pyowm Documentation

Claudio Sparpaglione

Jun 24, 2023

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
- **Geocoding API v1.0** allowing to perform direct/reverse geocoding

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

2.1 Very important news

OpenWeatherMap API recently "blocked" calls towards a few legacy API endpoints whenever requested by **clients using non-recent free API keys.**

This means that if you use PyOWM methods such as the ones for getting observed or forecasted weather, PyOWM might return authorization errors This behaviour is not showing if you use API keys issued a long time ago.

The proper way to obtain such data is to call the "OneCall" methods using your API key

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

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:

Very important news OpenWeatherMap API recently "blocked" calls towards a few legacy API endpoints whenever requested by **clients using non-recent free API keys.**

This means that if you use PyOWM methods such as the ones for getting observed or forecasted weather, PyOWM might return authorization errors This behaviour is not showing if you use API keys issued a long time ago.

The *proper way to obtain such data is to call the "OneCall" methods using your API key*

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
```

(continues on next page)

(continued from previous page)

```
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

The OWM API can be asked to return localized *detailed statuses* for weather data **In PyOWM this means that you can specify a language and you'll retrieve `Weather` objects having the `detailed_status` field localized in that language. Localization is not provided for `status` field instead, so pay attention to that.**

The list of supported languages is given by:

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

Check out https://openweathermap.org/current for reference on supported languages

English is the default language on the OWM API - but you can change it:

```
from pyowm.owm import OWM
from pyowm.utils.config import get_default_config
config_dict = get_default_config()
config_dict['language'] = 'fr'  # your language here, eg. French
owm = OWM('your-api-key', config_dict)
mgr = owm.weather_manager()
observation = mgr.weather_at_place('Paris, FR')
observation.weather.detailed_status  # Nuageux
```

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

You can easily get the City ID of a known toponym, as well as its geographic coordinates Also you can leverage direct/reverse geocoding

City IDs

The following calls will not result in any OWM API call in the background, so they will only happen locally to your machine.

Obtain the city ID registry

As easy as:

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

Get the ID of a city given its name

Once you've got it, use the city ID registry to lookup the ID of a city given its name:

```
list_of_tuples = london = reg.ids_for('London', matching='like')
print(list_of_tuples)      # This will give something like:
                           # [ (2643743, 'London', 'GB', None, 51.50853, -0.12574),
                           #   (1006984, 'East London', 'ZA', None, -33.015289, 27.
↪911619),
                           #   (1644003, 'Glondong', 'ID', None, -6.7924, 111.891602),
                           #   ... ]
```

This call searches for all the places that contain the string `'London'` in their names, in any part of the world. This is because the search matching criterion we've used is `like` (this is the default one, if you don't specify it)

The other available matching criterion is `exact`, which retrieves all places having exactly `'London'` as their name, in any part of the world (be careful that this operation is case-sensitive !)

Let's try to search for the same city with an exact match:

```
list_of_tuples = london = reg.ids_for('London', matching='exact')
print(list_of_tuples)      # This will give something like:
                           # [ (2643743, 'London', 'GB', None, 51.50853, -0.12574),
                           #   (4119617, 'London', 'US', 'AR', 35.328972, -93.25296),
                           #   (4298960, 'London', 'US', 'KY', 37.128979, -84.08326)
                           #   ... ]
```

As you can see, all results are exactly named `'London'`.

All the above searches give you back a list of tuples: each tuple is in the format `(city_id, name, country, state, lat, lon)` (fields as self-explanatory).

City disambiguation

As you might have guessed, there is a high probability that your city is not unique in the world, and multiple cities with the same name exist in other countries Therefore: whenever you search for a specific city in a specific country then also pass in the 2-letter country name and - even further - also specify a 2-letter state name if you're searching for places in the United States.

