plantpredict-python Documentation

Release 1.0.5

Stephen Kaplan

Oct 02, 2023

Contents

PlantPredict is a web-based, utility-scale energy prediction software package. This Python software development kit provides a library to access the full functionality of PlantPredict via its API.

Full documentation on the backend algorithms used in PlantPredict is available here.

The source code for plantpredict-python is available on GitHub.

CHAPTER 1

Contents

1.1 Installation & Setup

This SDK is currently compatible with Python 3.6/3.7 and backwards-compatible with Python 2.7. However, future versions may lose Python 2.7 compatibility due to official end of support of the Python 2 language on January 1, 2020. There are a variety of ways to set up a Python 3 environment and install this library. For the sake of simplicity, a generalized "basic" installation guide and a guide for users of the Anaconda Distribution are provided. The recommended setup for all users (including those new to Python/coding) is that of the Anaconda distribution, as it is more prevalent in the scientific and engineering community.

1.1.1 Setup Guide Using the Anaconda Distribution (Recommended)

The Anaconda Distribution is recommend if you are a scientist, engineer, researcher, or student. It comes bundled with many useful Python scientific/numerical libraries, a GUI for managing the libraries, and several open-source software development tools. Most importantly, just like the standard distribution of Python, it is free and open-source.

1. Install the latest version of the Anaconda Distribution, if not already installed.
2. (Optional, but recommended). Open the "Anaconda Prompt" terminal that comes with the Anaconda distribution, navigate to your project's directory and follow instructions for creating a conda environment and activating a conda environment.
3. Install the `plantpredict` package to your environment by typing the command `pip install plantpredict` into the terminal. (Note: `plantpredict` is not yet available via `conda install`/the Anaconda Navigator GUI, but will be added to conda-forge in future versions).
4. Follow the steps in *API Authentication* to obtain API credentials and authenticate with the server.
5. Use the tutorials in *Example Usage* as a starting point for your own scripting and analysis. Detailed documentation on each class and method can be found in *SDK Reference*.

1.1.2 Basic Installation

1. Install the latest version of Python, if not already installed.
2. (Optional, but recommended) Create a virtual environment. Open a terminal/command prompt, navigate to your new project's directory, and follow the instructions for installing and activating a virtualenv.
3. Install `plantpredict` via pip by typing the command `pip install plantpredict` into the terminal.
4. Follow the steps in *API Authentication* to obtain API credentials and authenticate with the server.
5. Use the tutorials in *Example Usage* as a starting point for your own scripting and analysis. Detailed documentation on each class and method can be found in *SDK Reference*.

1.2 API Authentication

PlantPredict uses the Amazon Cognito & OAuth 2.0 API for administering and managing access tokens. If you are a first time user of the PlantPredict API, you need a set of client credentials (client ID, and client secret).

1.2.1 Step 1: Generate/receive client credentials.

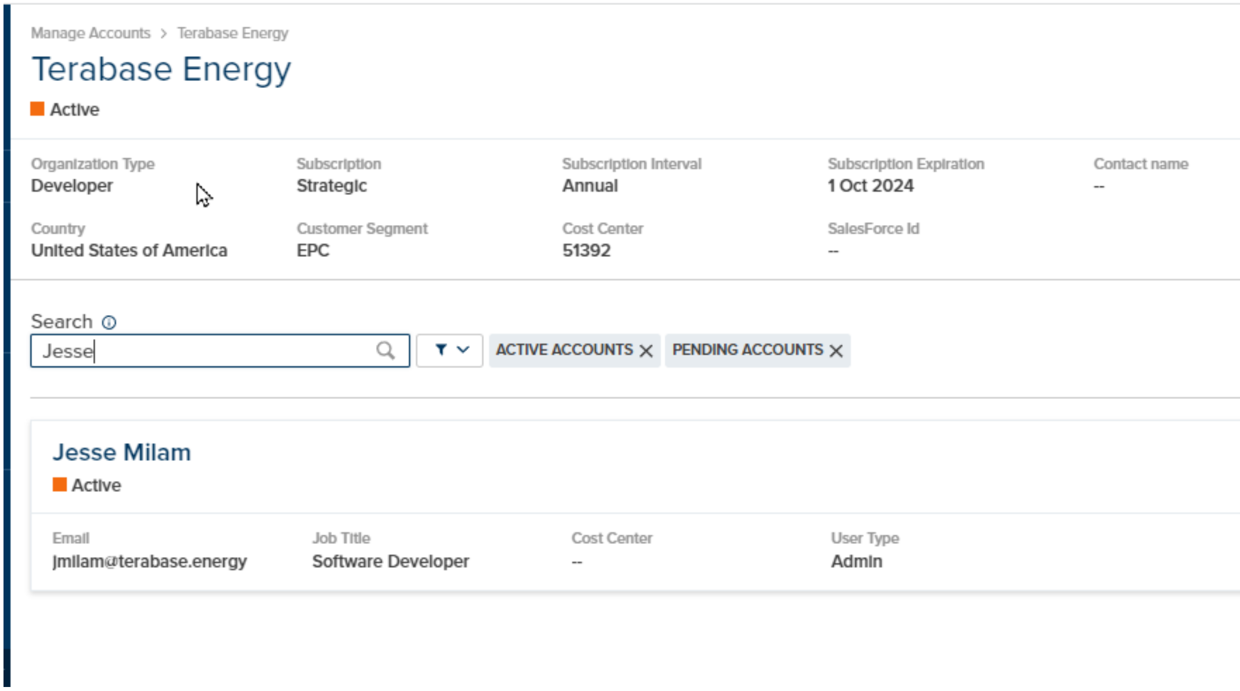
"I have never used PlantPredict and need an account".

Simply navigate to https://ui.plantpredict.com/signUp, provide the necessary information and complete your account registration.

"I have a PlantPredict account and am the company administrator."

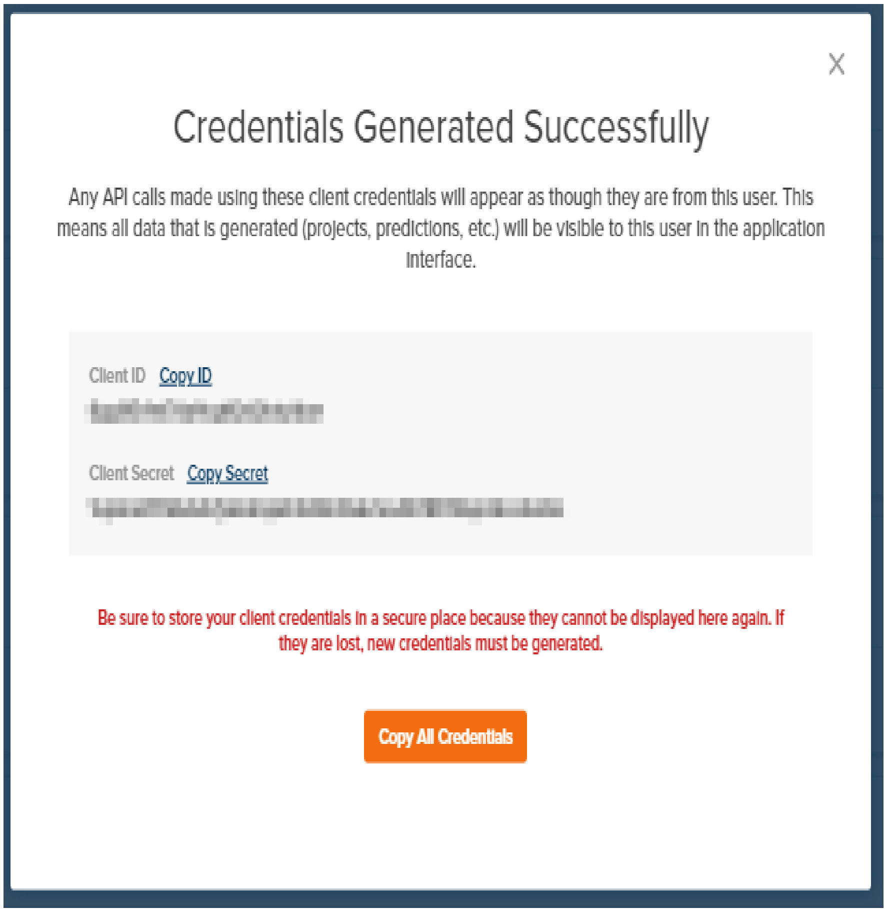
If you are the only person with a PlantPredict account in your organization/company, or the first person to have an account, you are likely the company admin. If you are a company admin, you will have a gear icon next to your name on the very bottom-left of the page when you log in on a web browser.

Click the gear icon. On the next page, search for the name of the person you would like to generate client credentials for, and click on their name.



Click on "Generate API Credentials" on the top right of the next page.

Copy each credential to your clipboard to be stored securely (step 2).



"I have a PlantPredict account but am not the company administrator."

Contact the person in your organization who is the company admin, and provide to them a link to this page.

1.2.2 Step 2: Store your API credentials securely.

1.2.3 Step 3: Authenticate and receive a token.

At the beginning of any script/Python session, execute the following code to authenticate with the PlantPredict servers to generate an access token, which is stored on an `Api` object.

```
import plantpredict

api = plantpredict.Api(
    client_id="INSERT CLIENT_ID FROM API CREDENTIALS",
    client_secret="INSERT CLIENT_SECRET FROM API CREDENTIALS"
)
```

The `Api` object is then used to instantiate other PlantPredict entities (see *Example Usage*).

> **Warning:** The access token will expire after 1 hour. If your script requires more than one hour to complete, the SDK will automatically generate a new token using a refresh token.

1.3 SDK Reference

Detailed information on how to use PlantPredict can be found in the User Manual. The manual is written in the context of PlantPredict's GUI, but is fully applicable to the API. Additional documentation for the general PlantPredict API, including a full list of endpoints and their respective inputs/outputs can be found here.

1.3.1 Classes

Api

class plantpredict.api.**Api**(*client_id*, *client_secret*, *base_url='https://api.plantpredict.terabase.energy'*, *auth_url='https://terabase-prd.auth.us-west-2.amazoncognito.com/oauth2/token'*)

Bases: `object`

project(***kwargs*)

prediction(***kwargs*)

powerplant(***kwargs*)

geo(***kwargs*)

inverter(***kwargs*)

module(***kwargs*)

weather(***kwargs*)

ashrae(***kwargs*)

Project

class plantpredict.project.**Project**(*api*, *id=None*, *name=None*, *latitude=None*, *longitude=None*)

Bases: `plantpredict.plant_predict_entity.PlantPredictEntity`

The Project entity in PlantPredict defines the location info and serves as a container for any number of Predictions.

create()

POST */Project/*

Creates a new `plantpredict.Project` entity in the PlantPredict database using the attributes assigned to the local object instance. Automatically assigns the resulting `id` to the local object instance. See the minimum required attributes (below) necessary to successfully create a new `plantpredict.Project`. Note that the full scope of attributes is not limited to the minimum required set.

Use `plantpredict.Project.assign_location_attributes()` to automatically assign all required (and non-required) location-related/geological attributes.

Required Attributes

Table 1: Minimum required attributes for successful Prediction creation

Field	Type	Description
name	str	Name of the project
latitude	float	North-South GPS coordinate of the Project location. Must be between `-90` and `90` - units `[decimal degrees]`.
longitude	float	East-West coordinate of the Project location, in decimal degrees. Must be between `-180` and `180` units `[decimal degrees]`.
country	str	Full name of the country of the Project's location.
country_code	str	Country code of the Project's location (ex. US for United States, AUS for Australia, etc.)
elevation	float	The elevation of the Project location above seal level units `[m]`.
standard_offset_from_utc	float	Time zone with respect to Greenwich Mean Time (GMT) in +/- hours offset.

delete()

HTTP Request: DELETE /Project/{ProjectId}

Deletes an existing Project entity in PlantPredict. The local instance of the Project entity must have attribute self.id identical to the project id of the Project to be deleted.

Returns A dictionary {"is_successful": True}.

Return type dict

get()

HTTP Request: GET /Project/{Id}

Retrieves an existing Project entity in PlantPredict and automatically assigns all of its attributes to the local Project object instance. The local instance of the Project entity must have attribute self.id identical to the project id of the Project to be retrieved.

Returns A dictionary containing all of the retrieved Project attributes.

