lie inside the specified rectangle (bounding box)

A bounding box is determined by specifying:

- the north latitude boundary (`lat_top`)
- the south latitude boundary (`lat_bottom`)
- the west longitude boundary (`lon_left`)
- the east longitude boundary (`lon_right`)

Also, an integer `zoom` level needs to be specified (defaults to 10): this works along with . The lower the zoom level, the “higher in the sky” OWM looks for cities inside the bounding box (think of it as the inverse of elevation)

The `clustering` parameter is off by default. With `clustering=True` you ask for server-side clustering of cities: this will result in fewer results when the bounding box shows high city density

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()

# This bounding box roughly encloses Cairo city (Egypt)
lat_top = 30.223475116500158
lat_bottom = 29.888280933159265
lon_left = 31.0034179688
lon_right = 31.5087890625

# This which should give you around 5 results
obs_list = mgr.weather_at_places_in_bbox(lon_left, lat_bottom, lon_right, lat_top,␣
↪zoom=10)
```

(continues on next page)

(continued from previous page)

```
# This only gives 1
obs_list = mgr.weather_at_places_in_bbox(lon_left, lat_bottom, lon_right, lat_top,␣
↪zoom=5)
```

Weather forecasts

Get forecast on a location

Just like for observed weather info, you can fetch weather forecast info on a specific toponym. As usual, provide toponym + country code for better results.

Forecast are provided for the next 5 days.

A `Forecast` object contains a list of `Weather` objects, each one having a specific reference time in the future. The time interval among `Weather` objects can be 1 day (`daily` forecast) or 3 hours ('3h' forecast).

Let's fetch forecast on Berlin (Germany):

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
daily_forecast = mgr.forecast_at_place('Berlin,DE', 'daily').forecast
3h_forecast = mgr.forecast_at_place('Berlin,DE', '3h').forecast
```

Now that you got the `Forecast` object, you can either manipulate it directly or use PyOWM conveniences to quickly slice and dice the embedded `Weather` objects

Let's take a look at the first option (see further on for the second one): a `Forecast` object is iterable on the weathers

```
nr_of_weathers = len(daily_forecast)
for weather in daily_forecast:
    weather.get_reference_time('iso'), weather.get_status()  # ('2020-03-10 14:00:00+0
↪','Clear')
                                                              # ('2020-03-11 14:00:00+0
↪','Clouds')
                                                              # ('2020-03-12 14:00:00+0
↪','Clouds')
                                                              # ...
```

Something useful is forecast actualization, as you might want to remove from the `Forecast` all the embedded `Weather` objects that refer to a time in the past with respect to now. This is useful especially if store the fetched forecast for subsequent computations.

```
# Say now is: 2020-03-10 18:30:00+0
daily_forecast.actualize()
for weather in daily_forecast:
    weather.get_reference_time('iso'), weather.get_status()  # ('2020-03-11 14:00:00+0
↪','Clouds')
                                                              # ('2020-03-12 14:00:00+0
↪','Clouds')
                                                              # ...
```

Know when a forecast weather streak starts and ends

Say we get the 3h forecast on Berlin. You want to know when the forecasted weather streak starts and ends

Use the `Forecaster` convenience class as follows.

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
forecaster = mgr.forecast_at_place('Berlin,DE', '3h')     # this gives you a␣
↪Forecaster object
forecaster.when_starts('iso')                             # 2020-03-10 14:00:00+00'
forecaster.when_ends('iso')                               # 2020-03-16 14:00:00+00'
```

Get forecasted weather for tomorrow

Say you want to know the weather on Berlin, say, globally for tomorrow. Easily done with the `Forecaster` convenience class and PyOWM's `timestamps` utilities:

```
from pyowm.utils import timestamps
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
daily_forecaster = mgr.forecast_at_place('Berlin,DE', 'daily')
tomorrow = timestamps.tomorrow()                                  # datetime object␣
↪for tomorrow
weather = daily_forecaster.get_weather_at(tomorrow)               # the weather you
↪'re looking for
```

Then say you want to know weather for tomorrow on Berlin at 5 PM:

```
from pyowm.utils import timestamps
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
3h_forecaster = mgr.forecast_at_place('Berlin,DE', '3h')
tomorrow_at_five = timestamps.tomorrow(17, 0)                     # datetime object␣
↪for tomorrow at 5 PM
weather = 3h_forecaster.get_weather_at(tomorrow_at_five)          # the weather you
↪'re looking for
```

You are provided with the `Weather` object that lies closest to the time that you specified (5 PM)

Is it going to rain tomorrow?

Say you want to know if you need to carry an umbrella around in Berlin tomorrow.

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
3h_forecaster = mgr.forecast_at_place('Berlin,DE', '3h')

# Is it going to rain tomorrow?
tomorrow = timestamps.tomorrow()                        # datetime object for tomorrow
3h_forecaster.will_be_rainy_at(tomorrow)                # True
```

Will it snow or be foggy in the next days?

In Berlin:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
3h_forecaster = mgr.forecast_at_place('Berlin,DE', '3h')

# Is it going to be snowy in the next 5 days ?
3h_forecaster.will_have_snow()     # False

# Is it going to be foggy in the next 5 days ?
3h_forecaster.will_have_fog()     # True
```

When will the weather be sunny in the next five days?

Always in Berlin:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
daily_forecaster = mgr.forecast_at_place('Berlin,DE', 'daily')

list_of_weathers = daily_forecaster.when_clear()
```

This will give you the list of `Weather` objects in the 5 days forecast when it will be sunny. So if only 2 in the next 5 days will be sunny, you'll get 2 objects The list will be empty if none of the upcoming days will be sunny.

Which of the next 5 days will be the coldest? And which one the most rainy ?

Always in Berlin:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
daily_forecaster = mgr.forecast_at_place('Berlin,DE', 'daily')

daily_forecaster.most_cold()      # this weather is of the coldest day
daily_forecaster.most_rainy()     # this weather is of the most rainy day
```

Get forecast on geographic coordinates

TBD

Get forecast on city ID

TBD

Get forecast on geographic coordinates

TBD

Meteostation historic measurements

This is a legacy feature of the OWM Weather API

Weather data measurements history for a specific meteostation is available in three sampling intervals:

- `'tick'` (which stands for minutely)
- `'hour'`
- `'day'`

The amount of datapoints returned can be limited. Queries can be made as follows:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()

station_id = 39276

# Get tick historic data for a meteostation
historian = mgr.station_tick_history(station_id, limit=4)  # only 4 data items

# Get hourly historic data for the same station, no limits
historian = mgr.station_hour_history(station_id)

# Get hourly historic data for the same station, no limits
historian = mgr.station_day_history(station_id)
```

All of the above mentioned calls return a `Historian` object. Each measurement is composed by:

- a UNIX epoch timestamp
- a temperature sample
- a humidity sample
- a pressure sample
- a rain volume sample
- wind speed sample

Use the convenience methods provided by `Historiam` to get time series for temperature, wind, etc.. These convenience methods are especially useful if you need to chart the historic time series of the measured physical entities:

```

# Get the temperature time series (in different units of measure)
historian.temperature_series()                     # defaults to Kelvin, eg. ␣
↪[(1381327200, 293.4), (1381327260, 293.6), ...]
historian.temperature_series(unit="celsius")       # now in Celsius
historian.temperature_series("fahrenheit")         # you get the gig

# Get the humidity time series
historian.humidity_series()

```

(continues on next page)

(continued from previous page)

```
# Get the pressure time series
historian.pressure_series()

# Get the rain volume time series
historian.rain_series()

# Get the wind speed time series
historian.wind_series()
```

Each of the above methods returns a list of tuples, each tuple being a couple in the form: `(UNIX epoch, measured value)`. Be aware that whenever measured values are missing `None` placeholders are put.

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

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

# Get the maximum rain value in the series
historian.max_rain()  # eg. (1381327200, 20.25)

# Get the average wind value in the series
historian.average_wind()  # eg. 4.816
```

Get raw meteostation measurements data

Make the proper call based on the sampling interval of interest and obtain the resulting `Historian` object:

```
raw_measurements_dict = historian.station_history.measurements  # dict of raw␣
↪measurement dicts, indexed by time of sampling:
```

The `raw_measurements_dict` contains multiple sub-dicts, each one being a a data item. Example:

```
{
  1362933983: {
    "temperature": 266.25,
    "humidity": 27.3,
    "pressure": 1010.02,
    "rain": None,
    "wind": 4.7
  }
  # [...]
}
```

OneCall data

With the OneCall Api you can get the current weather, hourly forecast for the next 48 hours and the daily forecast for the next seven days in one call.

One Call objects can be thought of as datasets that "photograhp" of observed and forecasted weather data for a location: such photos are given for a specific timestamp.

It is possible to get:

- current OneCall data: the "photo" given for today)

- historical OneCall data: “photos” given for past days, up to 5

Current OneCall data

What is the feels like temperature (℃) tomorrow morning?

Always in Berlin:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
one_call = mgr.one_call(lat=52.5244, lon=13.4105)

one_call.forecast_daily[0].temperature('celsius').get('feels_like_morn', None) #Ex.:␣
↪7.7
```

What’s the wind speed in three hours?

Attention: The first entry in forecast_hourly is the current hour. If you send the request at 18:36 UTC then the first entry in forecast_hourly is from 18:00 UTC.

Always in Berlin:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
one_call = mgr.one_call(lat=52.5244, lon=13.4105)

one_call.forecast_hourly[3].wind().get('speed', 0) # Eg.: 4.42
```

What’s the current humidity?

Always in Berlin:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
one_call = mgr.one_call(lat=52.5244, lon=13.4105)

one_call.current.humidity # Eg.: 81
```

Historical OneCall data

Remember the “photograph” metaphor for OneCall data. You can query for “photos” given for past days: when you do that, be aware that such a photo carries along weather forecasts (hourly and daily) that *might* refer to the past

This is because - as said above - the One Call API returns hourly forecasts for a streak of 48 hours and daily forecast for a streak of 7 days, both streaks beginning from the timestamp which the OneCall object refers to

In case of doubt, anyway, you can always *check the reference timestamp* for the `Weather` objects embedded into the OneCall object and check if it’s in the past or not.

What was the observed weather yesterday at this time?

Always in Berlin:

```
from pyowm.owm import OWM
from pyowm.utils import timestamps, formatting

owm = OWM('your-api-key')
mgr = owm.weather_manager()

# what is the epoch for yesterday at this time?
yesterday_epoch = formatting.to_UNIXtime(timestamps.yesterday())

one_call_yesterday = mgr.one_call_history(lat=52.5244, lon=13.4105, dt=yesterday_
↪epoch)

observed_weather = one_call_yesterday.current
```

What was the weather forecasted 3 days ago for the subsequent 48 hours ?

No way we move from Berlin:

```
from pyowm.owm import OWM
from pyowm.utils import timestamps
from datetime import datetime, timedelta, timezone

owm = OWM('your-api-key')
mgr = owm.weather_manager()

# what is the epoch for 3 days ago at this time?
three_days_ago_epoch = int((datetime.now() - timedelta(days=3)).
↪replace(tzinfo=timezone.utc).timestamp())

one_call_three_days_ago = mgr.one_call_history(lat=52.5244, lon=13.4105, dt=three_
↪days_ago_epoch)

list_of_forecasted_weathers = one_call_three_days_ago.forecast_hourly
```

5.1.2 PyOWM v3 software API documentation

This is the Python API documentation of PyOWM:

pyowm package

Subpackages

pyowm.agroapi10 package

Submodules

pyowm.agroapi10.agro_manager module

class `pyowm.agroapi10.agro_manager.`**`AgroManager`**(*API_key*, *config*)

Bases: `object`

A manager objects that provides a full interface to OWM Agro API.

Parameters

- **`API_key`** (`str`) – the OWM Weather API key
- **`config`** (`dict`) – the configuration dictionary

Returns an *AgroManager* instance

Raises *AssertionError* when no API Key is provided

`agro_api_version`()

`create_polygon`(*geopolygon*, *name=None*)

Create a new polygon on the Agro API with the given parameters

Parameters

- **`geopolygon`** (*pyowm.utils.geo.Polygon* instance) – the geopolygon representing the new polygon
- **`name`** (`str`) – optional mnemonic name for the new polygon

Returns a *pyowm.agro10.polygon.Polygon* instance

`delete_polygon`(*polygon*)

Deletes on the Agro API the Polygon identified by the ID of the provided polygon object.