Eg: if you search for the British `London` you'll get multiple results. You then should also specify the country (`GB`) in order 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 = reg.ids_for('London', matching='exact')                    # lots of␣
↪results, spread all over the world
list_of_tuples = reg.ids_for('London', country='GB', matching='exact')    # only one:␣
↪[(2643743, 'London', 'GB', None, 51.50853, -0.12574)]
london_gb = list_of_tuples[0]
id_of_lonfon_gb = london_gb[0]                                              # ID of␣
↪London, GB
```

Whenever searching cities in the US, you'd better also specify the relevant US-state. For instance, `'Ontario'` is a city in Canada and multiple aliases exist in different US-states:

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

# All Ontario cities in the uS
ontarios_in_us = reg.ids_for('Ontario', country='US', matching='exact')  # five␣
↪results

# Ontario in Canade
ontario_in_canada = reg.ids_for('Ontario', country='CA', matching='exact')  # one␣
↪result: [(6093943, 'Ontario', 'CA', None, 49.250141, -84.499832)]

# Ontario in the state of New York
ontario_in_ny = reg.ids_for('Ontario', country='US', state='NY', matching='exact')  #␣
↪one result: [(5129887, 'Ontario', 'US', 'NY', 43.220901, -77.283043)]
```

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 Tokyo (Japan):

```
from pyowm.owm import OWM
owm = OWM('your-api-key')
reg = owm.city_id_registry()
list_of_locations = reg.locations_for('Tokyo', country='JP', matching='exact')
tokyo = list_of_locations[0]
lat = tokyo.lat   # 35.689499
lon = tokyo.lon   # 139.691711
```

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')
```

(continues on next page)

(continued from previous page)

```
reg = owm.city_id_registry()
list_of_geopoints = reg.geopoints_for('Rome', matching='exact')
```

Direct/reverse geocoding

Simply put:

- DIRECT GEOCODING: from toponym to geocoords
- REVERSE GEOCODING: from geocoords to toponyms

Both geocoding actions are performed via a `geocoding_manager` object and will require an actual call to be made to the OWM API: so please bear that in mind because that will count against your amount of allowed API calls

Direct geocoding of a toponym

The call is very similar to `ids_for` and `locations_for`.

You at least need to specify the toponym name and country ISO code (eg. `GB`, `IT`, `JP`, ...), while if the input toponym is in the United States you should also specify the `state_code` parameter

The call returns a list of `Location` object instances (in case of no ambiguity, only one item in the list will be returned) You can then get the lat/lon from the object instances themselves

Results can be limited with the `limit` parameter

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

# geocode London (no country specified) - we'll get many results
list_of_locations = mgr.geocode('London')
a_london = list_of_locations[0]  # taking the first London in the list
a_london.lat
a_london.lon

# geocode London (Great Britain) - we'll get up to three Londons that exist in GB
list_of_locations = mgr.geocode('London', country='GB', limit=3)

# geocode London (Ohio, United States of America): we'll get all the Londons in Ohio
list_of_locations = mgr.geocode('London', country='US', state_code='OH')
```

Reverse geocoding of geocoordinates

With reverse geocoding you input a lat/lon float couple and retrieve a list all the `Location` objects associated with these coordinates.

Results can be limited with the `limit` parameter

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

# London
```

(continues on next page)

(continued from previous page)

```
lat = 51.5098
lon = -0.1180

# reverse geocode London
list_of_locations = mgr.reverse_geocode(lat, lon)  # list contains: City of London,␣
↪Islington, Lewisham, ...
```

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 "photograph" observed and forecasted weather 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 (°C) 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
```

Requesting only part of the available OneCall data, in imperial units

```
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, exclude='minutely,hourly', units=
↪'imperial')

# in this exacmple, the data in the one_call object will be in imperial units
# possible units are defined by the One Call API, here: https://openweathermap.org/
↪api/one-call-api
# as of 2020.08.07 available values are: 'metric' or 'imperial'
# the various units for the different options are shown here: https://openweathermap.
↪org/weather-data
one_call.current.temperature() # Eg.: 74.07 (deg F)

# the example above does not retrieve minutely or hourly data, so it will not be␣
↪available in the one_call object
# available exclude options are defined by the One Call API
# BUT using 'current' will error, as the pyowm one_call requires it
# as of 2020.08.07 available values are: 'minutely', 'hourly', 'daily'
# multiple exclusions may be combined with a comma, as above
one_call.forecast_hourly # empty because it was excluded from the request
```

Checking available National Weather Alerts for a location

Many countries have early warning systems in place to notify about upcoming severe weather events/conditions. Each alert has a title, a description, start/end timestamps and is tagged with labels. You can check if any national alert has been issued for a specific location this way:

```
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)
national_weather_alerts = one_call. national_weather_alerts

for alert in national_weather_alerts:
    alert.sender                      # issuing national authority
    alert.title                       # brief description
    alert.description                 # long description
    alert.start_time()                # start time in UNIX epoch
    alert.end_time(timeformat='ISO')  # end time in ISO format
```

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
```

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']
```