Return type dict

update()

HTTP Request: PUT /Project

Updates an existing Project entity in PlantPredict using the full attributes of the local Project instance. Calling this method is most commonly preceded by instantiating a local instance of Project with a specified project id, calling the Project.get() method, and changing any attributes locally.

Returns A dictionary {"is_successful": True}.

Return type dict

get_all_predictions(***kwargs*)

HTTP Request: GET /Project/{ProjectId}/Prediction

Retrieves the full attributes of every Prediction associated with the Project.

Returns A list of dictionaries, each containing the attributes of a Prediction entity.

Return type list of dict

search(*latitude*, *longitude*, *search_radius=1.0*)

HTTP Request: GET /Project/Search

Searches for all existing Project entities within a search radius of a specified latitude/longitude.

Parameters

- **latitude** (*float*) – North-South coordinate of the Project location, in decimal degrees.
- **longitude** (*float*) – East-West coordinate of the Project location, in decimal degrees.
- **search_radius** (*float*) – search radius in miles

Returns TODO

assign_location_attributes (***kwargs*)

Returns

Prediction

class plantpredict.prediction.**Prediction** (*api*, *id=None*, *project_id=None*, *name=None*)

Bases: `plantpredict.plant_predict_entity.PlantPredictEntity`

The `plantpredict.Prediction` entity models a single energy prediction within a `plantpredict.Project`.

create (*use_closest_ashrae_station=True*, *error_spa_var=2.0*, *error_model_acc=2.9*, *error_int_ann_var=3.0*, *error_sens_acc=5.0*, *error_mon_acc=2.0*, *year_repeater=1*, *status=1*)

POST */Project/ :py:attr:'project_id' /Prediction*

Creates a new `plantpredict.Prediction` entity in the PlantPredict database using the attributes assigned to the local object instance. Automatically assigns the resulting `id` to the local object instance. See the minimum required attributes (below) necessary to successfully create a new `plantpredict.Prediction`. Note that the full scope of attributes is not limited to the minimum required set. **Important Note:** the minimum required attributes necessary to create a `plantpredict.Prediction` is not sufficient to successfully call `plantpredict.Prediction.run()`.

Required Attributes

Table 2: Minimum required attributes for successful Prediction creation

Field	Type	Description
name	str	Name of prediction
project_id	int	ID of project within which to contain the prediction
year_repeater	int	Must be between `1` and `50` - unitless.

Example Code

First, import the plantpredict library and receive an authentication [EDIT THIS] plantpredict.self.api.access_token in your Python session, as shown in Step 3 of *API Authentication*. Then instantiate a local Prediction. object.

```
module_to_create = plantpredict.Prediction()
```

Populate the Prediction's require attributes by either directly assigning them…

```
from plantpredict.enumerations import PredictionStatusEnum

prediction_to_create.name = "Test Prediction"
prediction_to_create.project_id = 1000
prediction_to_create.status = PredictionStatusEnum.DRAFT_SHARED
prediction_to_create.year_repeater = 1
```

…OR via dictionary assignment.

```
prediction_to_create.__dict__ = {
    "name": "Test Prediction",
    "model": "Test Module",
    "status": PredictionStatusEnum.DRAFT_SHARED,
    "year_repeater": 1,
}
```

Create a new prediction in the PlantPredict database, and observe that the Module now has a unique database identifier.

```
prediction_to_create.create()

print(prediction_to_create.id)
```

Returns A dictionary containing the prediction id.

Return type dict

delete()

HTTP Request: DELETE /Project/{ProjectId}/Prediction/{Id}

Deletes an existing Prediction entity in PlantPredict. The local instance of the Project entity must have attribute self.id identical to the prediction id of the Prediction to be deleted.

Returns A dictionary {"is_successful": True}.

Return type dict

get(*id=None*, *project_id=None*)

HTTP Request: GET /Project/{ProjectId}/Prediction/{Id}

Retrieves an existing Prediction entity in PlantPredict and automatically assigns all of its attributes to the local Prediction object instance. The local instance of the Prediction entity must have attribute self.id identical to the prediction id of the Prediction to be retrieved.

Returns A dictionary containing all of the retrieved Prediction attributes.

Return type dict

update()

HTTP Request: PUT /Project/{ProjectId}/Prediction

Updates an existing Prediction entity in PlantPredict using the full attributes of the local Prediction instance. Calling this method is most commonly preceded by instantiating a local instance of Prediction with a specified prediction id, calling the Prediction.get() method, and changing any attributes locally.

Returns A dictionary {"is_successful": True}.

Return type dict

run(***kwargs*)

POST /Project/{ProjectId}/Prediction/{PredictionId}/Run

Runs the Prediction and waits for simulation to complete. The input variable "export_options" should take the

Parameters **export_options** – Contains options for exporting

Returns

get_results_summary(***kwargs*)

GET /Project/{ProjectId}/Prediction/{Id}/ResultSummary

get_results_details(***kwargs*)
GET /Project/{ProjectId}/Prediction/{Id}/ResultDetails

get_nodal_data(***kwargs*)
GET /Project/{ProjectId}/Prediction/{Id}/NodalJson

clone(***kwargs*)

Parameters **new_prediction_name** –

Returns

change_status(***kwargs*)
Change the status (and resulting sharing/privacy settings) of a prediction (ex. from py:attr:*DRAFT_PRIVATE* to py:attr:*DRAFT-SHARED*.

Parameters

- **new_status** (*int*) – Enumeration representing status to change prediction to. See (or import) *plantpredict.enumerations.PredictionStatusEnum*.
- **note** (*str*) – Description of reason for change.

Returns

PowerPlant

class plantpredict.powerplant.**PowerPlant**(*api*, *project_id=None*, *prediction_id=None*, *use_cooling_temp=True*, ***kwargs*)
Bases: plantpredict.plant_predict_entity.PlantPredictEntity

Represents the hierarchical structure of a power plant in PlantPredict. There is a one-to-one relationship between a *PowerPlant* and *Prediction*. It is linked to that prediction via the attributes project_id and prediction_id.

All classes that inherit from PlantPredictEntity follow the same general usage pattern. The core class methods (*get*, *create*, and *update*) require that certain attributes be assigned to the instance of the class in order to run successfully, rather than requiring direct variable inputs to the method call itself. For methods beyond these four, the input requirements might be either attribute assignments or variable inputs to the method.

Sample code for properly building a *PowerPlant* can be found in *Example Usage*. While a new *PowerPlant* can be initialized via its __init__() method, as in the following example:

```
powerplant = plantpredict.powerplant.PowerPlant(api, project_id=1, prediction_
↪id=2)
```

it is recommended to use the *Api* factory method *powerplant()*, as in the following example:

```
powerplant = api.powerplant(project_id=1, prediction_id=2)
```

where both cases assume that api is a properly defined *Api* object.

Note on parameters listed below: This list of attributes is comprehensive, but does not encompass 100% of parameters that might be available via *get()* after the associated prediction is run. The list includes all relevant attributes that a user should/can set upon building the *PowerPlant*, plus some of the post-prediction-run parameters.

Parameters

- **api** (plantpredict.api.Api) – An properly initialized instance of the PlantPredict API client class, *Api*, which is used for authentication with the PlantPredict servers, given a user's unique API credentials.

- **project_id** (*int, None*) – Unique identifier for the *Project* with which to associate the power plant. Must represent a valid, exiting project in the PlantPredict database.
- **prediction_id** (*int, None*) – Unique identifier for the *Prediction* with which to associate the power plant. Must represent a valid, existing Prediction on the given Project in the PlantPredict database, as represented by the input `project_id`.
- **use_cooling_temp** (*bool*) – If `True`, the `kva_rating` of each inverter in the power plant is calculated based on the 99.6 cooling temperature of the nearest ASHRAE station to the corresponding *Project* (as specified by `project_id`), the elevation of the *Project*, and the elevation/temperature curves of the inverter model specified by `inverter_id`. Defaults to `True`. If `False`, the `kva_rating` of each inverter in the power plant is set as the `apparent_power` of the inverter model specified by `inverter_id`.
- **lgia_limitation** (*float*) – Maximum power output limit for power plant according to its Large Generator Interconnection Agreement (LGIA). Must be between `0` and `2000` - units `[MWac]`.
- **availability_loss** (*float*) – Accounts for losses due to any plant-wide outage events such as inverter shutdowns/failures. Must be between `0` and `25` - units `[%]`.
- **power_factor** (*float*) – The ratio of the power that can be used and the product of the operating current and voltage (also referred to as Plant kVA Derate). Defaults to `1.0`. Must be between `0` and `1`, where `1` is a "unity" power factor. Defaults to `1.0` in `__init__()` and automatically recalculated when *create()* called.
- **transformers** (*list*) – Defaults to an empty list (`[]`). See "Example contents of `transformers`" below for sample contents. Use the "power plant builder" method *add_transformer()* to easily add a new transformer to the attribute `transformers`.
- **transmission_lines** (*list*) – Defaults to an empty list (`[]`). See "Example contents of `transmission_lines`" below for sample contents. Use the "power plant builder" method *add_transmission_line()* to easily add a new transmission line to the attribute `transmission_lines`.
- **blocks** (*list*) – Defaults to an empty list (`[]`). See "Example contents of `blocks`" below for sample contents. Use the "power plant builder" method *add_block()* to easily add a new block to the attribute `blocks`. Subsequently use the methods *add_array()*, *add_inverter()*, and *add_dc_field()* to build out the full power plant hierarchical structure.

Below are some samples of the more complex attributes that would be populated after calling *get()* on an existing power plant in PlantPredict. This also is a sample of what the contents might look like before creating a new powerplant with *create()* (or update an existing one with *update()*:

Example contents of `transformers`

```
powerplant.transformers = [{
    "id": 23982,
    "rating": 0.6,                  # units [MVA]
    "high_side_voltage": 4.0,       # units [kV]
    "no_load_loss": 0.5,            # units [%]
    "full_load_loss": 1.0,          # units [%]
    "ordinal": 1
}]
```

Example contents of `transmission_lines`

```
powerplant.transmission_lines = [{
    "id": 48373,
    "length": 2.0,                              # units [km]
    "resistance": 0.5,                          # units [Ohms/300 m]
    "number_of_conductors_per_phase": 3,
    "ordinal": 1
}]
```

Example contents of `blocks`

```
from plantpredict.enumerations import TrackingTypeEnum, ModuleOrientationEnum,␣
↪BacktrackingTypeEnum

powerplant.blocks = [{
    "name": 1,
    "id": 57383,
    "description": "Description of block."
    "repeater": 5,
    "energization_date": "2019-12-26T16:43:55.867Z",
    "use_energization_date": True,
    "arrays": [{
        "name": 1,
        "id": 22323,
        "description": "Description of array.",
        "repeater": 2,
        "ac_collection_loss": 1.0,                          # units [%]
        "das_load": 1.2,                                    # units [%]
        "cooling_load": 0.8,                                # units [%]
        "additional_losses": 0.1,                           # units [%]
        "match_total_inverter_kva": True,
        "transformer_enabled": True,
        "transformer_kva_rating": 600.0,                    # units [kVA]
        "transformer_high_side_voltage": 34.7,              # units [V]
        "transformer_no_load_loss": 0.2,                    # units [%]
        "transformer_full_load_loss": 0.7,                  # units [%]
        "tracker_motor_losses": 0.1,                        # units [%]
        "inverters": [{
            "name": "A",
            "id": 234290,
            "description": "Description of inverter."
            "repeater": 1,
            "inverter_id": 242,
            "inverter": {} # Inverter model contents
            "setpoint_kw": 600.0,                                   # units␣
↪[kW]
            "power_factor": 1.0,
            "kva_rating": 600.0,                                    # units␣
↪[kW]
            "dc_fields": [{
                "name": 1,
                "id": 235324,
                "description": "Description of DC field.",
                "repeater": 3,
                "module_id": 749,
                "module": {}  # Module model contents
                "tracking_type": TrackingTypeEnum.FIXED_TILT,
                "module_orientation": ModuleOrientationEnum.PORTRAIT,
```

(continues on next page)