Parameters **`polygon`** (*pyowm.agro10.polygon.Polygon* instance) – the *pyowm.agro10.polygon.Polygon* object to be deleted

Returns *None* if deletion is successful, an exception otherwise

`download_satellite_image`(*metaimage*, *x=None*, *y=None*, *zoom=None*, *palette=None*)

Downloads the satellite image described by the provided metadata. In case the satellite image is a tile, then tile coordinates and zoom must be provided. An optional palette ID can be provided, if supported by the downloaded preset (currently only NDVI is supported)

Parameters

- **`metaimage`** (a *pyowm.agroapi10.imagery.MetaImage* subtype) – the satellite image's metadata, in the form of a *MetaImage* subtype instance
- **`x`** (int or *None*) – x tile coordinate (only needed in case you are downloading a tile image)
- **`y`** (int or *None*) – y tile coordinate (only needed in case you are downloading a tile image)
- **`zoom`** (int or *None*) – zoom level (only needed in case you are downloading a tile image)
- **`palette`** (str or *None*) – ID of the color palette of the downloaded images. Values are provided by *pyowm.agroapi10.enums.PaletteEnum*

Returns a *pyowm.agroapi10.imagery.SatelliteImage* instance containing both image's metadata and data

`get_polygon`(*polygon_id*)

Retrieves a named polygon registered on the Agro API.

Parameters **`id`** (`str`) – the ID of the polygon

Returns a *pyowm.agro10.polygon.Polygon* object

`get_polygons`()

Retrieves all of the user's polygons registered on the Agro API.

Returns list of *pyowm.agro10.polygon.Polygon* objects

`search_satellite_imagery`(*polygon_id*, *acquired_from*, *acquired_to*, *img_type=None*, *preset=None*, *min_resolution=None*, *max_resolution=None*, *acquired_by=None*, *min_cloud_coverage=None*, *max_cloud_coverage=None*, *min_valid_data_coverage=None*, *max_valid_data_coverage=None*)

Searches on the Agro API the metadata for all available satellite images that contain the specified polygon and acquired during the specified time interval; and optionally matching the specified set of filters: - image type (eg. GeoTIF) - image preset (eg. false color, NDVI, …) - min/max acquisition resolution - acquiring satellite - min/max cloud coverage on acquired scene - min/max valid data coverage on acquired scene

Parameters

- **`polygon_id`** (`str`) – the ID of the reference polygon
- **`acquired_from`** (`int`) – lower edge of acquisition interval, UNIX timestamp
- **`acquired_to`** (`int`) – upper edge of acquisition interval, UNIX timestamp
- **`img_type`** (*pyowm.commons.databoxes.ImageType*) – the desired file format type of the images. Allowed values are given by *pyowm.commons.enums.ImageTypeEnum*
- **`preset`** (`str`) – the desired preset of the images. Allowed values are given by *pyowm.agroapi10.enums.PresetEnum*
- **`min_resolution`** (`int`) – minimum resolution for images, px/meters
- **`max_resolution`** (`int`) – maximum resolution for images, px/meters
- **`acquired_by`** (`str`) – short symbol of the satellite that acquired the image (eg. “l8”)
- **`min_cloud_coverage`** (`int`) – minimum cloud coverage percentage on acquired images
- **`max_cloud_coverage`** (`int`) – maximum cloud coverage percentage on acquired images
- **`min_valid_data_coverage`** (`int`) – minimum valid data coverage percentage on acquired images
- **`max_valid_data_coverage`** (`int`) – maximum valid data coverage percentage on acquired images

Returns a list of *pyowm.agro10.imagery.MetaImage* subtypes instances

`soil_data`(*polygon*)

Retrieves the latest soil data on the specified polygon

Parameters **`polygon`** (*pyowm.agro10.polygon.Polygon* instance) – the reference polygon you want soil data for

Returns a *pyowm.agro10.soil.Soil* instance

`stats_for_satellite_image`(*metaimage*)

Retrieves statistics for the satellite image described by the provided metadata. This is currently only supported ‘EVI’ and ‘NDVI’ presets

Parameters **`metaimage`** (a *pyowm.agroapi10.imagery.MetaImage* subtype) – the satellite image's metadata, in the form of a *MetaImage* subtype instance

Returns dict

update_polygon(*polygon*)

Updates on the Agro API the Polygon identified by the ID of the provided polygon object. Currently this only changes the mnemonic name of the remote polygon

Parameters **polygon** (*pyowm.agro10.polygon.Polygon* instance) – the *pyowm.agro10.polygon.Polygon* object to be updated

Returns *None* if update is successful, an exception otherwise

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

class pyowm.agroapi10.imagery.**MetaGeoTiffImage**(*url*, *preset*, *satellite_name*, *acquisition_time*, *valid_data_percentage*, *cloud_coverage_percentage*, *sun_azimuth*, *sun_elevation*, *polygon_id=None*, *stats_url=None*)

Bases: `pyowm.agroapi10.imagery.MetaImage`

Class representing metadata for a satellite image of a polygon in GeoTiff format

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

class pyowm.agroapi10.imagery.**MetaImage**(*url*, *preset*, *satellite_name*, *acquisition_time*, *valid_data_percentage*, *cloud_coverage_percentage*, *sun_azimuth*, *sun_elevation*, *polygon_id=None*, *stats_url=None*)

Bases: `object`

A class representing metadata for a satellite-acquired image

Parameters

- **url** (`str`) – the public URL of the image
- **preset** (`str`) – the preset of the image (supported values are listed by *pyowm.agroapi10.enums.PresetEnum*)
- **satellite_name** (`str`) – the name of the satellite that acquired the image (supported values are listed by *pyowm.agroapi10.enums.SatelliteEnum*)
- **acquisition_time** (`int`) – the UTC Unix epoch when the image was acquired
- **valid_data_percentage** (`float`) – approximate percentage of valid data coverage
- **cloud_coverage_percentage** (`float`) – approximate percentage of cloud coverage on the scene
- **sun_azimuth** (`float`) – sun azimuth angle at scene acquisition time
- **sun_elevation** (`float`) – sun zenith angle at scene acquisition time
- **polygon_id** (`str`) – optional id of the polygon the image refers to
- **stats_url** (str or *None*) – the public URL of the image statistics, if available

Returns an *MetaImage* object

acquisition_time(*timeformat='unix'*)

Returns the UTC time telling when the image data was acquired by the satellite

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

image_type = None

class pyowm.agroapi10.imagery.**MetaPNGImage**(*url*, *preset*, *satellite_name*, *acquisition_time*, *valid_data_percentage*, *cloud_coverage_percentage*, *sun_azimuth*, *sun_elevation*, *polygon_id=None*, *stats_url=None*)

Bases: `pyowm.agroapi10.imagery.MetaImage`

Class representing metadata for a satellite image of a polygon in PNG format

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

`class` `pyowm.agroapi10.imagery.`**`MetaTile`**(*url*, *preset*, *satellite_name*, *acquisition_time*, *valid_data_percentage*, *cloud_coverage_percentage*, *sun_azimuth*, *sun_elevation*, *polygon_id=None*, *stats_url=None*)

Bases: `pyowm.agroapi10.imagery.MetaImage`

Class representing metadata for a tile in PNG format

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

`class` `pyowm.agroapi10.imagery.`**`SatelliteImage`**(*metadata*, *data*, *downloaded_on=None*, *palette=None*)

Bases: `object`

Class representing a downloaded satellite image, featuring both metadata and data

Parameters

- **`metadata`** (a *pyowm.agro10.imagery.MetaImage* subtype instance) – the metadata for this satellite image
- **`data`** (either *pyowm.commons.image.Image* or *pyowm.commons.tile.Tile* object) – the actual data for this satellite image
- **`downloaded_on`** (int or *None*) – the UNIX epoch this satellite image was downloaded at
- **`palette`** (str or *None*) – ID of the color palette of the downloaded images. Values are provided by *pyowm.agroapi10.enums.PaletteEnum*

Returns a *pyowm.agroapi10.imagery.SatelliteImage* instance

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

Returns the UTC time telling when the satellite image was downloaded from the OWM Agro 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

`persist`(*path_to_file*)

Saves the satellite image to disk on a file

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

Returns *None*

pyowm.agroapi10.polygon module

`class` `pyowm.agroapi10.polygon.`**`Polygon`**(*id*, *name=None*, *geopolygon=None*, *center=None*, *area=None*, *user_id=None*)

Bases: `object`

A Polygon feature, foundational element for all Agro API operations

Parameters

- **`id`** (`str`) – the polygon's ID

- **name** – the polygon's name
- **geopolygon** (*pyowm.utils.geo.Polygon*) – the *pyowm.utils.geo.Polygon* instance that represents this polygon
- **center** (*pyowm.utils.geo.Point*) – the *pyowm.utils.geo.Point* instance that represents the central point of the polygon
- **area** (`float or int`) – the area of the polygon in hectares
- **user_id** (`str`) – the ID of the user owning this polygon

Returns a *Polygon* instance

Raises *AssertionError* when either id is *None* or geopolygon, center or area have wrong type

area_km

classmethod from_dict(*the_dict*)

pyowm.agroapi10.search module

class `pyowm.agroapi10.search.`**SatelliteImagerySearchResultSet**(*polygon_id*, *list_of_dict*, *query_timestamp*)

Bases: `object`

Class representing a filterable result set by a satellite imagery search against the Agro API 1.0. Each result is a *pyowm.agroapi10.imagery.MetaImage* subtype instance

all()

Returns all search results

Returns a list of *pyowm.agroapi10.imagery.MetaImage* instances

issued_on(*timeformat='unix'*)

Returns the UTC time telling when the query was performed against the OWM Agro 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

with_img_type(*image_type*)

Returns the search results having the specified image type

Parameters **image_type** (*pyowm.commons.databoxes.ImageType* instance) – the desired image type (valid values are provided by the *pyowm.commons.enums.ImageTypeEnum* enum)

Returns a list of *pyowm.agroapi10.imagery.MetaImage* instances

with_img_type_and_preset(*image_type*, *preset*)

Returns the search results having both the specified image type and preset

Parameters

- **image_type** (*pyowm.commons.databoxes.ImageType* instance) – the desired image type (valid values are provided by the *pyowm.commons.enums.ImageTypeEnum* enum)
- **preset** (`str`) – the desired image preset (valid values are provided by the *pyowm.agroapi10.enums.PresetEnum* enum)

Returns a list of *pyowm.agroapi10.imagery.MetaImage* instances

`with_preset`(*preset*)
Returns the seach results having the specified preset

Parameters **`preset`** (`str`) – the desired image preset (valid values are provided by the *pyowm.agroapi10.enums.PresetEnum* enum)

Returns a list of *pyowm.agroapi10.imagery.MetaImage* instances

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

`to_dict`()

pyowm.agroapi10.uris module

Module contents

pyowm.alertapi30 package

Submodules

pyowm.alertapi30.alert_manager module

class `pyowm.alertapi30.alert_manager.`**`AlertManager`**(*API_key*, *config*)

Bases: `object`

A manager objects that provides a full interface to OWM Alert API. It implements CRUD methods on Trigger entities and read/deletion of related Alert objects

Parameters

- **`API_key`** (*`str`*) – the OWM Weather API key
- **`config`** (*`dict`*) – the configuration dictionary

Returns an *AlertManager* instance

Raises *AssertionError* when no API Key is provided

`alert_api_version`()

`create_trigger`(*start*, *end*, *conditions*, *area*, *alert_channels=None*)

Create a new trigger on the Alert API with the given parameters :param start: time object representing the time when the trigger begins to be checked :type start: int, `datetime.datetime` or ISO8601-formatted string :param end: time object representing the time when the trigger ends to be checked :type end: int, `datetime.datetime` or ISO8601-formatted string :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 :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

`delete_alert`(*alert*)

Deletes the specified alert from the Alert API :param alert: the alert to be deleted :type alert: pyowm.alertapi30.alert.Alert‘ :return: `None` if the deletion was successful, an error otherwise

`delete_all_alerts_for`(*trigger*)

Deletes all of the alert that were fired for the specified Trigger :param trigger: the trigger whose alerts are to be cleared :type trigger: *pyowm.alertapi30.trigger.Trigger* :return: *None* if deletion is successful, an exception otherwise

`delete_trigger`(*trigger*)

Deletes from the Alert API the trigger record identified by the ID of the provided *pyowm.alertapi30.trigger.Trigger*, along with all related alerts

Parameters **`trigger`** (*pyowm.alertapi30.trigger.Trigger*) – the *pyowm.alertapi30.trigger.Trigger* object to be deleted

Returns *None* if deletion is successful, an exception otherwise

get_alert (*alert_id*, *trigger*)
Retrieves info about the alert record on the Alert API that has the specified ID and belongs to the specified parent Trigger object :param trigger: the parent trigger :type trigger: *pyowm.alertapi30.trigger.Trigger* :param alert_id: the ID of the alert :type alert_id *pyowm.alertapi30.alert.Alert* :return: an *pyowm.alertapi30.alert.Alert* instance

get_alerts_for (*trigger*)
Retrieves all of the alerts that were fired for the specified Trigger :param trigger: the trigger :type trigger: *pyowm.alertapi30.trigger.Trigger* :return: list of *pyowm.alertapi30.alert.Alert* objects

get_trigger (*trigger_id*)
Retrieves the named trigger from the Weather Alert API.