(continues on next page)

(continued from previous page)

```
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').weather.rain
rain_dict['1h']
rain_dict['3h']
```

Get current pressure on a location

Pressure is similar to rain, you get a dict with hPa values and these keys: `press` (atmospheric pressure on the ground, sea level if no sea level or ground level data) `sea_level` (on the sea level, if location is on the sea) and `grnd_level`. Note that `press` used below refers to the dict value in

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

Pressure values are given in the metric hPa, or hectopascals (1 hPa is equivalent to 100 pascals). You can easily convert these values to inches of mercury, or inHg, which is a unit commonly used in the United States. Similar to above, we can do:

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

# the default unit is hPa
pressure_dict_unspecified = obs.weather.barometric_pressure()
pressure_dict_in_hg = obs.weather.barometric_pressure(unit='inHg')
```

Get current visibility distance on a location

You might want to know how clearly you can see objects in Berlin. This is the visibility distance, an average distance taken from an Observation object and given in meters. You can also convert this value to kilometers or miles.

```
from pyowm.owm import OWM

owm = OWM('your-api-key')
mgr = owm.weather_manager()
obs = mgr.weather_at_place('Berlin,DE')

# the default value provided by our call (in meters)
visibility = obs.weather.visibility_distance

# kilometers is the default conversion unit
visibility_in_kms = obs.weather.visibility()
visibility_in_miles = obs.weather.visibility(unit='miles')
```

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+00:00`) 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')
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 = mgr.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 = 2643743 #London
weather = mgr.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 = [2643743 , 4517009, 5056033]
list_of_observations = mgr.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
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)

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

Weather forecasts

>>>IMPORTANT NOTE<<<: OpenWeatherMap has deprecated legacy weather forecasts endpoints, therefore you could get errors if you invoke them The recommended way to get weather forecasts is now the *OneCall* API

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
three_h_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+00:00'
forecaster.when_ends('iso')                               # 2020-03-16 14:00:00+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()
three_h_forecaster = mgr.forecast_at_place('Berlin,DE', '3h')
tomorrow_at_five = timestamps.tomorrow(17, 0)                          # datetime object␣
↪for tomorrow at 5 PM
weather = three_h_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.utils import timestamps
from pyowm.owm import OWM
owm = OWM('your-api-key')
mgr = owm.weather_manager()
three_h_forecaster = mgr.forecast_at_place('Berlin,DE', '3h')

# Is it going to rain tomorrow?
tomorrow = timestamps.tomorrow()                    # datetime object for tomorrow
three_h_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()
three_h_forecaster = mgr.forecast_at_place('Berlin,DE', '3h')

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

# Is it going to be foggy in the next 5 days ?
three_h_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

Air pollution data

Instead of getting a `weather_manager`, get from the main OWM object a `airpollution_manager` and use it

Getting air pollution concentrations and Air Quality Index on geographic coords

Air polluting agents concentration can be queried in one shot:

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

air_status = mgr.air_quality_at_coords(51.507351, -0.127758)  # London, GB

# you can then get values for all of these air pollutants
air_status.co
air_status.no
air_status.no2
air_status.o3
air_status.so2
air_status.pm2_5
air_status.pm10
```

(continues on next page)

(continued from previous page)

```
air_status.nh3

# and for air quality index
air_status.aqi
```

Getting forecasts for air pollution on geographic coords

We can get also get forecasts for air pollution agents concentration and air quality index:

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

list_of_forecasts = mgr.air_quality_forecast_at_coords(51.507351, -0.127758)  #␣
↪London, GB

# Each item in the list_of_forecasts is an AirStatus object
for air_status in list_of_forecasts:
    air_status.co
    air_status.no
    air_status.no2
    air_status.o3
    air_status.so2
    air_status.pm2_5
    air_status.pm10
    air_status.nh3
    air_status.aqi  # air quality index
```

Getting historical air pollution data on geographic coords

We can get also get historical values for air pollution agents concentration and air quality index:

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

# fetch history from a certain point in time up to now...
start = 1606223802  # November 24, 2020
list_of_historical_values = mgr.air_quality_history_at_coords(51.507351, -0.127758,␣
↪start)  # London, GB

# ...or fetch history on a closed timeframe in the past
end = 1613864065  # February 20, 2021
list_of_historical_values = mgr.air_quality_history_at_coords(51.507351, -0.127758,␣
↪start, end=end)  # London, GB

# Each item in the list_of_historical_values is an AirStatus object
for air_status in list_of_historical_values:
    air_status.co
    air_status.no
    air_status.no2
    air_status.o3
    air_status.so2
```