(continued from previous page)

```
                "tables_removed_for_pcs": 0,
                "modules_high": 4,
                "modules_wide": 18,
                "lateral_intermodule_gap": 0.02,  # units [m]
                "vertical_intermodule_gap": 0.02,  # units [m]
                "field_length": 20.0,  # units [m]
                "field_width": 11.0,  # units [m]
                "collector_bandwidth": 2.2,  # units [m]
                "table_length": 6.7,  # units [m]
                "tables_per_row": 3,
                "post_to_post_spacing": 1.8,  # units [m]
                "number_of_rows": 16,
                "table_to_table_spacing": 0.05,  # units [m]
                "module_azimuth": 180,  # units [degrees]
                "module_tilt": 30,  # units [degrees]
                "tracking_backtracking_type": BacktrackingTypeEnum.TRUE_TRACKING,
                "tracker_pitch_angle_d": 0,  # units [degrees]
                "minimum_tracking_limit_angle_d": -60.0,  # units [degrees]
                "maximum_tracking_limit_angle_d": 60.0,  # units [degrees]
                "tracker_stow_angle": 0,  # units [degrees]
                "post_height": 1.5,  # units [m]
                "structure_shading": 2.0,  # units [%]
                "backside_mismatch": 1.0,  # units [%]
                "field_dc_power": 800.0,  # units [kW]
                "modules_wired_in_series": 10,
                "number_of_series_strings_wired_in_parallel": 400,
                "planned_module_rating": 325.0,  # units [W]
                "sandia_conductive_coef": -3.47,
                "sandia_convective_coef": -0.0594,
                "cell_to_module_temp_diff": 3.0,  # units [deg-C]
                "heat_balance_conductive_coef": 30.7,
                "heat_balance_convective_coef": 0.0,
                "module_mismatch_coefficient": 1.0,  # units [%]
                "module_quality": 1.0,  # units [%]
                "light_induced_degradation": 1.0,  # units [%]
```

(continues on next page)

(continued from previous page)

```
                    "tracker_load_loss": 0.0,                                   ␣
↪   # units [%]
                    "dc_wiring_loss_at_stc": 1.5,                               ␣
↪   # units [%]
                    "dc_health": 1.0,                                           ␣
↪   # units [%]
                }],
            }],
    }],
}]
```

create()

POST */Project/* `project_id` */Prediction/* `prediction_id` */PowerPlant*

Creates a new power plant in the PlantPredict database with the attributes assigned to the instance of *PowerPlant*. Automatically attaches it to a project/prediction existing in PlantPredict associated with the assigned values for `project_id` and `prediction_id`. Also automatically calculates the average power factor (plant design derate) based on the power factors of each inverter. See *PowerPlant* documentation attributes required to successfully call this method.

Returns Dictionary with contents `{'is_successful': True}`.

Return type dict

get()

GET */Project/* `project_id` */Prediction/* `prediction_id` */PowerPlant*

Retrieves an existing *PowerPlant* from the PlantPredict database according to the values assigned for `project_id` and `prediction_id`, and automatically assigns all of its attributes to the object instance.

Returns A dictionary containing all of the retrieved *PowerPlant* attributes. (Matches the contents of the attributes `__dict__` after calling this method).

Return type dict

get_json()

update_from_json(*json_power_plant=None*)

update()

PUT */Project/* `project_id` */Prediction/* `prediction_id` */PowerPlant*

Updates an existing *PowerPlant* entity in PlantPredict using the full attributes of the object instance. Calling this method is most commonly preceded by instantiating an *PowerPlant* object with a particular `project_id` and `prediction_id` and calling *get()*, and changing any attributes locally.

Returns Dictionary with contents `{'is_successful': True}`.

Return type dict

add_transformer(*rating*, *high_side_voltage*, *no_load_loss*, *full_load_loss*, *ordinal*)

Appends a transformer to the attribute `transformers` to model the system-level of the power plant.

Parameters

- **rating** (*float*) – Transformer rating. Must be between `0.1` and `10000.0` - units `[MVA]`.

- **high_side_voltage** (*float*) – Transformer voltage. Must be between `1.0` and `1000.0` - units `[kV]`.
- **no_load_loss** (*float*) – Transformer loss at no load. Must be between `0.0` and `10.0` - units `[%]`.
- **full_load_loss** (*float*) – Transformer loss at full load. Must be between `0.0` and `10.0` - units `[%]`.
- **ordinal** (*int*) – Order in sequence of `transformers` and `transmission_lines` where `1` represents the closest entity to the power plant/farthest entity from the energy meter (1-indexed).

add_transmission_line (*length*, *resistance*, *number_of_conductors_per_phase*, *ordinal*)

Appends a transmission line to the attribute `transmission_lines` to model the system-level of the power plant.

Parameters

- **length** (*float*) – Length of transmission line. Must be between `0.1` and `100.0` - units `[km]`.
- **resistance** (*float*) – Transmission line resistivity (per 300m). Must be between `0.001` and `2` - units `[Ohms/300m]`.
- **number_of_conductors_per_phase** (*int*) – Number of conductors per phase. Must be between `1` and `10`.
- **ordinal** – Order in sequence of `transformers` and `transmission_lines` where `1` represents the closest entity to the power plant/farthest entity from the energy meter (1-indexed).

add_block (***kwargs*)

A "power plant builder" helper method that creates a new block and appends it to the attribute `blocks`. Block naming is sequential (numerically) - for instance, if there are 2 existing blocks with names `1` and `2` (accessible via key `name` on each block in list), the next block created by `add_block()` will automatically have `name` equal to `3`. This method does not currently account for the situation in which an existing power plant has blocks named non-sequentially.

Note that this addition is not persisted to PlantPredict unless *`update()`* is subsequently called.

Parameters

- **use_energization_date** (*bool*) – Enables use of energization date in power plant block. Defaults to `False`.
- **energization_date** (*str*) – Timestamp representing energization date of block. Uses format `2019-12-26T16:43:55.867Z` and defaults to `""`.

Returns Name of newly added block.

Return type int

clone_block (***kwargs*)

A "power plant builder" helper method that clones (copies) an existing block (and all of its children arrays/inverters/DC fields) and appends it to attribute `blocks`. Particularly useful when you want to create a new block that is similar to an existing block. Block naming is sequential (numerically) - for instance, if there are 2 existing blocks with names :py:data‘1‘ and `2` (accessible via key `name` on each block in list), the next block created by *`clone_block()`* will automatically have :py:data‘name‘ equal to :py:data‘3‘. This method does not currently account for the situation in which an existing power plant has blocks named non-sequentially.

Note that this addition is not persisted to PlantPredict unless *`update()`* is subsequently called.

Parameters **block_id_to_clone** (*int*) – Unique identifier of the block you wis you clone. Can be found in the relevant block dictionary (in list `self.blocks`) with key `id`.

Returns Name of newly cloned block.

Return type int

add_array(***kwargs*)

A "power plant builder" helper method that adds an array to the block specified by `block_name` on the *PowerPlant*. Array naming is sequential (numerically) - for instance, if there are 2 existing arrays with names `1` and `2` (accessible via key `name` for a given array dictionary), the next array created by *add_array()* will automatically have `name` equal to `3`. This method does not currently account for the situation in which an existing power plant has arrays named non-sequentially.

Note that this addition is not persisted to PlantPredict unless *update()* is subsequently called.

Parameters

- **block_name** (*int*) – Name (1-indexed integer) of the parent block to add the array to. Can be found in the relevant block dictionary (in attribute `blocks`) with key `id`. This value is returned for a new block when you create one with *add_block()*. Must be between `1` and `99`.
- **transformer_enabled** (*bool*) – If `True`, enables a medium-voltage (MV) transformer for the array. Defaults to `True`.
- **match_total_inverter_kva** (*bool*) – If `True`, the transformer size will match the total inverter kVA of the inverter behind the transformer, and the input `transformer_kva_rating` won't be used. Defaults to `True`.
- **transformer_kva_rating** (*float, None*) – User-specified transformer kVA rating. Only used if `match_total_inverter_kva` is set to `False`. Defaults to `None`. Must be between `0` and `20000` - units `[kVA]`.
- **repeater** (*int*) – Number of identical arrays of this type in the parent block. Defaults to `1`. Must be between `1` and `10000`.
- **ac_collection_loss** (*float*) – Accounts for ohmic losses in the AC wiring between the array and parent block. Defaults to `1`.Must be between `0` and `30` - units `[%]`.
- **das_load** (*float*) – Accounts for parasitic losses due to the data acquisition system (DAS). Can also be used for general time-constant parasitic loss accounting. Defaults to `800`. Must be between `0` and `5000` - units `[W]`.
- **cooling_load** (*float*) – Accounts for losses from the power conditioning system (PCS) shelter cooling system. Defaults to `0.0`. Must be between `0` and `5000` - units `[W]`.
- **additional_losses** (*float*) – Additional night time losses. Defaults to `0`. Must be between `0` and `20000` - units `[W]`.
- **transformer_high_side_voltage** (*float*) – Transformer high side voltage (the AC collection line voltage defines the high-side of a MV inverter). Defaults to `34.5`. Must be between `0` and `66` - units `[V]`.
- **transformer_no_load_loss** (*float*) – Accounts for transformer losses with no load. Defaults to `0.2`. Must be between `0` and `10` - units `[%]`.
- **transformer_full_load_loss** (*float*) – Accounts for transformer losses with full load. Defaults to `0.7`. Must be between `0` and `10` - units `[%]`.
- **description** (*str*) – Description of the array. Must be `250` characters or less. Defaults to `""`.

Raises ValueError – Raised if block_name is not a valid block name in the existing power plant.

Returns The name of the newly added array.

Return type int

add_inverter(***kwargs*)

A "power plant builder" helper method that adds an inverter to an array specified by array_name, which is a child of a block specified by block_name on the *PowerPlant*. Inverter naming is sequential (alphabetically) - for instance, if there are 2 existing inverters with names "A" and "B" (accessible via key name for a given inverter dictionary), the next array created by *add_inverter()* will automatically have name equal to "C". This method does not currently account for the situation in which an existing power plant has inverters named non-sequentially.

The inverter :py:data:'kva_rating' will be set based on the power plant-level attribute use_cooling_temp. If use_cooling_temp is True, this value is automatically calculated based on the 99.6 cooling temperature of the nearest ASHRAE station to the corresponding *Project* (as specified by the attribute project_id), the elevation of the *Project*, and the elevation/temperature curves of the inverter model specified by inverter_id. If use_cooling_temp is False, then kva_rating is set as the apparent_power of the inverter model specified by inverter_id.

Note that this addition is not persisted to PlantPredict unless *update()* is subsequently called.

Parameters

- **block_name** (*int*) – Name (1-indexed integer) of the parent block to add the inverter to. Can be found in the relevant block dictionary (in attribute blocks) with key id. This value is returned for a new block when you create one with *add_block()*. Must be between 1 and 99.
- **array_name** (*int*) – Name (1-indexed integer) of the parent array to add the inverter to. This value is returned for a new array when you create one with *add_array()*. Must be between 1 and 99.
- **inverter_id** (*int*) – Unique identifier of an inverter model in the PlantPredict Inverter database to use.
- **setpoint_kw** (*float, None*) – Inverter setpoint. Must be between 1 and 10000 - units [kW]. If left as default (None), will be automatically calculated as the product between power_factor and the inverter kVA rating.
- **power_factor** (*float*) – The ratio of the power that can be used and the product of the operating current and voltage (also referred to as design derate). Must be between 0 and 1, where 1 is a "unity" power factor. Defaults to 1.0.
- **repeater** (*int*) – Number of identical inverters of this type in the parent array. Must be between 1 and 10000. Defaults to 1.

Raises ValueError – Raised if block_name is not a valid block name in the existing power plant, or if the block_name is valid but array_name is not a valid array name in the block. Also raised if setpoint_kw is not None and power_factor is not 1.0.

Returns The name of the newly added inverter.

Return type str

calculate_post_to_post_spacing_from_gcr(***kwargs*)

Useful helper method for calculating post_to_post_spacing based on a desired ground coverage ratio (GCR). post_to_post_spacing is a required input for *add_dc_field()*.

Parameters

- **ground_coverage_ratio** (*float*) – Ratio of collector bandwidth to row spacing - units [decimal].
- **module_id** (*int*) – Unique identifier of the module to be used in the DC field.
- **modules_high** (*int*) – Number of modules high per table (number of ranks). Must be between 1 and 50.
- **module_orientation** (*int, None*) – Represents the orientation (portrait or landscape) of modules in the DC field. If left as default (None), is automatically set as the module_orientation of the module model specified by module_id. Use *ModuleOrientationEnum*.
- **vertical_intermodule_gap** (*float*) – Vertical gap between each module on the mounting structure. Defaults to 0.02. Must be between 0 and py:data:*1* - units :py:data:'[m]'.