Parameters **trigger_id** (`str`) – the ID of the trigger

Returns a *pyowm.alertapi30.trigger.Trigger* instance

get_triggers ()
Retrieves all of the user's triggers that are set on the Weather Alert API.

Returns list of *pyowm.alertapi30.trigger.Trigger* objects

update_trigger (*trigger*)
Updates on the Alert API the trigger record having the ID of the specified Trigger object: the remote record is updated with data from the local Trigger object.

Parameters **trigger** (*pyowm.alertapi30.trigger.Trigger*) – the Trigger with updated data

Returns `None` if update is successful, an error otherwise

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

classmethod from_dict (*the_dict*)
Parses a *Alert* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **the_dict** (*dict*) – the input dictionary

Returns a *Alert* instance or `None` if no data is available

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

to_dict ()
Dumps object to a dictionary

Returns a *dict*

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

`to_dict`()

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*)

`to_dict`()

pyowm.alertapi30.enums module

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

Bases: `object`

Allowed alert channels

`OWM_API_POLLING = <pyowm.alertapi30.alert.AlertChannel - name:  OWM API POLLING>`

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

classmethod from_dict(*the_dict*)

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

`to_dict`()

Module contents

pyowm.commons package

Submodules

pyowm.commons.cityids

pyowm.commons.cityidregistry module

class `pyowm.commons.cityidregistry.`**`CityIDRegistry`**(*filepath_regex*)

Bases: `object`

`MATCHINGS = {'exact': <function CityIDRegistry.<lambda>>, 'like': <function CityIDRe`

`geopoints_for`(*city_name*, *country=None*, *matching='nocase'*)

Returns a list of `pyowm.utils.geo.Point` objects corresponding to the int IDs and relative toponyms and 2-chars country of the cities matching the provided city name. The rule for identifying matchings is according to the provided *matching* parameter value. If *country* is provided, the search is restricted to the cities of the specified country. :param country: two character str representing the country where to search for the city. Defaults to *None*, which means: search in all countries. :param matching: str among *exact* (literal, case-sensitive matching), *nocase* (literal, case-insensitive matching) and *like* (matches cities whose name contains as a substring the string fed to the function, no matter the case). Defaults to *nocase*. :raises ValueError if the value for *matching* is unknown :return: list of *pyowm.utils.geo.Point* objects

`classmethod get_instance`()

Factory method returning the default city ID registry :return: a *CityIDRegistry* instance

`ids_for`(*city_name*, *country=None*, *matching='nocase'*)

Returns a list of tuples in the form (long, str, str) corresponding to the int IDs and relative toponyms and 2-chars country of the cities matching the provided city name. The rule for identifying matchings is according to the provided *matching* parameter value. If *country* is provided, the search is restricted to the cities of the specified country. :param country: two character str representing the country where to search for the city. Defaults to *None*, which means: search in all countries. :param matching: str among *exact* (literal, case-sensitive matching), *nocase* (literal, case-insensitive matching) and *like* (matches cities whose name contains as a substring the string fed to the function, no matter the case). Defaults to *nocase*. :raises ValueError if the value for *matching* is unknown :return: list of tuples

`locations_for`(*city_name*, *country=None*, *matching='nocase'*)

Returns a list of Location objects corresponding to the int IDs and relative toponyms and 2-chars country of the cities matching the provided city name. The rule for identifying matchings is according to the provided *matching* parameter value. If *country* is provided, the search is restricted to the cities of the specified country. :param country: two character str representing the country where to search for the city. Defaults to *None*, which means: search in all countries. :param matching: str among *exact* (literal, case-sensitive matching), *nocase* (literal, case-insensitive matching) and *like* (matches cities whose name contains as a substring the string fed to the function, no matter the case). Defaults to *nocase*. :raises ValueError if the value for *matching* is unknown :return: list of *weatherapi25.location.Location* objects

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

class `pyowm.commons.databoxes.`**`SubscriptionType`**(*name*, *subdomain*, *is_paid*)

Bases: `object`

Databox class representing a type of subscription to OpenWeatherMap web APIs

Parameters

- **`name`** (*`str`*) – the name of the subscription
- **`subdomain`** (*`str`*) – the root API subdomain associated to the subscription
- **`is_paid`** (*`bool`*) – tells if the subscription plan is paid

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*)

class `pyowm.commons.enums.`**`SubscriptionTypeEnum`**

Bases: `object`

Allowed OpenWeatherMap subscription types

`DEVELOPER = <pyowm.commons.databoxes.SubscriptionType - name=developer subdomain=pro pa`

`ENTERPRISE = <pyowm.commons.databoxes.SubscriptionType - name=enterprise subdomain=pro`

`FREE = <pyowm.commons.databoxes.SubscriptionType - name=free subdomain=api paid=False>`

PROFESSIONAL = <pyowm.commons.databoxes.SubscriptionType - name=professional subdomain=

STARTUP = <pyowm.commons.databoxes.SubscriptionType - name=startup subdomain=pro paid=

classmethod items()

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

classmethod lookup_by_name(*name*)

pyowm.commons.exceptions module

exception pyowm.commons.exceptions.**APIRequestError**

Bases: *pyowm.commons.exceptions.PyOWMError*

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

exception pyowm.commons.exceptions.**APIResponseError**

Bases: *pyowm.commons.exceptions.PyOWMError*

Generic base class for exceptions representing HTTP error status codes in OWM Weather API responses

exception pyowm.commons.exceptions.**BadGatewayError**

Bases: *pyowm.commons.exceptions.APIRequestError*

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

exception pyowm.commons.exceptions.**ConfigurationError**

Bases: *pyowm.commons.exceptions.PyOWMError*

Generic base class for configuration related errors

exception pyowm.commons.exceptions.**ConfigurationNotFoundError**

Bases: *pyowm.commons.exceptions.ConfigurationError*

Raised when configuration source file is not available

exception pyowm.commons.exceptions.**ConfigurationParseError**

Bases: *pyowm.commons.exceptions.ConfigurationError*

Raised on failures in parsing configuration data

exception pyowm.commons.exceptions.**InvalidSSLCertificateError**

Bases: *pyowm.commons.exceptions.APIRequestError*

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

exception pyowm.commons.exceptions.**NotFoundError**

Bases: *pyowm.commons.exceptions.APIResponseError*

Error class that represents the situation when an entity is not found.

exception pyowm.commons.exceptions.**ParseAPIResponseError**

Bases: *pyowm.commons.exceptions.PyOWMError*

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

exception pyowm.commons.exceptions.**PyOWMError**

Bases: `Exception`

Generic base class for PyOWM exceptions

exception `pyowm.commons.exceptions.`**`TimeoutError`**

Bases: *`pyowm.commons.exceptions.APIRequestError`*

Error class that represents response timeout conditions

exception `pyowm.commons.exceptions.`**`UnauthorizedError`**

Bases: *`pyowm.commons.exceptions.APIResponseError`*

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

pyowm.commons.http_client module

class `pyowm.commons.http_client.`**`HttpClient`**(*api_key*, *config*, *root_uri*, *admits_subdomains=True*)

Bases: `object`

An HTTP client encapsulating some config data and abstarcting away data raw retrieval

Parameters

- **`api_key`** (*`str`*) – the OWM API key
- **`config`** (*`dict`*) – the configuration dictionary (if not provided, a default one will be used)
- **`root_uri`** (*`str`*) – the root URI of the API endpoint
- **`admits_subdomains`** (*`bool`*) – if the root URI of the API endpoint admits subdomains based on the subcription type (default: True)

classmethod check_status_code(*status_code*, *payload*)

delete(*path*, *params=None*, *data=None*, *headers=None*)

get_geotiff(*path*, *params=None*, *headers=None*)

get_json(*path*, *params=None*, *headers=None*)

get_png(*path*, *params=None*, *headers=None*)

post(*path*, *params=None*, *data=None*, *headers=None*)

put(*path*, *params=None*, *data=None*, *headers=None*)

class `pyowm.commons.http_client.`**`HttpRequestBuilder`**(*root_uri_token*, *api_key*, *config*, *has_subdomains=True*)

Bases: `object`

`URL_TEMPLATE_WITHOUT_SUBDOMAINS = '{}://{}/{}'`

A stateful HTTP URL, params and headers builder with a fluent interface

`URL_TEMPLATE_WITH_SUBDOMAINS = '{}://{}.{}/{}'`

build()

with_api_key()

with_header(*key*, *value*)

with_headers(*headers*)

with_language()

with_path(*path_uri_token*)

with_query_params(*query_params*)

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

class `pyowm.commons.tile.`**`Tile`**(*x*, *y*, *zoom*, *map_layer*, *image*)

Bases: `object`

Wrapper class for an image tile :param x: horizontal tile number in OWM tile reference system :type x: int :param y: vertical tile number in OWM tile reference system :type y: int :param zoom: zoom level for the tile :type zoom: int :param map_layer: the name of the OWM map layer this tile belongs to :type map_layer: str :param image: raw image data :type image: *pyowm.commons.image.Image instance*

`bounding_polygon`()

Returns the bounding box polygon for this tile

Returns *pywom.utils.geo.Polygon* instance

`classmethod geoocoords_to_tile_coords`(*lon*, *lat*, *zoom*)

Calculates the tile numbers corresponding to the specified geocoordinates at the specified zoom level Coordinates shall be provided in degrees and using the Mercator Projection (http://en.wikipedia.org/wiki/Mercator_projection)

Parameters

- **`lon`** (*`int or float`*) – longitude
- **`lat`** (*`int or float`*) – latitude
- **`zoom`** (*`int`*) – zoom level

Returns a tuple (x, y) containing the tile-coordinates

`persist`(*path_to_file*)

Saves the tile to disk on a file

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

Returns *None*

classmethod tile_coords_for_point(*geopoint*, *zoom*)

Returns the coordinates of the tile containing the specified geopoint at the specified zoom level

Parameters

- **geopoint** (*pywom.utils.geo.Point*) – the input geopoint instance
- **zoom** (`int`) – zoom level

Returns a tuple (x, y) containing the tile-coordinates

classmethod tile_coords_to_bbox(*x*, *y*, *zoom*)

Calculates the lon/lat estrema of the bounding box corresponding to specific tile coordinates. Output coodinates are in degrees and in the Mercator Projection (http://en.wikipedia.org/wiki/Mercator_projection)

Parameters

- **x** – the x tile coordinates
- **y** – the y tile coordinates
- **zoom** – the zoom level

Returns tuple with (lon_left, lat_bottom, lon_right, lat_top)

Module contents

pyowm.airpollutionapi30 package

Subpackages

Submodules

pyowm.airpollutionapi30.airpollution_client module

class pyowm.airpollutionapi30.airpollution_client.**AirPollutionHttpClient**(*API_key*, *httpclient*)

Bases: `object`

A class representing the OWM Air Pollution web API, which is a subset of the overall OWM API.

Parameters

- **API_key** (`Unicode`) – a Unicode object representing the OWM Air Pollution web API key
- **httpclient** (an *httpclient.HttpClient* instance) – an *httpclient.HttpClient* instance that will be used to send requests to the OWM Air Pollution web API.

get_coi(*params_dict*)

Invokes the CO Index endpoint

Parameters **params_dict** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

get_no2(*params_dict*)

Invokes the NO2 Index endpoint

Parameters **`params_dict`** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

`get_o3` (*params_dict*)
Invokes the O3 Index endpoint

Parameters **`params_dict`** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

`get_so2` (*params_dict*)
Invokes the SO2 Index endpoint

Parameters **`params_dict`** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

pyowm.airpollutionapi30.airpollution_manager module

class `pyowm.airpollutionapi30.airpollution_manager.`**`AirPollutionManager`**(*API_key*, *config*)

Bases: `object`

A manager objects that provides a full interface to OWM Air Pollution API.