(continues on next page)

(continued from previous page)

```
    air_status.pm2_5
    air_status.pm10
    air_status.nh3
    air_status.aqi  # air quality index
```

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
```

(continues on next page)

(continued from previous page)

```
historian.humidity_series()


# 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
  }
  # [...]
}
```

5.1.2 FAQ about common errors

Frequently Asked Questions

Common fixes to common errors reported by the Community

AttributeError: 'OWM25' object has no attribute 'xxx'

Your code looks like:

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

AttributeError: 'OWM25' object has no attribute 'weather_manager'
```

This happens because **you are not running PyOWM v3** and this is because your code is currently **based on an old Python 2 setup** Python 2 is officially dead and should be removed in favor of Python 3.

What you should do is:

- install Python 3.6+
- install PyOWM v3+ with `pip3 install pyowm`

The above snippet should just work fine then.

Remember to port the rest of your code to Python 3: everything related to PyOWM v2 can be ported using this guide

UnauthorizedError: Invalid API Key provided

You are able to successfully create an `OWM` object and calling functions **other than One-Call** related ones (eg. getting observed or forecasted weather)

As stated in the documentation home page, OpenWeatherMap API recently "blocked" calls towards a few legacy API endpoints whenever requested by **clients using non-recent free API keys.**

This means that PyOWM might return authorization errors in that case.

This behaviour is not showing if you use API keys issued time ago - unfortunately I have no way to be more precise as OWM never stated this officially.

The proper way to obtain the data you are looking for is to call the "OneCall" PyOWM methods using your API key

So please refer to the documentation for this

I cannot use PyOWM 3 so I need to use PyOWM version 2.10

This may happen if you still use Python 2 or you use Python 3 but with a minor version that is not supported by PyOWM

Please install PyOWM 2.10 with:

```
pip2 install pyowm==2.10
```

And find the PyOWM 2.10 documentation here

ModuleNotFound error upon installing PyOWM development branch from Github

Installation of the (potentially unstable) development trunk used to be like this:

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

You would get something like:

```
Collecting git+https://github.com/csparpa/pyowm.git@develop
  Cloning https://github.com/csparpa/pyowm.git (to revision develop) to /tmp/pip-req-
↪build-_86bl7ty
    [......]
    ERROR: Command errored out with exit status 1:
     command: /home/me/.local/share/virtualenvs/backend-nPPHZqlJ/bin/python -c
↪'import sys, setuptools, tokenize; sys.argv[0] = '"'"'/tmp/pip-req-build-_86bl7ty/
↪setup.py'"'"'; __file__='"'"'/tmp/pip-req-build-_86bl7ty/setup.py'"'"';
↪f=getattr(tokenize, '"'"'open'"'"', open)(__file__);code=f.read().replace('"'"'\r\n'
↪"'"', '"'"'\n'"'"');f.close();exec(compile(code, __file__, '"'"'exec'"'"'))' egg_
↪info --egg-base /tmp/pip-pip-egg-info-ww_gs9y3
         cwd: /tmp/pip-req-build-_86bl7ty/
    Complete output (17 lines):
    Traceback (most recent call last):
      [......]
      File "/tmp/pip-req-build-_86bl7ty/pyowm/commons/tile.py", line 6, in <module>
        from pyowm.utils.geo import Polygon
      File "/tmp/pip-req-build-_86bl7ty/pyowm/utils/geo.py", line 4, in <module>
        import geojson
    ModuleNotFoundError: No module named 'geojson'
    ----------------------------------------
ERROR: Command errored out with exit status 1: python setup.py egg_info Check the␣
↪logs for full command output.
```

I've realized this way of installing is bad *as it does not install PyOWM's dependencies** along. Therefore the right way to go is:

```
$ git clone https://github.com/csparpa/pyowm.git
$ cd pyowm && git checkout develop
$ pip install -r requirements.txt && python setup.py install
```

5.1.3 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

pyowm.agroapi10.enums module

Module contents

5.1.4 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>,
        "max_retries": <int>|<None>
    },
    "proxies": {
        "http": <str>,
        "https": <str>
    }
}
```

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

  - `max_retries`: how many times PyOWM should retry to call the API if it responds with an error or timeouts. Defaults to `None`, which means: call forever.

- `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
    }
}
```