Returns Post to post spacing (row spacing) of DC field - units [m].

Return type float

static calculate_field_dc_power_from_dc_ac_ratio(*dc_ac_ratio*, *inverter_setpoint*)

Useful helper method for sizing the DC field capacity (field_dc_power) based on a desired DC AC ratio and known inverter setpoint. field_dc_power is a required input for *add_dc_field()*.

Parameters

- **dc_ac_ratio** (*float*) – Ratio of DC capacity of DC field to the AC capacity/inverter setpoint.
- **inverter_setpoint** (*float*) – Setpoint of parent inverter to the DC field. Can be found with key setpoint_kw in the dictionary representing the inverter. Must be between 1 and 10000 - units [kW].

Returns DC capacity for a DC field - units [kW].

Return type float

add_dc_field(***kwargs*)

A "power plant builder" helper method that adds a DC field to an inverter specified by inverter_name, which is a child of the array array_name, which is a child of a block specified by block_name on the *PowerPlant*. DC field naming is sequential (numerically) - for instance, if there are 2 existing DC fields with names 1 and 2 (accessible via key name for a given DC field dictionary), the next array created by *add_dc_field()* will automatically have name equal to 3. This method does not currently account for the situation in which an existing power plant has DC fields named non-sequentially.

Note that this addition is not persisted to PlantPredict unless *update()* is subsequently called.

Parameters

- **block_name** (*int*) – Name (1-indexed integer) of the parent block to add DC field to. Can be found in the relevant block dictionary (in attribute blocks) with key id. This value is returned for a new block when you create one with *add_block()*. Must be between 1 and 99.
- **array_name** (*int*) – Name (1-indexed integer) of the parent array to add DC field to. This value is returned for a new array when you create one with *add_array()*. Must be between 1 and 99.
- **inverter_name** (*str*) – Name (letter) of the parent array to add the DC field to. This value is returned for a new array when you create one with *add_inverter()*. Must be only 1 character.

- **module_id** (*int*) – Unique identifier of the module to be used in the DC field.
- **tracking_type** (*int*) – Represents the tracking type/mounting structure (Fixed Tilt or Tracker) of the DC field. Use *TrackingTypeEnum*. (Seasonal Tilt currently not supported in this package).
- **modules_high** (*int*) – Number of modules high per table (number of ranks). Must be between 1 and 50.
- **modules_wired_in_series** (*int*) – The number of modules electrically connected in series in a string.
- **post_to_post_spacing** (*float*) – Row spacing. Must be between 0.0 and 50.0 - units [m].
- **number_of_rows** (*int, None*) – Number of rows of tables in DC field. Must be between 1 and 10000. Defaults to 1.
- **strings_wide** (*int*) – Number of strings across per table. Multiplied by modules_wired_in_series to determine modules_wide. Must result in modules_wide between 1 and 100. Defaults to 1.
- **field_dc_power** (*float, None*) – DC capacity of the DC field. Defaults to *None*. Non-null value required if number_of_series_strings_wired_in_parallel is *None* and must be between 1 and 20000 - units [kW].
- **number_of_series_strings_wired_in_parallel** (*float, None*) – Number of strings of modules electrically connected in parallel in the DC field. Defaults to None. Non-null value required if field_dc_power is None, and must be between 1 and :py:data'10000'.
- **module_tilt** (*float, None*) – Tilt angle of modules in DC Field for a fixed tilt array. Defaults to None. Non-null value required required if tracking_type is equal to *FIXED_TILT*, and must be between 0 and 90 - units [degrees].
- **module_orientation** (*int, None*) – Represents the orientation (portrait or landscape) of modules in the DC field. If left as default (None), is automatically set as the module_orientation of the module model specified by module_id. Use *ModuleOrientationEnum*.
- **module_azimuth** (*float, None*) – Orientation of the entire DC field. The convention is 0.0 degrees for North-facing arrays. If left as default (None), is set to 180.0. Must be between 0 and 360 - units [degrees].
- **tracking_backtracking_type** (*int, None*) – Represents the backtracking algorithm (True-Tracking or Backtracking) used in DC Field. Use *BacktrackingTypeEnum*.
- **minimum_tracking_limit_angle_d** (*float*) – Minimum tracking angle for horizontal tracker array. Defaults to -60.0. Must be between -90 and 0 - units [degrees].
- **maximum_tracking_limit_angle_d** (*float*) – Maximum tracking angle for horizontal tracker array. Defaults to 60.0. Must be between 0 and 90 - units [degrees].
- **lateral_intermodule_gap** (*float*) – Lateral gap between each module on the mounting structure. Defaults to 0.02. Must be between 0 and py:data:*1* - units :py:data:'[m]'.

- **vertical_intermodule_gap** (*float*) – Vertical gap between each module on the mounting structure. Defaults to `0.02`. Must be between `0` and py:data:*1* - units :py:data:'[m]'.
- **table_to_table_spacing** (*float*) – Space between tables in each row. Defaults to `0.0`. Must be between `0` and `50`.
- **module_quality** (*float, None*) – Accounts for any discrepancy between manufacturer nameplate rating of module and actual performance. If left as default (`None`), is automatically set as the `module_quality` of the module model specified by `module_id`. Must be between `-200` and `99` - units `[%]`.
- **module_mismatch_coefficient** (*float, None*) – Accounts for losses due to mismatch in electrical characteristics among modules in the strings of the DC fields (and between strings in the DC field). If left as default (`None`), is automatically set as the `module_mismatch_coefficient` of the module model specified by `module_id`. Must be between `0` and `30` - units `[%]`.
- **light_induced_degradation** (*float, None*) – Accounts for losses due to light induced degradation. If left as default (`None`), is automatically set as the `light_induced_degradation` of the module model specified by `module_id`. Must be between `0` and `30` - units `[%]`.
- **dc_wiring_loss_at_stc** (*float*) – Accounts for losses across all electrical wiring in the DC field. Defaults to `1.5`. Must be between `0` and `30` - units `[%]`.
- **dc_health** (*float*) – Accounts for any losses related to DC health. Defaults to `1.0`. Must be between `-10` and `10` - units `[%]`.
- **heat_balance_conductive_coef** (*float, None*) – Thermal loss factor (constant component) of heat balance module surface temperature model. If left as default (`None`), is automatically set as the `heat_balance_conductive_coef` of the module model specified by `module_id`. Must be between `0` and `100`. This value is only used if `model_temp_model` is set to *HEAT_BALANCE* for the *Prediction* associated with the power plant by the attributes `project_id` and `prediction_id`.
- **heat_balance_convective_coef** (*float, None*) – Thermal loss factor (wind speed component) of heat balance module surface temperature model. If left as default (`None`), is automatically set as the `heat_balance_convective_coef` of the module model specified by `module_id`. Must be between `0` and `100`. This value is only used if `model_temp_model` is set to *HEAT_BALANCE* for the *Prediction* associated with the power plant by the attributes `project_id` and `prediction_id`.
- **sandia_conductive_coef** (*float, None*) – Coefficient `a` for the Sandia module surface temperature model. If left as default (`None`), is automatically set as the `sandia_conductive_coef` of the module model specified by `module_id`. Must be between `-5` and `0`. This value is only used if `model_temp_model` is set to *SANDIA* for the *Prediction* associated with the power plant by attributes `project_id` and `prediction_id`.
- **sandia_convective_coef** (*float, None*) – Coefficient `b` for the Sandia module surface temperature model. If left as default (`None`), is automatically set as the `sandia_convective_coef` of the module model specified by `module_id`. Must be between `-1` and `0`. This value is only used if `model_temp_model` is set to *SANDIA* for the *Prediction* associated with the power plant by attributes `project_id` and `prediction_id`.
- **cell_to_module_temp_diff** (*float, None*) – Difference between surface and cell temperature of modules. If left as default (`None`), is automatically set as the

cell_to_module_temp_diff of the module model specified by module_id. Must be between 0 and 15 - units [degrees-C].

- **tracker_load_loss** (*float*) – Accounts for losses from power use of horizontal tracker system. Defaults to 0.0. Must be between 0 and 100 - units [%].
- **post_height** (*float, None*) – Height of mounting structure (table) post. Defaults to None. If left as default (None), automatically calculated as ((collector_bandwidth * sin(tilt) / 2) + 1, where tilt is module_tilt if tracking_type is *FIXED_TILT*, or the largest of the absolute values of maximum_tracking_limit_angle_d/minimum_tracking_limit_angle_d if tracking_type is *HORIZONTAL_TRACKER*. However, if the calculated value is less than 1.5, post_height is defaulted to 1.5. Must be between 0 and 50 - units [m]. This value is only used if the module model specified with module_id is bifacial.
- **structure_shading** (*float*) – Accounts for backside of module losses from structure shading. Defaults to 0.0. Must be between 0 and 100 - units [%]. This value is only used if the module model specified with module_id is bifacial.
- **backside_mismatch** (*float, None*) – Accounts for losses due to inconsistent backside irradiance among modules in the DC field. Defaults to None. If left as default (None), is automatically set as the module_orientation of the module model specified by module_id. Must be between 0 and 100 - units [%]. This value is only used if the module model specified with module_id is bifacial.

Raises ValueError – Raised if block_name is not a valid block name in the existing power plant, or if the block_name is valid but array_name is not a valid array name in the block, or if array_name is valid but inverter_name is not a valid inverter in the array. Also raised if tracking_type is *FIXED_TILT* and module_tilt is None, or if tracking_type is *HORIZONTAL_TRACKER* and tracking_backtracking_type is None. Also raised if both field_dc_power and :py:data'number_of_series_strings_wired_in_parallel' are None or are both not None. Also raised if tracking_type is *SEASONAL_TILT*.

Returns The name of the newly added DC field.

Return type int

Weather

class plantpredict.weather.**Weather**(*api*, ***kwargs*)

Bases: plantpredict.plant_predict_entity.PlantPredictEntity

The full contents of the Weather database entity (in JSON) can be found under "GET /Weather/{Id}" in the general PlantPredict API documentation.

create()

POST /Weather

Creates a new Weather entity.

Required Attributes

Table 3: Minimum required attributes for successful Weather creation

Field	Type	Description
name	str	Name of weather file
country_code	str	Country code of the Weather's location (ex. US for United States, AUS for Australia, etc.) `plantpredict.Geo.get_location_info()` will return this information.
country	str	Full name of the country of the Weather's location. `plantpredict.Geo.get_location_info()` will return this information.
latitude	float	North-South coordinate of the Weather location (in decimal degrees).
longitude	float	East-West coordinate of the Weather location (in decimal degrees).
data_provider	int	Represents a weather data source. See (and/or import) *plantpredict.enumerations.WeatherDataProviderEnum* for a list of options.
weather_details	list of dict	The code block below contains an example of one timestamp (array element) of this field, as well as information on which dictionary keys are required.

```
weather_details[109] = {
    "index": 110,                              # REQUIRED | is equal to the list␣
↪index + 1
    "time_stamp": "2018-01-01T1:00:00",        # REQUIRED
    "global_horizontal_irradiance": 139.3,     # REQUIRED if no 'plane_of_array_
↪irradiance' | [W/m^2]
    "diffuse_horizontal_irradiance": 139.3,    # [W/m^2]
    "direct_normal_irradiance": 0.0,           # [W/m^2]
    "beam_horizontal_irradiance": 0.0,         # [W/m^2]
    "plane_of_array_irradiance": 100.0,        # REQUIRED if no 'global_
↪horizontal_irradiance' | [W/m^2]
    "temperature": 1.94,                       # REQUIRED | [degrees-Celsius]
    "relative_humidity": 74.5,                 # [%]
    "precipitable_water": 2.0,                 # [cm]
    "soiling_loss": 0.19                       # [%]
}
```

Returns A dictionary containing the weather id.

Return type dict

delete()

DELETE /Weather/{WeatherId}

Deletes an existing Weather entity in PlantPredict. The local instance of the Weather entity must have attribute self.id identical to the weather id of the Weather to be deleted.

Returns A dictionary {"is_successful": True}.