Parameters

- **`API_key`** (*`str`*) – the OWM AirPollution API key
- **`config`** (*`dict`*) – the configuration dictionary

Returns an *AirPollutionManager* instance

Raises *AssertionError* when no API Key is provided

`airpollution_api_version`()

`coindex_around_coords`(*lat*, *lon*, *start=None*, *interval=None*)
Queries the OWM AirPollution API for Carbon Monoxide values sampled in the surroundings of the provided geocoordinates and in the specified time interval. A *COIndex* object instance is returned, encapsulating a *Location* object and the list of CO samples If *start* is not provided, the latest available CO samples are retrieved If *start* is provided but *interval* is not, then *interval* defaults to the maximum extent, which is: *year*

Parameters

- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0
- **`start`** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the start value of the search time window start (defaults to `None`). If not provided, the latest available CO samples value are retrieved
- **`interval`** (*`str among: 'minute', 'hour', 'day', 'month, 'year'`*) – the length of the search time window starting at *start* (defaults to `None`). If not provided, 'year' is used

Returns a *COIndex* instance or `None` if data is not available

Raises *ParseResponseException* when OWM AirPollution API responses' data cannot be parsed, *APICallException* when OWM AirPollution API can not be reached, *ValueError* for wrong input values

`no2index_around_coords`(*lat*, *lon*, *start=None*, *interval=None*)

Queries the OWM AirPollution API for Nitrogen Dioxide values sampled in the surroundings of the provided geocoordinates and in the specified time interval. A *NO2Index* object instance is returned, encapsulating a *Location* object and the list of NO2 samples If *start* is not provided, the latest available NO2 samples are retrieved If *start* is provided but *interval* is not, then *interval* defaults to the maximum extent, which is: *year*

Parameters

- **lat** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **lon** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0
- **start** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the start value of the search time window start (defaults to `None`). If not provided, the latest available NO2 samples value are retrieved
- **interval** (*`str among: 'minute', 'hour', 'day', 'month, 'year'`*) – the length of the search time window starting at *start* (defaults to `None`). If not provided, 'year' is used

Returns a *NO2Index* instance or `None` if data is not available

Raises *ParseResponseException* when OWM AirPollution API responses' data cannot be parsed, *APICallException* when OWM AirPollution API can not be reached, *ValueError* for wrong input values

`ozone_around_coords`(*lat*, *lon*, *start=None*, *interval=None*)

Queries the OWM AirPollution API for Ozone (O3) value in Dobson Units sampled in the surroundings of the provided geocoordinates and in the specified time interval. An *Ozone* object instance is returned, encapsulating a *Location* object and the UV intensity value. If *start* is not provided, the latest available ozone value is retrieved. If *start* is provided but *interval* is not, then *interval* defaults to the maximum extent, which is: *year*

Parameters

- **lat** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **lon** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0
- **start** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the start value of the search time window start (defaults to `None`). If not provided, the latest available Ozone value is retrieved
- **interval** (*`str among: 'minute', 'hour', 'day', 'month, 'year'`*) – the length of the search time window starting at *start* (defaults to `None`). If not provided, 'year' is used

Returns an *Ozone* instance or `None` if data is not available

Raises *ParseResponseException* when OWM AirPollution API responses' data cannot be parsed, *APICallException* when OWM AirPollution API can not be reached, *ValueError* for wrong input values

`so2index_around_coords`(*lat*, *lon*, *start=None*, *interval=None*)

Queries the OWM AirPollution API for Sulphur Dioxide values sampled in the surroundings of the provided geocoordinates and in the specified time interval. A *SO2Index* object instance is returned, encapsulating a *Location* object and the list of SO2 samples If *start* is not provided, the latest available SO2 samples are retrieved If *start* is provided but *interval* is not, then *interval* defaults to the maximum extent, which is: *year*

Parameters

- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0
- **`start`** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the start value of the search time window start (defaults to `None`). If not provided, the latest available SO2 samples value are retrieved
- **`interval`** (*`str among: 'minute', 'hour', 'day', 'month, 'year'`*) – the length of the search time window starting at *start* (defaults to `None`). If not provided, 'year' is used

Returns a *SO2Index* instance or `None` if data is not available

Raises *ParseResponseException* when OWM AirPollution API responses' data cannot be parsed, *APICallException* when OWM AirPollution API can not be reached, *ValueError* for wrong input values

pyowm.airpollutionapi30.coindex module

`class` `pyowm.airpollutionapi30.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

`classmethod from_dict`(*the_dict*)

Parses a *COIndex* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *COIndex* instance or `None` if no data is available

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

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

`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

`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

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

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

`to_dict`()
Dumps object to a dictionary

Returns a *dict*

pyowm.airpollutionapi30.ozone module

class `pyowm.airpollutionapi30.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

classmethod from_dict(*the_dict*)

Parses an *Ozone* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **the_dict** (*dict*) – the input dictionary

Returns an *Ozone* instance or `None` if no data is available

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

is_forecast()

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

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

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

to_dict()

Dumps object to a dictionary

Returns a *dict*

pyowm.airpollutionapi30.no2index module

class `pyowm.airpollutionapi30.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

classmethod from_dict(*the_dict*)

Parses an *NO2Index* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *NO2Index* instance or `None` if no data is available

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

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

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

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

to_dict()

Dumps object to a dictionary

Returns a *dict*

pyowm.airpollutionapi30.so2index module

class `pyowm.airpollutionapi30.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

classmethod from_dict(*the_dict*)

Parses a *SO2Index* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *SO2Index* instance or `None` if no data is available

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

is_forecast()

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

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

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

to_dict()

Dumps object to a dictionary

Returns a *dict*

Module contents

pyowm.stationsapi30 package

Subpackages

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

`classmethod from_dict`(*the_dict*)

Parses an *AggregatedMeasurement* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns an *AggregatedMeasurement* instance or `None` if no data is available

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

`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

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

classmethod from_dict(*the_dict*)

Parses a *Station* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **the_dict** (*dict*) – the input dictionary

Returns a *Station* instance or `None` if no data is available

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

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_dict()

Dumps object to a dictionary

Returns a *dict*

pyowm.stationsapi30.stations_manager module

class pyowm.stationsapi30.stations_manager.**StationsManager**(*API_key*, *config*)

Bases: `object`

A manager objects that provides a full interface to OWM Stations API. Mainly it implements CRUD methods on Station entities and the corresponding measured datapoints.

Parameters

- **API_key** (*str*) – the OWM Weather API key
- **config** (*dict*) – the configuration dictionary

Returns a *StationsManager* instance

Raises *AssertionError* when no API Key is provided

create_station(*external_id*, *name*, *lat*, *lon*, *alt=None*)

Create a new station on the Station API with the given parameters

Parameters

- **external_id** (*str*) – the user-given ID of the station
- **name** (*str*) – the name of the station
- **lat** (*float*) – latitude of the station
- **lon** (*float*) – longitude of the station
- **alt** (*float*) – altitude of the station

Returns the new *pyowm.stationsapi30.station.Station* object

`delete_station`(*station*)

Deletes the Station API record identified by the ID of the provided *pyowm.stationsapi30.station.Station*, along with all its related measurements

Parameters **`station`** (*pyowm.stationsapi30.station.Station*) – the *pyowm.stationsapi30.station.Station* object to be deleted

Returns *None* if deletion is successful, an exception otherwise

`get_measurements`(*station_id*, *aggregated_on*, *from_timestamp*, *to_timestamp*, *limit=100*)

Reads measurements of a specified station recorded in the specified time window and aggregated on minute, hour or day. Optionally, the number of resulting measurements can be limited.

Parameters

- **`station_id`** (`str`) – unique station identifier
- **`aggregated_on`** (`string between 'm','h' and 'd'`) – aggregation time-frame for this measurement
- **`from_timestamp`** (`int`) – Unix timestamp corresponding to the beginning of the time window
- **`to_timestamp`** (`int`) – Unix timestamp corresponding to the end of the time window
- **`limit`** (`int`) – max number of items to be returned. Defaults to 100

Returns list of *pyowm.stationsapi30.measurement.AggregatedMeasurement* objects

`get_station`(*id*)

Retrieves a named station registered on the Stations API.

Parameters **`id`** (`str`) – the ID of the station

Returns a *pyowm.stationsapi30.station.Station* object

`get_stations`()

Retrieves all of the user's stations registered on the Stations API.

Returns list of *pyowm.stationsapi30.station.Station* objects

`send_buffer`(*buffer*)

Posts to the Stations API data about the Measurement objects contained into the provided Buffer instance.

Parameters **`buffer`** (*pyowm.stationsapi30.buffer.Buffer* instance) – the *pyowm.stationsapi30.buffer.Buffer* instance whose measurements are to be posted

Returns *None* if creation is successful, an exception otherwise

`send_measurement`(*measurement*)

Posts the provided Measurement object's data to the Station API.

Parameters **`measurement`** (*pyowm.stationsapi30.measurement.Measurement* instance) – the *pyowm.stationsapi30.measurement.Measurement* object to be posted

Returns *None* if creation is successful, an exception otherwise

`send_measurements`(*list_of_measurements*)

Posts data about the provided list of Measurement objects to the Station API. The objects may be related to different station IDs.

Parameters **`list_of_measurements`** (list of *pyowm.stationsapi30.measurement.Measurement* instances) – list of *pyowm.stationsapi30.measurement.Measurement* objects to be posted

Returns *None* if creation is successful, an exception otherwise

`stations_api_version`()

`update_station`(*station*)

Updates the Station API record identified by the ID of the provided *pyowm.stationsapi30.station.Station* object with all of its fields

Parameters **`station`** (*pyowm.stationsapi30.station.Station*) – the *pyowm.stationsapi30.station.Station* object to be updated

Returns *None* if update is successful, an exception otherwise

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'`

pyowm.tiles.tile_manager module

`class` `pyowm.tiles.tile_manager.`**`TileManager`**(*API_key*, *map_layer*, *config*)

Bases: `object`

A manager objects that reads OWM map layers tile images .

Parameters

- **`API_key`** (`str`) – the OWM Weather API key
- **`map_layer`** (`str`) – the layer for which you want tiles fetched. Allowed map layers are specified by the *pyowm.tiles.enum.MapLayerEnum* enumerator class.
- **`config`** (`dict`) – the configuration dictionary

Returns a *TileManager* instance

Raises *AssertionError* when no API Key or no map layer is provided, or map layer name is not a string

`get_tile`(*x*, *y*, *zoom*)

Retrieves the tile having the specified coordinates and zoom level

Parameters

- **`x`** (`int`) – horizontal tile number in OWM tile reference system

- **y** (*int*) – vertical tile number in OWM tile reference system
- **zoom** (*int*) – zoom level for the tile

Returns a *pyowm.tiles.Tile* instance

Module contents

pyowm.utils package

Submodules

pyowm.utils.config module

`pyowm.utils.config.`**`get_config_from`**(*path_to_file*)

Loads configuration data from the supplied file and returns it.

Parameters **`path_to_file`** (*str*) – path to the configuration file

Returns the configuration *dict*

Raises *ConfigurationNotFoundError* when the supplied filepath is not a regular file; *ConfigurationParseError* when the supplied file cannot be parsed

`pyowm.utils.config.`**`get_default_config`**()

Returns the default PyOWM configuration.

Returns the configuration *dict*

`pyowm.utils.config.`**`get_default_config_for_proxy`**(*http_url*, *https_url*)

Returns the PyOWM configuration to be used behind a proxy server

Parameters

- **`http_url`** (*str*) – URL connection string for HTTP protocol
- **`https_url`** (*str*) – URL connection string for HTTPS protocol

Returns the configuration *dict*

`pyowm.utils.config.`**`get_default_config_for_subscription_type`**(*name*)

Returns the PyOWM configuration for a specific OWM API Plan subscription type

Parameters **`name`** (*str*) – name of the subscription type

Returns the configuration *dict*

pyowm.utils.decorators module

`pyowm.utils.config.`**`get_config_from`**(*path_to_file*)

Loads configuration data from the supplied file and returns it.

Parameters **`path_to_file`** (*str*) – path to the configuration file

Returns the configuration *dict*

Raises *ConfigurationNotFoundError* when the supplied filepath is not a regular file; *ConfigurationParseError* when the supplied file cannot be parsed `pyowm.utils.config.`**`get_default_config`**`()`

Returns the default PyOWM configuration.