5.1.5 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 Python `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>
```

5.1.6 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/geocoords 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

5.1.7 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.

5.1.8 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 acres, 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)
```

(continues on next page)

(continued from previous page)

```
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!

Overview

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
```

(continues on next page)

(continued from previous page)

```
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
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+00:00'

# Get its palette
bnw_sat_image.palette               #  '2'

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

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

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

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

Querying for NDVI and EVI image stats

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

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

```
ndvi_metaimage   # metaimage for a NDVI image

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

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

Stats dictionaries contain:

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

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

Supported satellites

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

```
from pyowm.agroapi10.enums import SatelliteEnum

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

Currently only Landsat 8 and Sentinel 2 satellite imagery is available

Supported presets

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

```
from pyowm.agroapi10.enums import PresetEnum

PresetEnum.TRUE_COLOR  # 'truecolor'
```

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

Supported image types

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

```
from pyowm.agroapi10.enums import ImageTypeEnum

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

Currently only PNG and GEOTIFF imagery is available

Supported color palettes

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

```
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.

5.1.9 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, 0, timezone.utc),
    end=datetime.datetime(2018, 2, 15, 0, 0, 0, timezone.utc))
```

`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.

5.1.10 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
from pyowm.utils import timestamps

owm = OWM('apikey')

# get an air pollution manager object
mgr = owm.airpollution_manager()

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

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

# Get available CO Index in the last ...
coi = mgr.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 - with 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:

```
from pyowm import OWM
from pyowm.utils import timestamps

owm = OWM('apikey')

# get an air pollution manager object
mgr = owm.airpollution_manager()

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

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

# Get available O3 value in the last ...
oz = mgr.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 - with 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 data - please refer to that bit of the docs

5.1.11 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` contains 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
```

(continues on next page)

(continued from previous page)

```
mgr.send_measurement(raw_measurement_obj)

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

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

So when querying for measurements, you need to specify:

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

Example:

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

Buffers

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

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

Examples:

```
from pyowm.stationsapi30.buffer import Buffer

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

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

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

(continues on next page)

(continued from previous page)

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

5.1.12 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,
                        80) # clouds % coverage

# -- triggers --

# create a trigger
```

(continues on next page)

(continued from previous page)

```
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 measurement 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 respect to the time when the trigger will be created on the Alert API*, how many milliseconds after should it begin to be checked for conditions matching
- end_after_millis: *with respect to the time when the trigger will be created 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

5.1.13 Geocoding API examples

Geocoding API usage examples

The OWM Weather API gives you the possibility to perform direct and reverse geocoding:

- DIRECT GEOCODING: from toponym to geocoords
- REVERSE GEOCODING: from geocoords to toponyms

Please refer to the `Code Recipes` page, sections: `Direct/reverse geocoding`

5.1.14 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...
```

(continues on next page)

(continued from previous page)

```
# 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
```

5.1.15 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
        |
        |__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

5.1.16 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+00:00', 'date') # from␣
↪ISO-8601 string to datetime
unix_value = formatting.timeformat('2020-05-21 22:31:03+00:00', '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)
```

(continues on next page)

(continued from previous page)

```
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)
```

Pressure

OWM gives barometric pressure in hPa values, in a dict of three pressure items. You can convert these to inHg, which is a common unit of measurement in the United States.

```
from pyowm.utils import measurables

hpa_pressure_dict = {'press': 1000, 'sea_level': 1000, 'grnd_level': 1000}
inhg_pressure_dict = measurables.metric_pressure_dict_to_inhg(hpa_pressure_dict)
```

Visibility

A typical API response contains a single visibility distance value. This is described as the average visibility in meters. You can convert this value (from meters) using the function provided to either kms or miles.

```
from pyowm.utils import measurables

visibility = 1000

# the default return value is in kilometers
visibility_kms = measurables.visibility_distance_to(visibility)
visibility_miles = measurables.visibility_distance_to(visibility, 'miles')
```

`timestamps` module

All `datetime.datetime` objects returned by PyOWM are UTC offset-aware

```
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+00:00)

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 latest development version

You can install the development trunk with _pip_:

```
git clone https://github.com/csparpa/pyowm.git
cd pyowm && git checkout develop
pip install -r requirements.txt  # install dependencies
python setup.py install          # install develop branch code
```

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 and 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, coding, 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 succinct 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 web application) 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 advised 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