Return type dict

get()

GET /Weather/{Id}

Retrieves an existing Weather entity in PlantPredict and automatically assigns all of its attributes to the local Weather object instance. The local instance of the Weather entity must have attribute self.id identical to the weather id of the Weather to be retrieved.

Returns A dictionary containing all of the retrieved Weather attributes.

Return type dict

update()

PUT /Weather

Updates an existing Weather entity in PlantPredict using the full attributes of the local Weather object instance. Calling this method is most commonly preceded by instantiating a local instance of Weather with a specified weather id, calling the Weather.get() method, and changing any attributes locally.

The required fields are identical to those of `plantpredict.weather.create()` with the addition of: .. csv-table:: Minimum required attributes for successful Weather creation

delim ;

header Field; Type; Description

stub-columns 1

id; int; Unique identifier of existing Weather entity.

Returns A dictionary {"is_successful": True}.

Return type dict

get_details(***kwargs*)

GET /Weather/{Id}/Detail

Returns detailed time series of Weather entity.

Returns A list of dictionaries where each dictionary contains one timestamp of detailed weather data.

Return type list of dicts

search(***kwargs*)

GET /Weather/Search

Searches for all existing Weather entities within a search radius of a specified latitude/longitude.

Parameters

- **latitude** (*float*) – North-South coordinate of the Weather location, in decimal degrees.
- **longitude** (*float*) – East-West coordinate of the Project location, in decimal degrees.
- **search_radius** (*float*) – search radius in miles

Returns #TODO

Return type list of dicts

download(***kwargs*)

POST /Weather/Download/{Provider}

Parameters

- **latitude** (*float*) –
- **longitude** (*float*) –
- **provider** (*int*) – Represents a weather data source. See (and/or import) *plantpredict.enumerations.WeatherSourceTypeAPIEnum* for a list of options.

Returns #TODO

Return type dict

change_status(***kwargs*)
POST /Weather/Status Change the status (and resulting sharing/privacy settings) of a weather file (ex. from py:attr:*DRAFT_PRIVATE* to py:attr:*DRAFT-SHARED*. :param int new_status: Enumeration representing status to change weather to. See (or import)

plantpredict.enumerations.LibraryStatusEnum.

Parameters **note** (*str*) – Description of reason for change.

Returns

generate_weather(***kwargs*)
Post /Weather/GenerateWeather

Returns a synthetic weather time series based on monthly data. The monthly data must be defined as a list of dicts in a class attribute "monthly_values"

Returns A dictionary with all weather parameters, including and especially hourly synthetic data in "weather_details".

Return type dict

Module

class plantpredict.module.**Module**(*api*, ***kwargs*)
Bases: `plantpredict.plant_predict_entity.PlantPredictEntity`

The `Module` entity models all of the characteristics of a photovoltaic solar module (panel).

create()
POST */Module*

Creates a new `plantpredict.Module` entity in the PlantPredict database using the attributes assigned to the local object instance. Automatically assigns the resulting `id` to the local object instance. See the minimum required attributes (below) necessary to successfully create a new `plantpredict.Module`. Note that the full scope of attributes is not limited to the minimum required set.

Required Attributes

Table 4: Minimum required attributes for successful Module creation

Field	Type	Description
name	str	Name of module file
model	str	Model number/name of module (can be the same as `name`)
manufacturer	str	Module manufacturer
length	float	Long side of the module. Must be between `0.0` and `10000.0` - units `[mm]`.
width	float	Short side of the module. Must be between `0.0` and `10000.0` - units `[mm]`.
cell_technology_type	int	Represents the cell technology type (CdTe, poly c-Si PERC, etc). Use *plantpredict.enumerations.CellTechnologyTypeEnum*.
pv_model	int	Represents the 1-diode model type (1-Diode, 1-Diode with recombination). Use *plantpredict.enumerations.PVModelTypeEnum*.
construction_type	int	Represents the module construction (Glass-Glass, Glass-Backsheet). Use *plantpredict.enumerations.ConstructionTypeEnum*.
stc_short_circuit_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_open_circuit_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_mpp_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_mpp_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_power_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
stc_short_circuit_current_temp_coef	float	Must be between `-0.3` and `2.0` - units `[%/deg-C]`.
stc_open_circuit_voltage_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
saturation_current_at_stc	float	Must be between `1e-13` and `1e-6` - units `[A]`.
diode_ideality_factor_at_stc	float	Must be between `0.1` and `5.0` - unitless.
linear_temp_dependence_on_gamma	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
exponential_dependency_on_shunt_resistance	float	Must be between `1.0` and `100.0` - unitless.
series_resistance_at_stc	float	Must be between `0.0` and `100.0` - units `[Ohms]`
dark_shunt_resistance	float	Must be between `100.0` and `100000.0` - units `[Ohms]`.
shunt_resistance_at_stc	float	Must be between `0.0` and `100000.0` - units `[Ohms]`.
bandgap_voltage	float	Must be between `0.5` and `4.0` - units `[V]`.
heat_absorption_coef_alpha_t	float	Must be between `0.1` and `1.0`.
reference_irradiance	float	Must be between `400.0` and `1361.0` - units `[W/m^2]`.
built_in_voltage	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `3.0` - units `[V]`.
recombination_parameter	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `30.0` - units `[V]`

Example Code

First, import the plantpredict library and create an instance of *plantpredict.api.Api* in your Python session, to authenticate as shown in Step 3 of *API Authentication*. Then instantiate a local *plantpredict.module.Module* object.

```
module_to_create = plantpredict.Module()
```

Populate the Module's require attributes by either directly assigning them. . .

```
from plantpredict.enumerations import CellTechnologyTypeEnum, PVModelTypeEnum,
↪ConstructionTypeEnum

module_to_create.name = "Test Module"
module_to_create.model = "Test Module"
module_to_create.manufacturer = "Solar Company"
module_to_create.length = 2009
module_to_create.width = 1232
module_to_create.cell_technology_type = CellTechnologyTypeEnum.CDTE
module_to_create.pv_model = PVModelTypeEnum.ONE_DIODE_RECOMBINATION
module_to_create.construction_type = ConstructionTypeEnum.GLASS_GLASS
module_to_create.stc_short_circuit_current = 2.54
module_to_create.stc_open_circuit_voltage = 219.2
module_to_create.stc_mpp_current = 2.355
module_to_create.stc_mpp_voltage = 182.55
module_to_create.stc_power_temp_coef = -0.32
module_to_create.stc_short_circuit_current_temp_coef = 0.04
module_to_create.stc_open_circuit_voltage_temp_coef = -0.28
module_to_create.saturation_current_at_stc = 2.415081e-12
module_to_create.diode_ideality_factor_at_stc = 1.17
module_to_create.linear_temp_dependence_on_gamma = -0.08
module_to_create.exponential_dependency_on_shunt_resistance = 5.5
module_to_create.series_resistance_at_stc = 5.277
module_to_create.dark_shunt_resistance = 6400
module_to_create.shunt_resistance_at_stc = 6400
module_to_create.bandgap_voltage = 1.5
module_to_create.heat_absorption_coef_alpha_t = 0.9
module_to_create.reference_irradiance = 1000

# required for modules with recombination
module_to_create.built_in_voltage = 0.9
module_to_create.recombination_parameter = 0.9
```

. . . OR via dictionary assignment.

```
module_to_create.__dict__ = {
    "name": "Test Module",
    "model": "Test Module",
    "manufacturer": "Solar Company",
    "length": 2009,
    "width": 1232,
    "cell_technology_type": CellTechnologyTypeEnum.CDTE,
    "pv_model": PVModelTypeEnum.ONE_DIODE_RECOMBINATION,
    "construction_type": ConstructionTypeEnum.GLASS_GLASS,
    "stc_short_circuit_current": 2.54,
    "stc_open_circuit_voltage": 219.2,
    "stc_mpp_current": 2.355,
    "stc_mpp_voltage": 182.55,
    "stc_power_temp_coef": -0.32,
    "stc_short_circuit_current_temp_coef": 0.04,
    "stc_open_circuit_voltage_temp_coef": -0.28,
    "saturation_current_at_stc": 2.415081e-12,
    "diode_ideality_factor_at_stc": 1.17,
```

(continues on next page)

(continued from previous page)

```
    "linear_temp_dependence_on_gamma": -0.08,
    "exponential_dependency_on_shunt_resistance": 5.5,
    "dark_shunt_resistance": 6400,
    "shunt_resistance_at_stc": 6400,
    "bandgap_voltage": 1.5,
    "heat_absorption_coef_alpha_t": 0.9,
    "reference_irradiance": 1000,
    "built_in_voltage": 0.9,
    "recombination_parameter": 0.9
}
```

Create a new module in the PlantPredict database, and observe that the Module now has a unique database identifier.

```
module_to_create.create()

print(module_to_create.id)
```

Returns A dictionary containing the module id.

Return type dict

delete()

DELETE */Module/* `id`

Deletes an existing `plantpredict.Module` entity in the PlantPredict database according to the `id` of the local object instance.

Example Code

First, import the plantpredict library and create an instance of *`plantpredict.api.Api`* in your Python session, to authenticate as shown in Step 3 of *API Authentication*. Then instantiate a local *`plantpredict.module.Module`* object with the `id` of the target Module in the PlantPredict database.

```
module_to_delete = plantpredict.Module(id=99999)
```

Delete the Module.

```
module_to_delete.delete()
```

Returns A dictionary {“is_successful”: True}.

Return type dict

get()

GET */Module/* `id`

Retrieves an existing `plantpredict.Module` entity from the PlantPredict database according to the `id` of the local object instance, and automatically assigns all of its attributes to the local object instance.

Example Code

First, import the plantpredict library and create an instance of *`plantpredict.api.Api`* in your Python session, to authenticate as shown in Step 3 of *API Authentication*. Then instantiate a local *`plantpredict.module.Module`* object with the `id` of the target module in the PlantPredict database.

```
module_to_get = plantpredict.Module(id=99999)
```

Retrieve the Module from the PlantPredict database.

```
module_to_get.get()
```

This will automatically assign all of that Module's attributes to the local object instance. All of the attributes are now readily accessible in the local Python session.

```
module_name = module_to_get.name
isc = module_to_get.stc_short_circuit_current
```

Returns A dictionary containing all of the retrieved Module attributes. (Matches the result of calling *self.__dict__* after calling this method).

Return type dict

update()

PUT */Module*

Updates an existing `plantpredict.Module` entity in PlantPredict using the full attributes of the local object instance. Calling this method is most commonly preceded by instantiating a local instance of `plantpredict.Module` with a specified `id`, calling `plantpredict.Module.get()`, and changing any attributes locally.

Example Code

First, import the plantpredict library and create an instance of *`plantpredict.api.Api`* in your Python session, to authenticate as shown in Step 3 of *API Authentication*. Then instantiate a local *`plantpredict.module.Module`* object with the `id` of the target module in the PlantPredict database.

```
module_to_update = plantpredict.Module(id=99999)
```

Retrieve the Module from the PlantPredict database.

```
module_to_update.get()
```

This will automatically assign all of that Module's attributes to the local object instance. Any/all of the attributes can now be modified locally.

```
module.name = "New Name"
module.shunt_resistance_at_stc = 8000
```

Persist (update) the local changes to the PlantPredict database.

```
module.update()
```

Returns A dictionary {"is_successful": True}.

Return type dict

upload_pan_file(***kwargs*)

creates a new module from a source .pan file

parse_pan_file(***kwargs*)

creates a new module from a source .pan file

create_from_json(***kwargs*)
creates a new module from a source JSON file

get_module_list(***kwargs*)

Returns a list of all modules to which a user has access.

generate_single_diode_parameters_default(***kwargs*)
POST */Module/Generator/GenerateSingleDiodeParametersDefault*

Generates single-diode parameters from module electrical characteristics available on any standard manufacturers' module datasheet. Detailed documentation on the algorithm and assumptions can be found here. (Note: The values in the table titled "Defaulted Inputs" are used in the algorithm and returned in the response of this method). An example of using this method in practice can be found in *Example Usage*.