Returns the configuration *dict*

`pyowm.utils.config.`**`get_default_config_for_proxy`**`(`*http_url*, *https_url*`)`

Returns the PyOWM configuration to be used behind a proxy server

Parameters

- **`http_url`** (*`str`*) – URL connection string for HTTP protocol
- **`https_url`** (*`str`*) – URL connection string for HTTPS protocol

Returns the configuration *dict*

`pyowm.utils.config.`**`get_default_config_for_subscription_type`**`(`*name*`)`

Returns the PyOWM configuration for a specific OWM API Plan subscription type

Parameters **`name`** (*`str`*) – name of the subscription type

Returns the configuration *dict*

pyowm.utils.geo module

`class` `pyowm.utils.geo.`**`Geometry`**

Bases: `object`

Abstract parent class for geotypes

`geojson``()`

Returns a GeoJSON string representation of this geotype, compliant to RFC 7946 (https://tools.ietf.org/html/rfc7946) :return: str

`to_dict``()`

Returns a dict representation of this geotype :return: dict

`class` `pyowm.utils.geo.`**`GeometryBuilder`**

Bases: `object`

`classmethod build``(`*the_dict*`)`

Builds a *pyowm.utils.geo.Geometry* subtype based on the geoJSON geometry type specified on the input dictionary :param the_dict: a geoJSON compliant dict :return: a *pyowm.utils.geo.Geometry* subtype instance :raises *ValueError* if unable to the geometry type cannot be recognized

`class` `pyowm.utils.geo.`**`MultiPoint`**`(`*list_of_tuples*`)`

Bases: *`pyowm.utils.geo.Geometry`*

A MultiPoint geotype. Represents a set of geographic points

Parameters **`list_of_tuples`** (*`list`*) – list of tuples, each one being the decimal (lon, lat) coordinates of a geopoint

Returns a *MultiPoint* instance

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

Builds a MultiPoint instance out of a geoJSON compliant dict :param the_dict: the geoJSON dict :return: *pyowm.utils.geo.MultiPoint* instance

`classmethod from_points``(`*iterable_of_points*`)`

Creates a MultiPoint from an iterable collection of *pyowm.utils.geo.Point* instances :param iterable_of_points: iterable whose items are *pyowm.utils.geo.Point* instances :type iterable_of_points: iterable :return: a *MultiPoint* instance

geojson()
Returns a GeoJSON string representation of this geotype, compliant to RFC 7946 (https://tools.ietf.org/html/rfc7946) :return: str

latitudes
List of decimal latitudes of this MultiPoint instance :return: list of tuples

longitudes
List of decimal longitudes of this MultiPoint instance :return: list of tuples

to_dict()
Returns a dict representation of this geotype :return: dict

class pyowm.utils.geo.**MultiPolygon**(*iterable_of_list_of_lists*)
Bases: *pyowm.utils.geo.Geometry*

A MultiPolygon geotype. Each MultiPolygon represents a set of (also djsjoint) Polygons. Each MultiPolygon is composed by an iterable whose elements are the list of lists defining a Polygon geotype. Please refer to the *pyowm.utils.geo.Point* documentation for details

Parameters **iterable_of_list_of_lists** (*iterable*) – iterable whose elements are list of lists of tuples

Returns a *MultiPolygon* instance

Raises *ValueError* when last point and fist point do not coincide or when no points are specified at all

classmethod from_dict(*the_dict*)
Builds a MultiPolygoninstance out of a geoJSON compliant dict :param the_dict: the geoJSON dict :return: *pyowm.utils.geo.MultiPolygon* instance

classmethod from_polygons(*iterable_of_polygons*)
Creates a *MultiPolygon* instance out of an iterable of Polygon geotypes :param iterable_of_polygons: list of *pyowm.utils.geo.Point* instances :type iterable_of_polygons: iterable :returns: a *MultiPolygon* instance

geojson()
Returns a GeoJSON string representation of this geotype, compliant to RFC 7946 (https://tools.ietf.org/html/rfc7946) :return: str

to_dict()
Returns a dict representation of this geotype :return: dict

class pyowm.utils.geo.**Point**(*lon*, *lat*)
Bases: *pyowm.utils.geo.Geometry*

A Point geotype. Represents a single geographic point

Parameters

- **lon** (*int of float*) – decimal longitude for the geopoint
- **lat** (*int of float*) – decimal latitude for the geopoint

Returns a *Point* instance

Raises *ValueError* when negative values are provided

bounding_square_polygon(*inscribed_circle_radius_km=10.0*)
Returns a square polygon (bounding box) that circumscribes the circle having this geopoint as centre and having the specified radius in kilometers. The polygon's points calculation is based on theory exposed by: http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates by Jan Philip Matuschek, owner of the intellectual property of such material. In short: - locally to the geopoint, the Earth's surface is approximated

to a sphere with radius = Earth's radius - the calculation works fine also when the bounding box contains the Earth's poles and the 180 deg meridian

Parameters **`inscribed_circle_radius_km`** (*`int or float`*) – the radius of the inscribed circle, defaults to 10 kms

Returns a *pyowm.utils.geo.Polygon* instance

`classmethod from_dict`(*the_dict*)
Builds a Point instance out of a geoJSON compliant dict :param the_dict: the geoJSON dict :return: *pyowm.utils.geo.Point* instance

`geojson`()
Returns a GeoJSON string representation of this geotype, compliant to RFC 7946 (https://tools.ietf.org/html/rfc7946) :return: str

`lat`

`lon`

`to_dict`()
Returns a dict representation of this geotype :return: dict

`class` `pyowm.utils.geo.`**`Polygon`**(*list_of_lists*)
Bases: *`pyowm.utils.geo.Geometry`*

A Polygon geotype. Each Polygon is made up by one or more lines: a line represents a set of connected geographic points and is conveyed by a list of points, the last one of which must coincide with the its very first one. As said, Polygons can be also made up by multiple lines (therefore, Polygons with "holes" are allowed) :param list_of_lists: list of lists, each sublist being a line and being composed by tuples - each one being the decimal (lon, lat) couple of a geopoint. The last point specified MUST coincide with the first one specified :type list_of_lists: list :returns: a *MultiPoint* instance :raises: *ValueError* when last point and fist point do not coincide or when no points are specified at all

`classmethod from_dict`(*the_dict*)
Builds a Polygon instance out of a geoJSON compliant dict :param the_dict: the geoJSON dict :return: *pyowm.utils.geo.Polygon* instance

`classmethod from_points`(*list_of_lists*)
Creates a *Polygon* instance out of a list of lists, each sublist being populated with *pyowm.utils.geo.Point* instances :param list_of_lists: list :type: list_of_lists: iterable_of_polygons :returns: a *Polygon* instance

`geojson`()
Returns a GeoJSON string representation of this geotype, compliant to RFC 7946 (https://tools.ietf.org/html/rfc7946) :return: str

`points`
Returns the list of *Point* instances representing the points of the polygon :return: list of *Point* objects

`to_dict`()
Returns a dict representation of this geotype :return: dict

`pyowm.utils.geo.`**`assert_is_lat`**(*val*)
Checks it the given value is a feasible decimal latitude

Parameters **`val`** (*`int of float`*) – value to be checked

Returns *None*

Raises *ValueError* if value is out of latitude boundaries, *AssertionError* if type is wrong

`pyowm.utils.geo.`**`assert_is_lon`**(*val*)
Checks it the given value is a feasible decimal longitude

Parameters **`val`** (*`int of float`*) – value to be checked

Returns *None*

Raises *ValueError* if value is out of longitude boundaries, *AssertionError* if type is wrong

pyowm.utils.measurables module

`pyowm.utils.measurables.`**`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.measurables.`**`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.measurables.`**`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.measurables.`**`metric_wind_dict_to_beaufort`**(*d*)

Converts all the wind values in a dict from meters/sec to the corresponding Beaufort scale level (which is not an exact number but rather represents a range of wind speeds - see: https://en.wikipedia.org/wiki/Beaufort_scale). Conversion table: https://www.windfinder.com/wind/windspeed.htm

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

Returns a dict with the same keys as the input dict and values converted to Beaufort level

`pyowm.utils.measurables.`**`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.measurables.`**`metric_wind_dict_to_km_h`**(*d*)

Converts all the wind values in a dict from meters/sec to km/hour.

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

Returns a dict with the same keys as the input dict and values converted to km/hour `pyowm.utils.measurables.`**`metric_wind_dict_to_knots`**(*d*)

Converts all the wind values in a dict from meters/sec to knots

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

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

pyowm.utils.formatting module

`pyowm.utils.formatting.`**`ISO8601_to_UNIXtime`**(*iso*)

Converts an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00` to the correspondant UNIXtime

Parameters **`iso`** (*`string`*) – the ISO8601-formatted string

Returns an int UNIXtime

Raises *TypeError* when bad argument types are provided, *ValueError* when the ISO8601 string is badly formatted

`class` `pyowm.utils.formatting.`**`UTC`**

Bases: `datetime.tzinfo`

`dst`(*dt*)

datetime -> DST offset as timedelta positive 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.formatting.`**`datetime_to_UNIXtime`**(*date*)

Converts a `datetime.datetime` object to its correspondent UNIXtime

Parameters **`date`** (`datetime.datetime`) – the `datetime.datetime` object

Returns an int UNIXtime

Raises *TypeError* when bad argument types are provided

`pyowm.utils.formatting.`**`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.formatting.`**`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.formatting.`**`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.formatting.`**`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.timestamps module

`pyowm.utils.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.timestamps.`**`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.weather module

`pyowm.utils.weather.`**`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.weather.`**`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.weather.`**`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.weather.`**`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.weather.`**`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

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

classmethod from_dict(*the_dict*)

Parses an *UVIndex* 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 **`the_dict`** (*`dict`*) – the input dict

Returns an *UVIndex* instance or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the input dict embeds an HTTP status error

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

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

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

to_dict()

Dumps object to a dictionary

Returns a *dict*

`pyowm.uvindexapi30.uvindex.`**`uv_intensity_to_exposure_risk`**(*uv_intensity*)

pyowm.uvindexapi30.uvindex_manager module

class `pyowm.uvindexapi30.uvindex_manager.`**`UVIndexManager`**(*API_key*, *config*)

Bases: `object`

A manager objects that provides a full interface to OWM UV Index API.

Parameters

- **`API_key`** (*`str`*) – the OWM UV Index API key
- **`config`** (*`dict`*) – the configuration dictionary

Returns a *UVIndexManager* instance

Raises *AssertionError* when no API Key is provided

`uvindex_api_version`()

`uvindex_around_coords`(*lat*, *lon*)

Queries for Ultra Violet value sampled in the surroundings of the provided geocoordinates and in the specified time interval. A *UVIndex* object instance is returned, encapsulating a *Location* object and the UV intensity value.

Parameters

- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0

Returns a *UVIndex* instance or `None` if data is not available

Raises *ParseResponseException* when OWM UV Index API responses' data cannot be parsed, *APICallException* when OWM UV Index API can not be reached, *ValueError* for wrong input values

`uvindex_forecast_around_coords`(*lat*, *lon*)

Queries for forecast Ultra Violet values in the next 8 days in the surroundings of the provided geocoordinates.

Parameters

- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0

Returns a list of *UVIndex* instances or empty list if data is not available

Raises *ParseResponseException* when OWM UV Index API responses' data cannot be parsed, *APICallException* when OWM UV Index API can not be reached, *ValueError* for wrong input values

`uvindex_history_around_coords`(*lat*, *lon*, *start*, *end=None*)

Queries for UV index historical values in the surroundings of the provided geocoordinates and in the specified time frame. If the end of the time frame is not provided, that is intended to be the current datetime.

Parameters

- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0

- **`lon`** (*`int/float`*) – the location’s longitude, must be between -180.0 and 180.0
- **`start`** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the time value for the start query boundary
- **`end`** (int, `datetime.datetime` or ISO8601-formatted string) – the object conveying the time value for the end query boundary (defaults to `None`, in which case the current datetime will be used)

Returns a list of *UVIndex* instances or empty list if data is not available

Raises *ParseResponseException* when OWM UV Index API responses’ data cannot be parsed, *APICallException* when OWM UV Index API can not be reached, *ValueError* for wrong input values

pyowm.uvindexapi30.uv_client module

class `pyowm.uvindexapi30.uv_client.`**`UltraVioletHttpClient`**(*API_key*, *httpclient*)

Bases: `object`

An HTTP client class for the OWM UV web API, which is a subset of the overall OWM API.

Parameters

- **`API_key`** (*`Unicode`*) – a Unicode object representing the OWM UV web API key
- **`httpclient`** (an *httpclient.HttpClient* instance) – an *httpclient.HttpClient* instance that will be used to send requests to the OWM Air Pollution web API.