Required Attributes

Table 5: Minimum required attributes

Field	Type	Description
cell_technology_type	int	Represents the cell technology type (CdTe, poly c-Si PERC, etc). Use `plantpredict.enumerations.CellTechnologyTypeEnum`.
pv_model	int	Represents the 1-diode model type (1-Diode, 1-Diode with recombination). Use `plantpredict.enumerations.PVModelTypeEnum`.
number_of_cells_in_series	int	Number of cells in one string of cells - unitless
reference_irradiance	float	Must be between `400.0` and `1361.0` - units `[W/m^2]`. However, the calculation is always made at `1000 W/m^2`.
reference_temperature	float	Must be between `-20.0` and `80.0` - units `[deg-C]`. However, the calculation is always made at `25 deg-C`.
stc_max_power	float	Must be between `0.0` and `1000.0` - units `[W]`.
stc_short_circuit_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_open_circuit_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_mpp_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_mpp_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_power_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
stc_short_circuit_current_temp_coef	float	Must be between `-0.3` and `2.0` - units `[%/deg-C]`.

Generated Parameters

Table 6: Generated Parameters

Field	Type	Description
series_resistance_at_stc	float	units `[Ohms]`
maximum_series_resistance	float	units `[Ohms]`
recombination_parameter	float	units `[V]`
maximum_recombination_parameter	float	units `[V]`
shunt_resistance_at_stc	float	units `[Ohms]`
exponential_dependency_on_shunt_resistance	float	Defaulted to 5.5 - unitless
dark_shunt_resistance	float	units `[Ohms]`
saturation_current_at_stc	float	units `[A]`
diode_ideality_factor_at_stc	float	unitless
linear_temp_dependence_on_gamma	float	units `[%/deg-C]`
light_generated_current	float	units `[A]`

Returns Dictionary mirroring local module object with newly generated parameters.

Return type dict

generate_single_diode_parameters_advanced(***kwargs*)

POST */Module/Generator/GenerateSingleDiodeParametersAdvanced*

Solves for unknown single-diode parameters from module electrical characteristics and known single-diode parameters. This method is considered "advanced" because it requires more inputs to generate the remaining single-diode parameters. Whereas, the "default" method `plantpredict.Module.generate_single_diode_parameters_default()` is relatively basic in that it requires less inputs and automatically calculates more of the parameters. An example of using this method in practice can be found in *Example Usage*.

Required Attributes

Table 7: Minimum required attributes

Field	Type	Description
cell_technology_type	int	Represents the cell technology type (CdTe, poly c-Si PERC, etc). Use *plantpredict.enumerations.CellTechnologyTypeEnum*.
pv_model	int	Represents the 1-diode model type (1-Diode, 1-Diode with recombination). Use *plantpredict.enumerations.PVModelTypeEnum*.
number_of_cells_in_series	int	Number of cells in one string of cells - unitless
reference_irradiance	float	Must be between `400.0` and `1361.0` - units `[W/m^2]`. However, the calculation is always made at `1000 W/m^2`.
reference_temperature	float	Must be between `-20.0` and `80.0` - units `[deg-C]`. However, the calculation is always made at `25 deg-C`.
stc_max_power	float	Must be between `0.0` and `1000.0` - units `[W]`.
stc_short_circuit_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_open_circuit_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_mpp_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_mpp_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_power_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
stc_short_circuit_current_temp_coef	float	Must be between `-0.3` and `2.0` - units `[%/deg-C]`.
series_resistance_at_stc	float	Must be between `0.0` and `100.0` - units `[Ohms]`
shunt_resistance_at_stc	float	Must be between `0.0` and `100000.0` - units `[Ohms]`.
dark_shunt_resistance	float	Must be between `100.0` and `100000.0` - units `[Ohms]`.
recombination_parameter	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `30.0`
exponential_dependency_on_shunt_resistance	float	Must be between `1.0` and `100.0` - unitless.
bandgap_voltage	float	Must be between `0.5` and `4.0` - units `[V]`.
built_in_voltage	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `3.0` - units `[V]`.

Generated Parameters

Table 8: Generated Parameters

Field	Type	Description
maximum_series_resistance	float	units [Ohms]
maximum_recombination_parameter	float	units [V]
saturation_current_at_stc	float	units [A]
diode_ideality_factor_at_stc	float	unitless
linear_temp_dependence_on_gamma	float	units [%/deg-C]
light_generated_current	float	units [A]

Returns Dictionary mirroring local module object with newly generated parameters.

Return type dict

calculate_effective_irradiance_response(***kwargs*)
POST */Module/Generator/CalculateEffectiveIrradianceResponse*

Calculates the relative efficiency for any number of irradiance conditions with respect to performance at `1000 W/m^2` for a given temperature. Detailed documentation on this calculation can be found here. Unlike other of the `plantpredict.Module` methods related to generating module files, this method only returns a dictionary, and does not also auto-assign any attributes to the local object.

Required Attributes

Table 9: Minimum required attributes

Field	Type	Description
effective_irradiance_response	list of dict	Contains irradiance/temperature conditions at which to calculate relative efficiency. See example code below for usage.
cell_technology_type	int	Represents the cell technology type (CdTe, poly c-Si PERC, etc). Use *plantpredict.enumerations.CellTechnologyTypeEnum*.
pv_model	int	Represents the 1-diode model type (1-Diode, 1-Diode with recombination). Use *plantpredict.enumerations.PVModelTypeEnum*.
number_of_cells_in_series	int	Number of cells in one string of cells - unitless
reference_irradiance	float	Must be between `400.0` and `1361.0` - units `[W/m^2]`. However, only the irradiance values provided in :py:attr:'effective_irradiance_response are used in this calculation.
reference_temperature	float	Must be between `-20.0` and `80.0` - units `[deg-C]`. However, only the temperature values provided in :py:attr:'effective_irradiance_response are used in this calculation.
stc_max_power	float	Must be between `0.0` and `1000.0` - units `[W]`.
stc_short_circuit_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_open_circuit_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_mpp_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_mpp_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_power_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
stc_short_circuit_current_temp_coef	float	Must be between `-0.3` and `2.0` - units `[%/deg-C]`.
series_resistance_at_stc	float	Must be between `0.0` and `100.0` - units `[Ohms]`
shunt_resistance_at_stc	float	Must be between `0.0` and `100000.0` - units `[Ohms]`.
dark_shunt_resistance	float	Must be between `100.0` and `100000.0` - units `[Ohms]`.
recombination_parameter	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `30.0`
exponential_dependency_on_shunt_resistance	float	Must be between `1.0` and `100.0` - unitless.
bandgap_voltage	float	Must be between `0.5` and `4.0` - units `[V]`.
built_in_voltage	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `3.0` - units `[V]`.
saturation_current_at_stc	float	Must be between `1e-13` and `1e-6` - units `[A]`.
diode_ideality_factor_at_stc	float	Must be between `0.1` and `5.0` - unitless.
linear_temp_dependence_on_gamma	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
light_generated_current	float	Must be between `0.1` and `100.0` - units `[A]`

Example Code

First, import the plantpredict library and create an instance of *plantpredict.api.Api* in your Python session, to authenticate as shown in Step 3 of *API Authentication*. Then instantiate a local *plantpredict.module.Module* object as shown in previous examples. Then, assuming that all of the other required attributes have been assigned to the local object, assign the attribute `effective_irradiance_response` as follows (this determines which conditions the relative efficiencies will be calculated at):

```
module.effective_irradiance_response = [
    {'temperature': 25, 'irradiance': 1000},
    {'temperature': 25, 'irradiance': 800},
    {'temperature': 25, 'irradiance': 600},
    {'temperature': 25, 'irradiance': 400},
    {'temperature': 25, 'irradiance': 200}
]
```

Important note: For each dictionary in :py:attr:'effective_irradiance_response', there is an optional field (in addition to py:attr:'temperature' and py:attr:'irradiance'), :py:attr:'relative_efficiency'. For this method, that field does not have to be defined - it is used for :py:meth:'optimize_series_resistance' to be used as a target for tuning the series resistance. The EIR calculated by this method will be different from the target. In the context of creating a new module file, a user would probably want to compare the model-calculated EIR (determined from this method), to the target relative efficiencies in :py:attr:'effective_irradiance_response', which is why they have :py:attr:'temperature' and :py:attr:'irradiance' in common.

Call this method to generate the model-calculated effective irradiance response.

```
module.calculate_effective_irradiance_response()
```

Which returns the following sample response (a relative efficiency of 0.99 represents 99% or -1% efficiency relative to `[W/m^2]` at the same temperature):

```
[
    {'temperature': 25, 'irradiance': 1000, 'relative_efficiency': 1.0},
    {'temperature': 25, 'irradiance': 800, 'relative_efficiency': 1.02},
    {'temperature': 25, 'irradiance': 600, 'relative_efficiency': 1.001},
    {'temperature': 25, 'irradiance': 400, 'relative_efficiency': 0.99},
    {'temperature': 25, 'irradiance': 200, 'relative_efficiency': 0.97}
]
```

Returns A list of dictionaries containing the calculated relative efficiencies (see Example Code above).

Return type list of dict

optimize_series_resistance(***kwargs*)

POST */Module/Generator/OptimizeSeriesResistance*

While this method can be called independently, it is most commonly used after first calling `plantpredict.Module.generate_single_diode_parameters_advanced()` or `plantpredict.Module.generate_single_diode_parameters_default()`. Automatically "tunes" `series_resistance_at_stc` to bring the model-calculated effective irradiance (EIR) response close to a user-specified target EIR. Also recalculates single-diode parameters dependent on `series_resistance_at_stc`. Detailed documentation on the algorithm used to accomplish this can be found here. An example of using this method in practice can be found in *Example Usage*.

Required Attributes

Table 10: Minimum required attributes

Field	Type	Description
effective_irradiance_response	list of dict	List of dictionaries each containing temperature, irradiance, and the target efficiency relative to STC at those conditions.
cell_technology_type	int	Represents the cell technology type (CdTe, poly c-Si PERC, etc). Use *plantpredict.enumerations.CellTechnologyTypeEnum*.
pv_model	int	Represents the 1-diode model type (1-Diode, 1-Diode with recombination). Use *plantpredict.enumerations.PVModelTypeEnum*.
number_of_cells_in_series	int	Number of cells in one string of cells - unitless
reference_irradiance	float	Must be between `400.0` and `1361.0` - units `[W/m^2]`. While required, this value isn't used in the calculation.
reference_temperature	float	Must be between `-20.0` and `80.0` - units `[deg-C]`. While required, this value isn't used in the calculation.
stc_max_power	float	Must be between `0.0` and `1000.0` - units `[W]`.
stc_short_circuit_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_open_circuit_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_mpp_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_mpp_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_power_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
stc_short_circuit_current_temp_coef	float	Must be between `-0.3` and `2.0` - units `[%/deg-C]`.
series_resistance_at_stc	float	Must be between `0.0` and `100.0` - units `[Ohms]`
shunt_resistance_at_stc	float	Must be between `0.0` and `100000.0` - units `[Ohms]`.
dark_shunt_resistance	float	Must be between `100.0` and `100000.0` - units `[Ohms]`.
recombination_parameter	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `30.0`
exponential_dependency_on_shunt_resistance	float	Must be between `1.0` and `100.0` - unitless.
bandgap_voltage	float	Must be between `0.5` and `4.0` - units `[V]`.
built_in_voltage	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `3.0` - units `[V]`.
saturation_current_at_stc	float	Must be between `1e-13` and `1e-6` - units `[A]`.
diode_ideality_factor_at_stc	float	Must be between `0.1` and `5.0` - unitless.
linear_temp_dependence_on_gamma	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
light_generated_current	float	Must be between `0.1` and `100.0` - units `[A]`

Returns Dictionary mirroring local module object with newly generated parameters.

Return type dict

process_key_iv_points(***kwargs*)

POST */Module/Generator/ProcessKeyIVPoints*

Processes "Key IV Points" data, either from a `file template` or similar data structure. This is used as a pre-processing step to module file generation - it returns the minimum required fields for `plantpredict.generate_single_diode_parameters_default()`. It also automatically assigns the resulting attributtes to the local object instance of `plantpredict.Module`. Detailed algorithmic documentation for this method can be found here. See "Example Code" below for sample usage and "Generated Parameters" for the resulting attributes assigned to the local object instance.

Example Code

If the user is using the Excel template, usage of this method is straightforward:

```
module = plantpredict.Module()
module.process_key_iv_points(file_path="path_to_key_iv_points_template.xlsx")
```

However, the user can also manually construct the data structure and call the method as follows:

```
input_data = [
    {
        "temperature": 15,
        "irradiance": 1000,
        "short_circuit_current": 0.174,
        "open_circuit_voltage": 83.71,
        "mpp_current": 0.150,
        "mpp_voltage": 70.28,
        "max_power": 10.56,
    },
    {
        "temperature": 25,
        "irradiance": 1000,
        "short_circuit_current": 1.749,
        "open_circuit_voltage": 89.71,
        "mpp_current": 1.590,
        "mpp_voltage": 72.04,
        "max_power": 114.52,
    },
    {
        "temperature": 25,
        "irradiance": 800,
        "short_circuit_current": 1.399,
        "open_circuit_voltage": 88.85,
        "mpp_current": 1.272,
        "mpp_voltage": 72.20,
        "max_power": 91.85,
    },
    {
        "temperature": 25,
        "irradiance": 600,
        "short_circuit_current": 1.049,
        "open_circuit_voltage": 87.75,
        "mpp_current": 0.951,
        "mpp_voltage": 72.75,
        "max_power": 68.68,
    },
    {
        "temperature": 25,
        "irradiance": 400,
        "short_circuit_current": 0.700,
        "open_circuit_voltage": 86.27,
        "mpp_current": 0.630,
        "mpp_voltage": 71.92,
```

(continues on next page)

(continued from previous page)

```
        "max_power": 45.29,
    },
    {
        "temperature": 25,
        "irradiance": 200,
        "short_circuit_current": 0.350,
        "open_circuit_voltage": 83.67,
        "mpp_current": 0.311,
        "mpp_voltage": 70.32,
        "max_power": 21.88,
    },
        "temperature": 50,
        "irradiance": 1000,
        "short_circuit_current": 1.768,
        "open_circuit_voltage": 83.05,
        "mpp_current": 1.599,
        "mpp_voltage": 65.71,
        "max_power": 105.07,
    }
]

module.process_key_iv_points(key_iv_points_data=input_data)
```

While the only *required* temperature/irradiance conditions is STC (25 deg-C / 1000 W/m^2), more input data is required to generate temperature coefficients and effective irradiance response (see Generated Parameters).

Generated Parameters

The following parameters are automatically assigned as attributes to the local instance of `plantpredict.Module`.

Table 11: Generated Parameters

Field	Type	Description
stc_short_circuit_current	float	Always returned with minimum required input (data at STC) - units `[A]`
stc_open_circuit_voltage	float	Always returned with minimum required input (data at STC) - units `[V]`
stc_mpp_current	float	Always returned with minimum required input (data at STC) - units `[A]`
stc_mpp_voltage	float	Always returned with minimum required input (data at STC) - units `[V]`
stc_max_power	float	Always returned with minimum required input (data at STC) - units `[W]`
stc_short_circuit_current_temp_coef	float	Only returned if data provided at `1000 W/m^2` and at least one temperature other than `25 deg-C` - units `[%/deg-C]`
stc_open_circuit_voltage_temp_coef	float	Only returned if data provided at `1000 W/m^2` and at least one temperature other than `25 deg-C` - units `[%/deg-C]`
stc_power_temp_coef	float	Only returned if data provided at `1000 W/m^2` and at least one temperature other than `25 deg-C` - units `[%/deg-C]`
effective_irradiance_response	dict	Only returned if data provided at multiple irradiances for a single temperature - see example output below for contents.

The following is an example of the dictionary output (mirrors "Generated Parameters").

```
{
    "stc_short_circuit_current": 1.7592,
    "stc_open_circuit_voltage": 90.2189,
    "stc_mpp_current": 1.6084,
    "stc_mpp_voltage": 72.4938,
    "stc_short_circuit_current_temp_coef": 0.0519,
    "stc_open_circuit_voltage_temp_coef": -0.3081,
    "stc_power_temp_coef": -0.3535,
    "effective_irradiance_response": [
        {"temperature": 25, "irradiance": 1000, "relative_efficiency": 1.0},
        {"temperature": 25, "irradiance": 800, "relative_efficiency": 1.0039},
        {"temperature": 25, "irradiance": 600, "relative_efficiency": 1.0032},
        {"temperature": 25, "irradiance": 400, "relative_efficiency": 0.9925},
        {"temperature": 25, "irradiance": 200, "relative_efficiency": 0.9582},
    ]
}
```

Parameters

- **file_path** (*str*) – File path to the .xlsx template for Key IV Points (input option 1).
- **key_iv_points_data** (*lists of dict*) – List of dictionaries containing module electrical characteristics at STC and other temperature/irradiance conditions (input option 2).

Returns Dictionary containing STC electrical parameters, temperature coefficients, and effective irradiance response, depending on the scope of the input data provided (see "Generated Parameters" above).

Return type dict

process_iv_curves(***kwargs*)

POST */Module/Generator/ProcessIVCurves*

Processes any number of full IV Curve measurements, either from a `file template` or similar data structure. This is used as a pre-processing step to module file generation - it returns the extracted electrical characteristics at the set of temperature/irradiance conditions corresponding to those of the provided IV Curves. The output data structure matches the exact data input structure for `plantpredict.Module.process_key_iv_points()`. (The methods are meant to be used in succession in order to effectively extract electrical characteristics at STC, temperature coefficients, and effective irradiance response from a set of IV curves). Detailed algorithmic documentation for this method can be found here. See "Example Code" below for sample usage.

Example Code

If the user is using the Excel template, usage of this method is straightforward:

```
module = plantpredict.Module()
module.process_iv_curves(file_path="path_to_iv_curves_template.xlsx")
```

However, the user can also manually construct the data structure and call the method as follows:

```
input_data = [
    {
        "temperature": 25,
        "irradiance": 1000,
        "data_points": [
            {"current": 9.43, "voltage": 0.0},
```

(continues on next page)

(continued from previous page)

```
            # ... insert at least 40 total IV points ...
            {"current": 0.0, "voltage": 46.39}
        ]
    }
]

module.process_iv_curves(iv_curve_data=input_data)
```

Which will return a data structure:

```
[
    {
        "temperature": 25,
        "irradiance": 1000,
        "short_circuit_current": 9.43,
        "open_circuit_voltage": 46.39,
        "mpp_current": 8.9598,
        "mpp_voltage": 38.1285,
        "max_power": 341.6237
    }
]
```

Reminder: While only one IV curve is provided in the example, multiply IV curves can be supplied.

Parameters

- **file_path** (*str*) – File path to the .xlsx template for Full IV Curves. (At least 40 points are required for each IV curve.)
- **iv_curve_data** (*list dict*) – List of dictionaries, each representing an IV curve at a particular temperature/irradiance. (At least 40 points are required for each IV curve.)

Returns List of dictionaries, each containing extracted module electrical characteristics corresponding to the IV curve provided at a particular temperature/irradiance condition.

Return type list of dict

generate_iv_curve(***kwargs*)

POST */Module/Generator/GenerateIVCurve*

Generates an IV curve given An example of using this method in practice can be found in *Example Usage*.

Required Attributes

Table 12: Minimum required attributes

Field	Type	Description
cell_technology_type	int	Represents the cell technology type (CdTe, poly c-Si PERC, etc). Use *plantpredict.enumerations.CellTechnologyTypeEnum*.
pv_model	int	Represents the 1-diode model type (1-Diode, 1-Diode with recombination). Use *plantpredict.enumerations.PVModelTypeEnum*.
number_of_cells_in_series	int	Number of cells in one string of cells - unitless
reference_irradiance	float	Must be between `400.0` and `1361.0` - units `[W/m^2]`. The IV curve will represent this irradiance.
reference_temperature	float	Must be between `-20.0` and `80.0` - units `[deg-C]`. The IV curve will represent this temperature.
stc_max_power	float	Must be between `0.0` and `1000.0` - units `[W]`.
stc_short_circuit_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_open_circuit_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_mpp_current	float	Must be between `0.1` and `100.0` - units `[A]`.
stc_mpp_voltage	float	Must be between `0.4` and `1000.0` - units `[V]`.
stc_power_temp_coef	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
stc_short_circuit_current_temp_coef	float	Must be between `-0.3` and `2.0` - units `[%/deg-C]`.
series_resistance_at_stc	float	Must be between `0.0` and `100.0` - units `[Ohms]`
shunt_resistance_at_stc	float	Must be between `0.0` and `100000.0` - units `[Ohms]`.
dark_shunt_resistance	float	Must be between `100.0` and `100000.0` - units `[Ohms]`.
recombination_parameter	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `30.0`
exponential_dependency_on_shunt_resistance	float	Must be between `1.0` and `100.0` - unitless.
bandgap_voltage	float	Must be between `0.5` and `4.0` - units `[V]`.
built_in_voltage	float	Required only if `pv_model` is *plantpredict.enumerations.PVModelTypeEnum.ONE_DIODE_RECOMBINATION*. Must be between `0.0` and `3.0` - units `[V]`.
saturation_current_at_stc	float	Must be between `1e-13` and `1e-6` - units `[A]`.
diode_ideality_factor_at_stc	float	Must be between `0.1` and `5.0` - unitless.
linear_temp_dependence_on_gamma	float	Must be between `-3.0` and `3.0` - units `[%/deg-C]`.
light_generated_current	float	Must be between `0.1` and `100.0` - units `[A]`

Example Output

```
[
    {"current": 9.43, "voltage": 0.0},
    # ... list will be equal in length to num_iv_points ...
    {"current": 0.0, "voltage": 46.39}
]
```

Parameters **num_iv_points** (*int*) – Number of IV points to generate (defaults to 100).

Returns List of IV generated IV points (See "Example Output")

Return type list

calculate_basic_data_at_conditions(***kwargs*)

Returns

Inverter

class plantpredict.inverter.**Inverter**(*api*, ***kwargs*)
Bases: `plantpredict.plant_predict_entity.PlantPredictEntity`

create()
POST /Inverter

delete()
DELETE /Inverter/{Id}

get()
GET /Inverter/{Id}

update()
PUT /Inverter

upload_ond_file(***kwargs*)
creates a new inverter from a source .ond file

parse_ond_file(***kwargs*)
creates a new inverter from a source .ond file

create_from_json(***kwargs*)
creates a new inverter from a source JSON file

change_status(***kwargs*)

Parameters

- **new_status** –
- **note** –

Returns

get_kva(***kwargs*)
Uses the given elevation and temperature to interpolate a kVa rating from the inverter's kVa curves.

Parameters

- **elevation** (*float*) – Elevation at which to evaluate the inverter kVa rating - units `[m]`.
- **temperature** (*float*) – Temperature at which to evaluate the inverter kVa rating - units `[deg-C]`.
- **use_cooling_temp** (*bool*) – Determines if the calculation should use the plant design cooling temperature (at 99.6 degrees).

Returns # TODO after new API response is implemented

get_inverter_list(***kwargs*)

Returns a list of all inverter to which a user has access.

Geo

class plantpredict.geo.**Geo**(*api*, *latitude=None*, *longitude=None*)

Bases: `object`

The `Geo` entity is used to get location-related information for a given latitude/longitude. Its methods can be used individually, but typically location related info is needed in the context of a PlantPredict `Project` entity. In this case the user can simply call the method `Project.get_location_info()` which calls all `Geo` class methods and automatically assigns all location-related attributes to that instance of `Project`. Note: This API resource does not represent a database entity in PlantPredict. This is a simplified connection to the Google Maps API. See Google Maps API Reference for further functionality. (https://developers.google.com/maps/)

get_location_info(***kwargs*)

GET */Geo/* `latitude` / `longitude` */Location*

Retrieves pertinent location info for a given latitude and longitude such as locality, state/province, country, etc. In addition to returning a dictionary with this information, the method also automatically assigns the contents of the dictionary to the instance of `Geo` as attributes.

Required Attributes

Table 13: Minimum required attributes on object to call this method successfully.

Field	Type	Description
latitude	float	North-South GPS coordinate. Must be between `-90` and `90` - units `[decimal degrees]`.
longitude	float	East-West GPS coordinate Must be between `-180` and `180` units `[decimal degrees]`.

Example Code

Instantiate a local object instance of `Geo` with latitude and longitude as inputs (which automatically assigns them as attributes to the object). Then call the method on the object.

```
geo = api.geo(latitude=35.1, longitude=-106.7)
geo.get_location_info()
```

Example Response

The method returns a dictionary as shown in the example below, and assigns its contents as attributes to the local object instance of `Geo`.

```
{
    "country": "United States",
    "country_code": "US",
    "locality": "Albuquerque",
    "region": "North America",
    "state_province": "New Mexico",
    "state_province_code": "NM"
}
```

Returns A dictionary with location information as shown in "Example Response".