`get_uvi`(*params_dict*)

Invokes the UV Index endpoint

Parameters **`params_dict`** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

`get_uvi_forecast`(*params_dict*)

Invokes the UV Index Forecast endpoint

Parameters **`params_dict`** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

`get_uvi_history`(*params_dict*)

Invokes the UV Index History endpoint

Parameters **`params_dict`** – dict of parameters

Returns a string containing raw JSON data

Raises *ValueError*, *APIRequestError*

pyowm.uvindexapi30.uris module

Module contents

pyowm.weatherapi25 package

Subpackages

Submodules

pyowm.weatherapi25.forecast module

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

classmethod `from_dict`(*the_dict*)

Parses a *Forecast* instance out of a raw data dictionary. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *Forecast* instance or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the input dictionary embeds an HTTP status error

`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

`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

to_dict()

Dumps object to a dictionary

Returns a *dict*

pyowm.weatherapi25.forecaster module

class `pyowm.weatherapi25.forecaster.`**Forecaster**(*forecast*)

Bases: `object`

A class providing convenience methods for manipulating weather forecast data. The class encapsulates a *Forecast* instance and provides abstractions on the top of it in order to let programmers exploit weather forecast data in a human-friendly fashion.

Parameters **forecast** (*Forecast*) – a *Forecast* instance

Returns a *Forecaster* instance

get_weather_at(*timeobject*)

Gives the *Weather* item in the forecast that is closest in time to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns a *Weather* object

most_cold()

Returns the *Weather* object in the forecast having the lowest min temperature. Was 'temp_min' key missing for every *Weather* instance in the forecast, `None` would be returned.

Returns a *Weather* object or `None` if no item in the forecast is eligible

most_hot()

Returns the *Weather* object in the forecast having the highest max temperature. Was 'temp_max' key missing for every *Weather* instance in the forecast, `None` would be returned.

Returns a *Weather* object or `None` if no item in the forecast is eligible

most_humid()

Returns the *Weather* object in the forecast having the highest humidty.

Returns a *Weather* object or `None` if no item in the forecast is eligible

most_rainy()

Returns the *Weather* object in the forecast having the highest precipitation volume. Was the 'all' key missing for every *Weather* instance in the forecast,‘‘None‘‘ would be returned.

Returns a *Weather* object or `None` if no item in the forecast is eligible

most_snowy()

Returns the *Weather* object in the forecast having the highest snow volume. Was the 'all' key missing for every *Weather* instance in the forecast, `None` would be returned.

Returns a *Weather* object or `None` if no item in the forecast is eligible

most_windy()

Returns the *Weather* object in the forecast having the highest wind speed. Was the 'speed' key missing for every *Weather* instance in the forecast, `None` would be returned.

Returns a *Weather* object or `None` if no item in the forecast is eligible

`when_clear``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having clear sky as weather condition.

Returns a list of *Weather* objects

`when_clouds``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having clouds as weather condition.

Returns a list of *Weather* objects

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

Returns the GMT time of the end of the forecast coverage, which is the time of the most recent *Weather* item in the forecast

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 a long or a str

Raises *ValueError* when invalid time format values are provided

`when_fog``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having fog as weather condition.

Returns a list of *Weather* objects

`when_hurricane``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having hurricane as weather condition.

Returns a list of *Weather* objects

`when_rain``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having rain as weather condition.

Returns a list of *Weather* objects

`when_snow``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having snow as weather condition.

Returns a list of *Weather* objects

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

Returns the GMT time of the start of the forecast coverage, which is the time of the most ancient *Weather* item in the forecast

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 a long or a str

Raises *ValueError* when invalid time format values are provided

`when_storm``()`

Returns a sublist of the *Weather* list in the forecast, containing only items having storm as weather condition.

Returns a list of *Weather* objects

when_tornado()
Returns a sublist of the *Weather* list in the forecast, containing only items having tornado as weather condition.

Returns a list of *Weather* objects

will_be_clear_at(*timeobject*)
Tells if at the specified time the condition is clear sky. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

will_be_cloudy_at(*timeobject*)
Tells if at the specified time the condition is clouds. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

will_be_foggy_at(*timeobject*)
Tells if at the specified time the condition is fog. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

will_be_hurricane_at(*timeobject*)
Tells if at the specified time the condition is hurricane. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

will_be_rainy_at(*timeobject*)
Tells if at the specified time the condition is rain. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

will_be_snowy_at(*timeobject*)
Tells if at the specified time the condition is snow. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **timeobject** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

`will_be_stormy_at`(*timeobject*)
Tells if at the specified time the condition is storm. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **`timeobject`** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

`will_be_tornado_at`(*timeobject*)
Tells if at the specified time the condition is tornado. The check is performed on the *Weather* item of the forecast which is closest to the time value conveyed by the parameter

Parameters **`timeobject`** (long/int, `datetime.datetime` or str)) – may be a UNIX time, a `datetime.datetime` object or an ISO8601-formatted string in the format `YYYY-MM-DD HH:MM:SS+00`

Returns boolean

`will_have_clear`()
Tells if into the forecast coverage exist one or more *Weather* items related to clear sky conditions

Returns boolean

`will_have_clouds`()
Tells if into the forecast coverage exist one or more *Weather* items related to cloud conditions

Returns boolean

`will_have_fog`()
Tells if into the forecast coverage exist one or more *Weather* items related to fog conditions

Returns boolean

`will_have_hurricane`()
Tells if into the forecast coverage exist one or more *Weather* items related to hurricanes

Returns boolean

`will_have_rain`()
Tells if into the forecast coverage exist one or more *Weather* items related to rain conditions

Returns boolean

`will_have_snow`()
Tells if into the forecast coverage exist one or more *Weather* items related to snow conditions

Returns boolean

`will_have_storm`()
Tells if into the forecast coverage exist one or more *Weather* items related to storms

Returns boolean

`will_have_tornado`()
Tells if into the forecast coverage exist one or more *Weather* items related to tornadoes

Returns boolean

pyowm.weatherapi25.historian module

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

`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

`class` `pyowm.weatherapi25.location.`**`Location`**`(`*name*, *lon*, *lat*, *_id*, *country=None*`)`

Bases: `object`

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

Further reference about OWM city IDs can be found at: http://bugs.openweathermap.org/projects/api/wiki/Api_2_5_weather#3-By-city-ID

Parameters

- **`name`** (*`Unicode`*) – the location's toponym
- **`lon`** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0
- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **`_id`** – the location's OWM city ID
- **`country`** (*`Unicode`*) – the location's country (`None` by default)

Returns a *Location* instance

Raises *ValueError* if lon or lat values are provided out of bounds

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

Parses a *Location* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *Location* instance or `None` if no data is available

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

`to_dict``()`

Dumps object to a dictionary

Returns a *dict*

`to_geopoint``()`

Returns the geoJSON compliant representation of this location

Returns a `pyowm.utils.geo.Point` instance

pyowm.weatherapi25.observation module

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

classmethod from_dict(*the_dict*)

Parses an *Observation* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns an *Observation* instance or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the input dict embeds an HTTP status error

classmethod from_dict_of_lists(*the_dict*)

Parses a list of *Observation* instances out of raw input dict containing a list. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *list* of *Observation* instances or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the OWM API returns an HTTP status error

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

to_dict()

Dumps object to a dictionary

Returns a *dict*

pyowm.weatherapi25.stationhistory module

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

classmethod `from_dict`(*d*)

Parses a *StationHistory* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **`the_dict`** (*dict*) – the input dictionary

Returns a *StationHistory* instance or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the input dict embeds an HTTP status error

`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

`to_dict`()

Dumps object to a dictionary

Returns a *dict*

pyowm.weatherapi25.weather module

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*, *utc_offset=None*, *uvi=None*)

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** (*str*) – weather-related icon name
- **visibility_distance** (*float*) – visibility distance
- **dewpoint** (*float*) – dewpoint
- **humidex** (*float*) – Canadian humidex
- **heat_index** (*float*) – heat index
- **utc_offset** (*int or None*) – offset with UTC time zone in seconds
- **uvi** (*int, float or None*) – UV index

Returns a *Weather* instance

Raises *ValueError* when negative values are provided for non-negative quantities

classmethod from_dict(*the_dict*)

Parses a *Weather* instance out of a data dictionary. Only certain properties of the data dictionary are used: if these properties are not found or cannot be parsed, an exception is issued.

Parameters **the_dict** (*dict*) – the input dictionary

Returns a *Weather* instance or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the input dict embeds an HTTP status error

classmethod from_dict_of_lists(*the_dict*)

Parses a list of *Weather* instances out of an input dict. Only certain properties of the data are used: if these properties are not found or cannot be parsed, an error is issued.

Parameters **the_dict** (`dict`) – the input dict

Returns a list of *Weather* instances or `None` if no data is available

Raises *ParseAPIResponseError* if it is impossible to find or parse the data needed to build the result, *APIResponseError* if the input dict an HTTP status error

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 or a *datetime.datetime* object

Raises ValueError when negative values are provided

sunrise_time(*timeformat='unix'*)

Returns the GMT time of sunrise. Can be *None* in case of polar nights.

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 'None', an int, a str or a *datetime.datetime* object

Raises ValueError

sunset_time(*timeformat='unix'*)

Returns the GMT time of sunset. Can be *None* in case of polar days.

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 'None', an int, a str or a *datetime.datetime* object

Raises ValueError

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

to_dict()

Dumps object to a dictionary

Returns a *dict*

weather_icon_url()

Returns weather-related icon URL as a string.

Returns the icon URL.

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), '*miles_hour*, '*kilometers_hour*', '*knots*' or '*beaufort*'

Returns a dict containing wind info

pyowm.weatherapi25.weather_manager module

class `pyowm.weatherapi25.weather_manager.`**WeatherManager**(*API_key*, *config*)

Bases: `object`

A manager objects that provides a full interface to OWM Weather API.

Parameters

- **API_key** (*str*) – the OWM AirPollution API key
- **config** (*dict*) – the configuration dictionary

Returns a *WeatherManager* instance

Raises *AssertionError* when no API Key is provided

forecast_at_coords(*lat*, *lon*, *interval*, *limit=None*)

Queries the OWM Weather API for weather forecast for the specified geographic coordinates with the given time granularity. A *Forecaster* object is returned, containing a *Forecast*: this instance encapsulates *Weather* objects corresponding to the provided granularity.

Parameters

- **lat** (*int/float*) – location's latitude, must be between -90.0 and 90.0
- **lon** (*int/float*) – location's longitude, must be between -180.0 and 180.0
- **interval** (str among *3h* and 'daily') – the granularity of the forecast, among *3h* and 'daily'
- **limit** (int or `None`) – the maximum number of *Weather* items to be retrieved (default is `None`, which stands for any number of items)

Returns a *Forecaster* instance or `None` if forecast data is not available for the specified location

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached

forecast_at_id(*id*, *interval*, *limit=None*)

Queries the OWM Weather API for weather forecast for the specified city ID (eg: 5128581) with the given time granularity. A *Forecaster* object is returned, containing a *Forecast*: this instance encapsulates *Weather* objects corresponding to the provided granularity.

Parameters

- **id** (*int*) – the location's city ID
- **interval** (str among *3h* and 'daily') – the granularity of the forecast, among *3h* and 'daily'
- **limit** (int or `None`) – the maximum number of *Weather* items to be retrieved (default is `None`, which stands for any number of items)

Returns a *Forecaster* instance or `None` if forecast data is not available for the specified location

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached

`forecast_at_place`(*name*, *interval*, *limit=None*)

Queries the OWM Weather API for weather forecast for the specified location (eg: "London,uk") with the given time granularity. A *Forecaster* object is returned, containing a *Forecast*: this instance encapsulates *Weather* objects corresponding to the provided granularity.

Parameters

- **`name`** (*`str`*) – the location's toponym
- **`interval`** (str among *3h* and 'daily') – the granularity of the forecast, among *3h* and 'daily'
- **`limit`** (int or `None`) – the maximum number of *Weather* items to be retrieved (default is `None`, which stands for any number of items)

Returns a *Forecaster* instance or `None` if forecast data is not available for the specified location

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached

`one_call`(*lat: Union[int, float], lon: Union[int, float]*) → pyowm.weatherapi25.one_call.OneCall

Queries the OWM Weather API with one call for current weather information and forecast for the specified geographic coordinates. One Call API provides the following weather data for any geographical coordinate: - Current weather - Hourly forecast for 48 hours - Daily forecast for 7 days

A *OneCall* object is returned with the current data and the two forecasts.

Parameters

- **`lat`** (*`int/float`*) – location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – location's longitude, must be between -180.0 and 180.0

Returns a *OneCall* instance or `None` if the data is not available for the specified location

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached

`one_call_history`(*lat: Union[int, float], lon: Union[int, float], dt: int = None*)

Queries the OWM Weather API with one call for historical weather information for the specified geographic coordinates.

A *OneCall* object is returned with the current data and the two forecasts.

Parameters

- **`lat`** (*`int/float`*) – location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – location's longitude, must be between -180.0 and 180.0
- **`dt`** (*`int`*) – timestamp from when the historical data starts. Cannot be less then now - 5 days. Default = None means now - 5 days

Returns a *OneCall* instance or `None` if the data is not available for the specified location

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached

`station_day_history`(*station_ID*, *limit=None*)

Queries the OWM Weather API for historic weather data measurements for the specified meteostation (eg: 2865), sampled once a day. A *Historian* object instance is returned, encapsulating a *StationHistory* objects which contains the measurements. The total number of retrieved data points can be limited using the appropriate parameter

Parameters

- **`station_ID`** (*`int`*) – the numeric ID of the meteostation
- **`limit`** (int or `None`) – the maximum number of data points the result shall contain (default is `None`, which stands for any number of data points)

Returns a *Historian* instance or `None` if data is not available for the specified meteostation

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached, *ValueError* if the limit value is negative

`station_hour_history`(*station_ID*, *limit=None*)

Queries the OWM Weather API for historic weather data measurements for the specified meteostation (eg: 2865), sampled once a hour. A *Historian* object instance is returned, encapsulating a *StationHistory* objects which contains the measurements. The total number of retrieved data points can be limited using the appropriate parameter

Parameters

- **`station_ID`** (*`int`*) – the numeric ID of the meteostation
- **`limit`** (int or `None`) – the maximum number of data points the result shall contain (default is `None`, which stands for any number of data points)

Returns a *Historian* instance or `None` if data is not available for the specified meteostation

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached, *ValueError* if the limit value is negative

`station_tick_history`(*station_ID*, *limit=None*)

Queries the OWM Weather API for historic weather data measurements for the specified meteostation (eg: 2865), sampled once a minute (tick). A *StationHistory* object instance is returned, encapsulating the measurements: the total number of data points can be limited using the appropriate parameter

Parameters

- **`station_ID`** (*`int`*) – the numeric ID of the meteostation
- **`limit`** (int or `None`) – the maximum number of data points the result shall contain (default is `None`, which stands for any number of data points)

Returns a *StationHistory* instance or `None` if data is not available for the specified meteostation

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached, *ValueError* if the limit value is negative

`weather_api_version`()

`weather_around_coords`(*lat*, *lon*, *limit=None*)

Queries the OWM Weather API for the currently observed weather in all the locations in the proximity of the specified coordinates.

Parameters

- **`lat`** (*`int/float`*) – location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – location's longitude, must be between -180.0 and 180.0

- **`limit`** – the maximum number of *Observation* items in the returned list (default is `None`, which stands for any number of items)
- **`limit`** – int or `None`

Returns a list of *Observation* objects or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached, *ValueError* when coordinates values are out of bounds or negative values are provided for limit

`weather_at_coords`(*lat*, *lon*)

Queries the OWM Weather API for the currently observed weather at the specified geographic (eg: 51.503614, -0.107331).

Parameters

- **`lat`** (*`int/float`*) – the location's latitude, must be between -90.0 and 90.0
- **`lon`** (*`int/float`*) – the location's longitude, must be between -180.0 and 180.0

Returns an *Observation* instance or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed or *APICallException* when OWM Weather API can not be reached

`weather_at_id`(*id*)

Queries the OWM Weather API for the currently observed weather at the specified city ID (eg: 5128581)

Parameters **`id`** (*`int`*) – the location's city ID

Returns an *Observation* instance or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed or *APICallException* when OWM Weather API can not be reached

`weather_at_ids`(*ids_list*)

Queries the OWM Weather API for the currently observed weathers at the specified city IDs (eg: [5128581,87182])

Parameters **`ids_list`** (*`list of int`*) – the list of city IDs

Returns a list of *Observation* instances or an empty list if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed or *APICallException* when OWM Weather API can not be reached

`weather_at_place`(*name*)

Queries the OWM Weather API for the currently observed weather at the specified toponym (eg: "London,uk")

Parameters **`name`** (*`str`*) – the location's toponym

Returns an *Observation* instance or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed or *APICallException* when OWM Weather API can not be reached

`weather_at_places`(*pattern*, *searchtype*, *limit=None*)

Queries the OWM Weather API for the currently observed weather in all the locations whose name is matching the specified text search parameters. A twofold search can be issued: *'accurate'* (exact matching) and *'like'* (matches names that are similar to the supplied pattern).

Parameters

- **`pattern`** (*`str`*) – the string pattern (not a regex) to be searched for the toponym

- **searchtype** – the search mode to be used, must be *'accurate'* for an exact matching or *'like'* for a likelihood matching
- **limit** – the maximum number of *Observation* items in the returned list (default is `None`, which stands for any number of items)
- **limit** – int or `None`

Type searchtype: str

Returns a list of *Observation* objects or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached, *ValueError* when bad value is supplied for the search type or the maximum number of items retrieved

weather_at_places_in_bbox(*lon_left*, *lat_bottom*, *lon_right*, *lat_top*, *zoom=10*, *cluster=False*)

Queries the OWM Weather API for the weather currently observed by meteostations inside the bounding box of latitude/longitude coords.

Parameters

- **lat_top** (*int/float*) – latitude for top margin of bounding box, must be between -90.0 and 90.0
- **lon_left** (*int/float*) – longitude for left margin of bounding box must be between -180.0 and 180.0
- **lat_bottom** (*int/float*) – latitude for the bottom margin of bounding box, must be between -90.0 and 90.0
- **lon_right** (*int/float*) – longitude for the right margin of bounding box, must be between -180.0 and 180.0
- **zoom** (*int*) – zoom level (defaults to: 10)
- **cluster** (*bool*) – use server clustering of points

Returns a list of *Observation* objects or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed, *APICallException* when OWM Weather API can not be reached, *ValueError* when coordinates values are out of bounds or negative values are provided for limit

weather_at_zip_code(*zipcode*, *country*)

Queries the OWM Weather API for the currently observed weather at the specified zip code and country code (eg: 2037, au).

Parameters

- **zip** (*string*) – the location's zip or postcode
- **country** (*string*) – the location's country code

Returns an *Observation* instance or `None` if no weather data is available

Raises *ParseResponseException* when OWM Weather API responses' data cannot be parsed or *APICallException* when OWM Weather API can not be reached

pyowm.weatherapi25.weathercoderegistry module

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

classmethod get_instance()

Factory method returning the default weather code registry :return: 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

Submodules

pyowm.config module

pyowm.constants module

pyowm.owm module

class pyowm.owm.**OWM**(*api_key*, *config=None*)

Bases: `object`

Entry point class providing ad-hoc API clients for each OWM web API.

Parameters

- **api_key** (*str*) – the OWM API key
- **config** (*dict*) – the configuration dictionary (if not provided, a default one will be used)

agro_manager()

Gives a *pyowm.agro10.agro_manager.AgroManager* instance that can be used to read/write data from the Agricultural API. :return: a *pyowm.agro10.agro_manager.AgroManager* instance

airpollution_manager()

Gives a *pyowm.airpollutionapi30.airpollution_manager.AirPollutionManager* instance that can be used to fetch air pollution data. :return: a *pyowm.airpollutionapi30.airpollution_manager.AirPollutionManager* instance

alert_manager()

Gives an *AlertManager* instance that can be used to read/write weather triggers and alerts data. :return: an *AlertManager* instance

city_id_registry()

Gives the *CityIDRegistry* singleton instance that can be used to lookup for city IDs.

Returns a *CityIDRegistry* instance

configuration

Returns the configuration dict for the PyOWM

Returns *dict*

stations_manager()
Gives a *StationsManager* instance that can be used to read/write meteostations data. :returns: a *StationsManager* instance

tile_manager(*layer_name*)
Gives a *pyowm.tiles.tile_manager.TileManager* instance that can be used to fetch tile images. :param layer_name: the layer name for the tiles (values can be looked up on *pyowm.tiles.enums.MapLayerEnum*) :return: a *pyowm.tiles.tile_manager.TileManager* instance

uvindex_manager()
Gives a *pyowm.uvindexapi30.uvindex_manager.UVIndexManager* instance that can be used to fetch UV data. :return: a *pyowm.uvindexapi30.uvindex_manager.UVIndexManager* instance

version
Returns the current version of the PyOWM library

Returns *tuple*

weather_manager()
Gives a *pyowm.weatherapi25.weather_manager.WeatherManager* instance that can be used to fetch air pollution data. :return: a *pyowm.weatherapi25.weather_manager.WeatherManager* instance

Module contents

5.1.3 Description of PyOWM configuration

PyOWM configuration description

PyOWM can be configured at your convenience.

The library comes with a pre-cooked configuration that you can change according to your needs. The configuration is formulated as a Python dictionary.

Default configuration

The default config is the `DEFAULT_CONFIG` dict living in the `pyowm.config` module (check it to know the defaults)

Configuration format

The config dict is formatted as follows:

```
{
    "subscription_type": <pyowm.commons.enums.SubscriptionTypeEnum>,
    "language": <str>,
    "connection": {
        "use_ssl": <bool>
        "verify_ssl_certs": <bool>>,
        "use_proxy": <bool>,
        "timeout_secs": <int>
    },
    "proxies": {
        "http": <str>,
        "https": <str>
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Here are the keys:

- `subscription_type`: this object represents an OWM API Plan subscription. Possible values are: `free|startup|developer|professional|enterprise`
- `language`: 2-char string representing the language you want the weather statuses returned in. Currently serving: `en|ru|ar|zh_cn|ja|es|it|fr|de|pt` and more. Check here for a comprehensive list of supported languages
- `connection`:
  - `use_ssl`: whether to use SSL or not for API calls
  - `verify_ssl_certs`: speaks by itself..
  - `use_proxy`: whether to use a proxy server or not (useful if you're eg. in a corporate network). HTTP and SOCKS5 proxies are allowed
  - `timeout_secs`: after how many seconds the API calls should be timeouted
- `proxies` (this sub-dict is ignored if `use_proxy == False`)
  - `http`: the HTTP URL of the proxy server
  - `https`: the HTTPS/SOCKS5 URL of the proxy server

Providing a custom configuration

You can either pass in your custom dict to the global `OWM` object upon instantiation:

```
from pyowm import OWM
owm = OWM('my-api-key', config=my_custom_config_dict)  # pass in your dict as a named␣
↪argument
```

or you can put your custom configuration inside a JSON text file and have it read by PyOWM:

```
from pyowm.owm import OWM
from pyowm.utils.config import get_config_from
config_dict = get_config_from('/path/to/configfile.json')  # This utility comes in␣
↪handy
owm = OWM('your-free-api-key', config_dict)
```

Be aware that the JSON file must be properly formatted and that the unspecified non-mandatory keys will be filled in with default values. Here is an example:

```
{
    "api_key": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
    "subscription_type": "professional",
    "language": "ru",
    "connection": {
        "use_ssl": true,
        "verify_ssl_certs": true,
        "timeout_secs": 1
    }
}
```

5.1.4 Global PyOWM instantiation documentation

Global PyOWM library usage examples

The PyOWM library has one main entry point: the `OWM` class. You just need to instantiate it to get started!

Please refer to the `Code Recipes` page, section: `Library initialization`, to get info about how to instantiate the PyOWM library

Dumping PyOWM objects to Python dictionaries

PyOWM object instances (eg. `Weather` or `Location` objects) can be dumped to `dicts`:

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
weather = mgr.weather_at_place('London,GB').weather  # get the weather at London,GB␣
↪now
dump_dict = weather.to_dict()
```

This is useful as you can save the dump dictionaries to files (eg. using Pyhon `json` or `pickle` modules)

Printing objects

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

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
print(owm)   # <pyowm.weatherapi25.owm25.OWM25 - API key=*******i-key, subscription␣
↪type=free, PyOWM version=3.0.0>
```

5.1.5 City ID registry documentation

City ID Registry usage examples

Using city IDS instead of toponyms or geographic coordinates is the preferred way of querying the OWM weather API

You can obtain the city ID for your toponyms/geoocoords of interest via the `City ID Registry`.

Please refer to the `Code Recipes` page, section: `Identifying cities and places via city IDs`, to get info about it

5.1.6 Weather API examples

Weather API usage examples

The OWM Weather API gives you data about currently observed and forecast weather data upon cities in the world.

Please refer to the `Code Recipes` page, sections: `Weather data`, `Weather forecasts` and `Meteostation historic measurements` to know more.

5.1.7 Agro API examples

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]]])

# 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
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
```

(continues on next page)

(continued from previous page)

```
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'

```

(continues on next page)

(continued from previous page)

```
# 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.

5.1.8 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

Querying UV index observations

Getting the data is easy:

```
from pyowm import OWM
owm = OWM('apikey')
mgr = owm.uvindex_manager()
uvi = mgr.uvindex_around_coords(lat, lon)
```

The query returns an UV Index value entity instance

Querying UV index forecasts

As easy as:

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

Querying UV index history

As easy as:

```
uvi_history_list = mgr.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.

`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.

5.1.9 Air Pollution API examples

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:

```
from pyowm import OWM
owm = OWM('apikey')

# 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

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

5.1.10 Stations API examples

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
```

(continues on next page)

(continued from previous page)

```
# 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()


# 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()
```

(continues on next page)

(continued from previous page)

```
my_custom_be.persist_buffer(buf)
```

5.1.11 Alerts API examples

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.

```
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('apikey')
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,
```

(continues on next page)

(continued from previous page)

```
                        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 --

# 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`:

```
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

5.1.12 Map tiles client examples

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
```

5.1.13 PyOWM Exceptions

Exceptions

PyOWM uses custom exception classes. Here you can learn which classes are used and when such exceptions are cast by the library

Exceptions Hierarchy

```
Exception
|
|___PyOWMError
    |
    |___ConfigurationError
    |    |
    |    |__ConfigurationNotFoundError
    |    |__ConfigurationParseError
    |
    |___APIRequestError
    |    |
    |    |__BadGatewayError
    |    |__TimeoutError
    |    |__InvalidSSLCertificateError
    |
    |___APIResponseError
```

(continues on next page)

(continued from previous page)

```
          |
          |__NotFoundError
          |__UnauthorizedError
          |__ParseAPIResponseError
```

Exception root causes

- `PyOWMError` is the base class. Never raised directly
- `ConfigurationError` parent class for configuration-related exceptions. Never raised directly
- `ConfigurationNotFoundError` raised when trying to load configuration from a non-existent file
- `ConfigurationParseError` raised when configuration can be loaded from the file but is in a wrong, unparsable format
- `APIRequestError` base class for network/infrastructural issues when invoking OWM APIs
- `BadGatewayError` raised when upstream OWM API backends suffer communication issues.
- `TimeoutError` raised when calls to the API suffer timeout due to slow response times upstream
- `InvalidSSLCertificateError` raised when it is impossible to verify the SSL certificates provided by the OWM APIs
- `APIResponseError` base class for non-ok API responses from OWM APIs
- `NotFoundError` raised when the user tries to access resources that do not exist on the OWM APIs
- `UnauthorizedError` raised when the user tries to access resources she is not authorized to access (eg. you need a paid API subscription)
- `ParseAPIResponseError` raised upon impossibility to parse the JSON payload of API responses

5.1.14 Utility functions examples

PyOWM utility functions usage example

PyOWM provides a few packages that contain utility functions.

Some of them are specifically designed to be used by the core PyOWM classes but others you can use to make your life easier when operating PyOWM!

All utility modules live inside the `pyowm.utils` package

Here are most useful modules:

- `config`: handling of PyOWM configuration
- `formatting`: formatting of timestamp entities (Python native types, UNIX epochs and ISO-8601 strings)
- `geo`: handling of geographic entities such as points, polygons and their geoJSON representation
- `measureables`: conversions among physical units (eg. temperature, wind)
- `timestamps`: human friendly timestamps generation

`config` module

```
from pyowm.utils.config import get_default_config, get_default_config_for_
↪subscription_type, \
    get_default_config_for_proxy, get_config_from

config_dict = get_default_config()                                      # loads the␣
↪default config dict
config_dict = get_default_config_for_subscription_type('professional')    # loads the␣
↪config dict for the specified subscription type
config_dict = get_config_from('/path/to/configfile.json')               # loads the␣
↪config dict from the  specified JSON file
config_dict = get_default_config_for_proxy('http_url', 'https_url')      # loads the␣
↪config dict to be used behind a proxy whose URLs are specified
```

`formatting` module

```
from datetime import datetime as dt
from pyowm.utils import formatting

unix_value = formatting.timeformat(dt.today(), 'unix')         # from datetime to UNIX
iso_str_value = formatting.timeformat(dt.today(), 'iso')       # from datetime to ISO-
↪8601 string
datetime_value = formatting.timeformat(1590100263, 'date')     # from UNIX to datetime
iso_str_value = formatting.timeformat(1590100263, 'iso')       # from UNIX to ISO-8601␣
↪string
datetime_value = formatting.timeformat('2020-05-21 22:31:03+00', 'date') # from ISO-
↪8601 string to datetime
unix_value = formatting.timeformat('2020-05-21 22:31:03+00', 'unix') # from ISO-8601␣
↪string to UNIX
```

`geo` module

The module provides classes to represent geometry features:

- `Point`
- `Multipoint` (aka point set)
- `Polygon`
- `Multipolygon` (aka polygon set)

Geometry features are used eg. in OWM Alert API to provide geographical boundaries for alert setting.

PyOWM uses standard geometry types defined by the GeoJSON Format Specification - RFC 7946 data interchange format.

Common geometry methods

All geometry types can be dumped to a GeoJSON string and to a Python `dict`

```
from pyowm.utils import geo
point = geo.Point(20.8, 30.9)
point.geojson()  # '{"type": "Point", "coordinates": [20.8, 30.9]}'
point.to_dict()  #  {'type': 'Point', 'coordinates': [20.8, 30.9]}
```

All geometry types also feature a static factory method: you provide the dictionary and the factory returns the object instance

```
from pyowm.utils.geo import Point
point_dict = {'type': 'Point', 'coordinates': [20.8, 30.9]}
point = Point.from_dict(point_dict)
```

Please refer to the GeoJSON specification about how to properly format the dictionaries to be given the factory methods

`Point` class

A point is a couple of geographic coordinates: longitude and latitude

```
from pyowm.utils import geo
lon = 20.8
lat = 30.9
point = geo.Point(lon, lat)
coords = point.lon, point.lat  # 20.8, 30.9
```

As circle shapes are not part of the GeoJSON specification, you can approximate the circle having a specific `Point` instance at its center with a square polygon: we call it bounding square polygon. You just need to provide the radius of the circle you want to approximate (in kms):

```
from pyowm.utils import geo
point = geo.Point(20.8, 30.9)
polygon = point.bounding_square_polygon(inscribed_circle_radius_km=2.0)   # default␣
↪radius: 10 km
```

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.

From City IDs to `Point` objects

The City ID Registry class can return the geopoints that correspond to one or more named cities:

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

This, in combination with the `bounding_square_polygon` method, makes it possible to easily get polygons to cover large squared areas centered on largely spread city areas - such as London,GB itself:

```
london_city_centre = geopoints[0]
london_city_bounding_polygon = london_city_centre.bounding_square_polygon(inscribed_
↪circle_radius_km=12)
```

`MultiPoint` class

A `MultiPoint` object represent a set of `Point` objects

```
from pyowm.utils.geo import Point, MultiPoint
point_1 = Point(20.8, 30.9)
point_2 = Point(1.2, 0.4)

# many ways to instantiate
multipoint = MultiPoint.from_points([point_1, point_2])
multipoint = MultiPoint([20.8, 30.9], [1.2, 0.4])

multipoint.longitudes  # [20.8, 1.2]
multipoint.latitudes   # [30.9, 0.4]
```

`Polygon` class

A `Polygon` object represents a shape made by a set of geopoints, connected by lines. Polygons are allowed to have "holes". Each line of a polygon must be closed upon itself: this means that the last geopoint defined for the line *must* coincide with its first one.

```
from pyowm.utils.geo import Polygon, Point
point_1_coords = [2.3, 57.32]
point_2_coords = [23.19, -20.2]
point_3_coords = [-120.4, 19.15]
point_1 = Point(point_1_coords)
point_2 = Point(point_2_coords)
point_3 = Point(point_3_coords)


# many ways to instantiate
line = [point_1_coords, point_2_coords, point_3_coords , point_1_coords]  # last␣
↪point equals the first point
polygon = Polygon([line])
line = [point_1, point_2, point_3, point_1]  # last point equals the first point
polygon = Polygon.from_points([line])
```

`MultiPolygon` class

A `MultiPolygon` object represent a set of `Polygon` objects Same philosophy here as for `MultiPoint` class, polygons can cross:

```
from pyowm.utils.geo import Point, Polygon, MultiPolygon
point_1 = Point(20.8, 30.9)
point_2 = Point(1.2, 0.4)
point_3 = Point(49.9, 17.4)
point_4 = Point(178.4, 78.3)
polygon_1 = Polygon.from_points([point_1, point_2, point_3, point_1])
polygon_2 = Polygon.from_points([point_3, point_4, point_2, point_3])

multipoint = MultiPolygon.from_polygons([polygon_1, polygon_2])
```

Building geometries

There is a useful factory method for geometry types, which you can use to turn a geoJSON-formatted dictionary into the corresponding topology type:

```
from pyowm.utils.geo import GeometryBuilder
point_dict = {
    "type": "Point",
    "coordinates": [53, 37]
}
point = GeometryBuilder.build(point_dict)      # this is a `Point` instance

wrongly_formatted_dict = {"a": 1, "b": 99}
GeometryBuilder.build(wrongly_formatted_dict)  # you get an exception
```

`measurables` module

This module provides utilities numeric conversions

Temperature

You have a `dict` whose values represent temperature units in Kelvin: you can convert them to Fahrenheit and Celsius.

```
from pyowm.utils import measurables

fahrenheit_temperature_dict = measurables.kelvin_dict_to(kelvin_temperature_dict,
↪'fahrenheit')
celsius_temperature_dict = measurables.kelvin_dict_to(kelvin_temperature_dict,
↪'celsius')
```

Wind

On the same line as temperatures, you can convert wind values among meters/sec, kilometers/hour, miles/hour, knots and the Beaufort scale. The pivot unit of measure for wind is meters/sec

```
from pyowm.utils import measurables

kmhour_wind_dict = measurables.metric_wind_dict_to_km_h(msec_wind_dict)
mileshour_wind_dict = measurables.metric_wind_dict_to_imperial(msec_wind_dict)
knots_wind_dict = measurables.metric_wind_dict_to_knots(msec_wind_dict)
beaufort_wind_dict = measurables.metric_wind_dict_to_beaufort(msec_wind_dict)
```

`timestamps` module

```
from pyowm.utils import timestamps

timestamps.now()                                         # Current time in `datetime.
↪datetime` object (default)
timestamps.now('unix')                                   # epoch
timestamps.now('iso')                                    # ISO8601-formatted str␣
↪(YYYY-MM-DD HH:MM:SS+00)
```

(continues on next page)

(continued from previous page)

```
timestamps.tomorrow()                                    # Tomorrow at this time
timestamps.tomorrow(18, 7)                               # Tomorrow at 6:07 PM
timestamps.yesterday(11, 27)                             # Yesterday at 11:27 AM

timestamps.next_three_hours()                            # 3 hours from now
timestamps.last_three_hours(date=timestamps.tomorrow())  # tomorrow but 3 hours␣
↪before this time

timestamps.next_hour()
timestamps.last_hour(date=timestamps.yesterday())        # yesterday but 1 hour␣
↪before this time


# And so on with: next_week/last_week/next_month/last_month/next_year/last_year
```

5.2 Legacy PyOWM v2 documentation

Please refer to historical archives on Readthedocs or the GitHub repo for this

CHAPTER 6

Installation

6.1 pip

The easiest method of all:

```
$ pip install pyowm
```

If you already have PyOWM 2.x installed and want to upgrade to safely update it to the latest 2.x release just run:

```
$ pip install --upgrade pyowm>=2.0,<3.0
```

6.2 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!

6.3 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

6.4 Distribution packages

On Windows you have EXE installers

On ArchLinux you can use 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)
```

On OpenSuse you can use with YaST/Zypper package manager, run:

```
zypper install python-pyowm
```

CHAPTER 7

How to contribute

There are multiple ways to contribute to the PyOWM project! Find the one that suits you best

7.1 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

7.2 Reporting a PyOWM bug

That's simple: what you need to do is just open a new issue on GitHub.

7.2.1 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"

7.2.2 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

7.3 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!

7.4 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

7.4.1 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.

7.4.2 Guidelines for code branching

Simple ones:

- the “develop” branch contains work-in-progress code
- 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!!!**

7.4.3 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

7.5 Contributing on PyOWM bundling/distributing

Please open a GitHub issue and get in touch to discuss your idea!

CHAPTER 8

PyOWM Community

Find us on Slack !

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index