Return type dict

get_elevation(***kwargs*)

GET */Geo/* `latitude` / `longitude` */Elevation*

Retrieves the elevation in meters for a given latitude and longitude. In addition to returning a dictionary with this information, the method also automatically assigns the contents of the dictionary to the instance of `Geo` as attributes.

Required Attributes

Table 14: Minimum required attributes on object to call this method successfully.

Field	Type	Description
latitude	float	North-South GPS coordinate. Must be between `-90` and `90` - units `[decimal degrees]`.
longitude	float	East-West GPS coordinate Must be between `-180` and `180` units `[decimal degrees]`.

Example Code

Instantiate a local object instance of `Geo` with latitude and longitude as inputs (which automatically assigns them as attributes to the object). Then call the method on the object.

```
geo = api.geo(latitude=35.1, longitude=-106.7)
geo.get_elevation()
```

Example Response

The method returns a dictionary as shown in the example below, and assigns its contents as attributes to the local object instance of `Geo`.

```
{
    "elevation": 1553.614
}
```

Returns A dictionary with location information as shown in "Example Response".

Return type dict

get_time_zone(***kwargs*)

GET */Geo/* `latitude` / `longitude` */TimeZone*

Retrieves the time zone as a time shift in hours with respect to GMT for a given latitude and longitude. In addition to returning a dictionary with this information, the method also automatically assigns the contents of the dictionary to the instance of `Geo` as attributes.

Required Attributes

Table 15: Minimum required attributes on object to call this method successfully.

Field	Type	Description
latitude	float	North-South GPS coordinate. Must be between `-90` and `90` - units `[decimal degrees]`.
longitude	float	East-West GPS coordinate Must be between `-180` and `180` units `[decimal degrees]`.

Example Code

Instantiate a local object instance of `Geo` with latitude and longitude as inputs (which automatically assigns them as attributes to the object). Then call the method on the object.

```
geo = api.geo(latitude=35.1, longitude=-106.7)
geo.get_time_zone()
```

Example Response

The method returns a dictionary as shown in the example below, and assigns its contents as attributes to the local object instance of `Geo`.

```
{
    "time_zone": -7.0
}
```

Returns A dictionary with location information as shown in "Example Response".

Return type dict

ASHRAE

class `plantpredict.ashrae.`**`ASHRAE`**(*api*, *latitude=None*, *longitude=None*, *station_name=None*)

Bases: `object`

The *ASHRAE* class is used to get key information for an ASHRAE station. It can be used on its own for any application, but mostly exists to find and assign plant design temperatures for a particular location to a *Prediction*.

`get_station`(***kwargs*)

Returns the ASHRAE station matching the specified name and shortest distance from the specified latitude and longitude. Sets the returned information as attributes on the instance of this class.

Parameters **`station_name`** (*str*) – Valid name of ASHRAE weather station

Returns # TODO once new http response is implemented

`get_closest_station`(***kwargs*)

Returns the ASHRAE station with the shortest distance from the specified latitude and longitude. Sets the returned information as attributes on the instance of this class.

Returns # TODO once new http response is implemented

1.3.2 Helpers

`plantpredict.helpers.`**`load_from_excel`**(*file_path*, *sheet_name=None*)

Loads the data from an Excel file into a list of dictionaries, where each dictionary represents a row in the Excel file and the keys of each dictionary represent each column header in the Excel file. The method creates this list of dictionaries via a Pandas dataframe.

Parameters

- **`file_path`** (*str*) – The full file path (appended with .xlsx) of the Excel file to be loaded.
- **`sheet_name`** – Name of a particular sheet in the file to load (optional, defaults to the first sheet in the

Excel file). :type sheet_name: str :return: List of dictionaries, each dictionary representing a row in the Excel file. :rtype: list of dict `plantpredict.helpers.`**`export_to_excel`**(*data*, *file_path*, *sheet_name='Sheet1'*, *field_order=None*, *sorting_fields=None*)

Writes data from a list of dictionaries to an Excel file, where each dictionary represents a row in the Excel file and the keys of each dictionary represent each column header in the Excel file.

Parameters

- **`data`** (*`list of dict`*) – List of dictionaries, each dictionary representing a row in the Excel file.
- **`file_path`** – The full file path (appended with .xlsx) of the Excel file to be written to. This will overwrite

data if both file_path and sheet_name already exist. :type file_path: str :param sheet_name: Name of a particular sheet in the file to write to (optional, defaults to "Sheet1"). :type sheet_name: str :param field_order: List of keys from data ordered to match the intended Excel column ordering (left to right). Must include all keys/columns. Any keys omitted from the list will not be written as columns. (optional) :type field_order: list of str :param sorting_fields: List of keys from data to be used as sorting columns (small to large) in Excel. Can be any length from 1 column to every column. The order of the list will dictate the sorting order. :type sorting_fields: list of str :return: None

1.3.3 Data Enumerations

class `plantpredict.enumerations.`**`AirMassModelTypeEnum`**

Air Mass Model

`BIRD_HULSTROM = 0`

`KASTEN_SANDIA = 1`

class `plantpredict.enumerations.`**`BacktrackingTypeEnum`**

Backtracking Type

`TRUE_TRACKING = 0`

`BACKTRACKING = 1`

class `plantpredict.enumerations.`**`CellTechnologyTypeEnum`**

Cell Technology

`NTYPE_MONO_CSI = 1`

`PTYPE_MONO_CSI_PERC = 2`

`PTYPE_MONO_CSI_BSF = 3`

`POLY_CSI_PERC = 4`

`POLY_CSI_BSF = 5`

`CDTE = 6`

`CIGS = 7`

class `plantpredict.enumerations.`**`CleaningFrequencyEnum`**

Cleaning Frequency

`NONE = 0`

`DAILY = 1`

`MONTHLY = 2`

`QUARTERLY = 3`

YEARLY = 4

class plantpredict.enumerations.**ConstructionTypeEnum**
Construction Type

GLASS_GLASS = 1

GLASS_BACKSHEET = 2

class plantpredict.enumerations.**DataSourceEnum**
Data Source

MANUFACTURER = 1

PVSYST = 2

UNIVERSITY_OF_GENEVA = 3

PHOTON = 4

SANDIA_DATABASE = 5

CUSTOM = 6

class plantpredict.enumerations.**DegradationModelEnum**
Degradation Model

NONE = 0

STEPPED_AC = 1

LINEAR_AC = 2

LINEAR_DC = 3

NON_LINEAR_DC = 4

class plantpredict.enumerations.**DiffuseDirectDecompositionModelEnum**
Diffuse Direct Decomposition Model

ERBS = 0

REINDL = 1

DIRINT = 2

NONE = 3

class plantpredict.enumerations.**DiffuseShadingModelEnum**
Diffuse Shading Model

NONE = 0

SCHAAR_PANCHULA = 1

class plantpredict.enumerations.**DirectBeamShadingModelEnum**
Direct Beam Shading Model

LINEAR = 0

NONE = 1

TWO_DIMENSION = 2

FRACTIONAL_EFFECT = 3

CSI_3_DIODE = 4

MODULE_FILE_DEFINED = 5

class plantpredict.enumerations.**EntityTypeEnum**
Entity Type

PROJECT = 1

MODULE = 2

INVERTER = 3

WEATHER = 4

PREDICTION = 5

class plantpredict.enumerations.**ESSChargeAlgorithmEnum**
Energy Storage System (ESS) Charge Algorithm

LGIA_EXCESS = 0

ENERGY_AVAILABLE = 1

CUSTOM = 2

class plantpredict.enumerations.**ESSDispatchCustomCommandEnum**
Energy Storage System (ESS) Dispatch Custom Command

NONE = 0

DISCHARGE = 1

CHARGE = 2

class plantpredict.enumerations.**FacialityEnum**
Faciality

MONOFACIAL = 0

BIFACIAL = 1

class plantpredict.enumerations.**IncidenceAngleModelTypeEnum**
Incidence Angle Model Type

SANDIA = 2

ASHRAE = 3

NONE = 4

TABULAR_IAM = 5

PHYSICAL = 6

class plantpredict.enumerations.**LibraryStatusEnum**
Library Status (for Module, Inverter, Weather)

UNKNOWN = 0

DRAFT_PRIVATE = 1

DRAFT_SHARED = 2

ACTIVE = 3

RETIRED = 4

GLOBAL = 5

GLOBAL_RETIRED = 6

class plantpredict.enumerations.**ModuleDegradationModelEnum**
Module Degradation Model

UNSPECIFIED = 0

LINEAR = 1

NONLINEAR = 2

class plantpredict.enumerations.**ModuleOrientationEnum**
Module Orientation

LANDSCAPE = 0

PORTRAIT = 1

class plantpredict.enumerations.**ModuleShadingResponseEnum**
Module Shading Response

NONE = 0

LINEAR = 1

FRACTIONAL_EFFECT = 2

CSI_3_DIODE = 3

CUSTOM = 4

class plantpredict.enumerations.**ModuleTemperatureModelEnum**
Module Temperature Model

HEAT_BALANCE = 0

SANDIA = 1

NOCT = 2

class plantpredict.enumerations.**ModuleTypeEnum**
Module Type

SINGLE_DIODE = 0

ADVANCED_DIODE = 1

class plantpredict.enumerations.**PerezModelCoefficientsEnum**
Perez Coefficients

PLANT_PREDICT = 0

ALL_SITES_COMPOSITE_1990 = 1

ALL_SITES_COMPOSITE_1988 = 2

SANDIA_COMPOSITE_1988 = 3

USA_COMPOSITE_1988 = 4

FRANCE_1988 = 5

PHOENIX_1988 = 6

ELMONTE_1988 = 7

OSAGE_1988 = 8

ALBUQUERQUE_1988 = 9

CAPE_CANAVERAL_1988 = 10

ALBANY_1988 = 11

class plantpredict.enumerations.**PredictionStatusEnum**
Prediction Status

DRAFT_PRIVATE = 1

DRAFT_SHARED = 2

ANALYSIS = 3

BID = 4

CONTRACT = 5

DEVELOPMENT = 6

AS_BUILT = 7

WARRANTY = 8

ARCHIVED = 9

class plantpredict.enumerations.**PredictionVersionEnum**
Prediction Version

VERSION_3 = 3

VERSION_4 = 4

VERSION_5 = 5

VERSION_6 = 6

VERSION_7 = 7

VERSION_8 = 8

VERSION_9 = 9

VERSION_10 = 10

VERSION_11 = 11

class plantpredict.enumerations.**ProcessingStatusEnum**
Processing Status

NONE = 0

QUEUED = 1

RUNNING = 2

SUCCESS = 3

ERROR = 4

class plantpredict.enumerations.**ProjectStatusEnum**
Project Status

ACTIVE = 0

ARCHIVED = 1

class plantpredict.enumerations.**PVModelTypeEnum**
PV Model

ONE_DIODE_RECOMBINATION = 0

ONE_DIODE = 1

ONE_DIODE_RECOMBINATION_NONLINEAR = 3

class plantpredict.enumerations.**SoilingModelTypeEnum**
Soiling Model

CONSTANT_MONTHLY = 0

WEATHER_FILE = 1

NONE = 2

class plantpredict.enumerations.**SpectralShiftModelEnum**
Spectral Shift Model

NO_SPECTRAL_SHIFT = 0

ONE_PARAM_PWAT_OR_SANDIA = 1

TWO_PARAM_PWAT_AND_AM = 2

MONTHLY_OVERRIDE = 3

class plantpredict.enumerations.**SpectralWeatherTypeEnum**
Spectral Weather Type

NONE = 0

NGAN_PWAT = 1

NGAN_RH = 2

NGAN_DEWPOINT = 3

class plantpredict.enumerations.**TrackingTypeEnum**
Tracking Type

FIXED_TILT = 0

HORIZONTAL_TRACKER = 1

SEASONAL_TILT = 2

class plantpredict.enumerations.**TranspositionModelEnum**
Transposition Model

HAY = 0

PEREZ = 1

class plantpredict.enumerations.**CircumsolarTreatmentTypeEnum**
Circumsolar Allocation Type

DIFFUSE = 0

DIRECT = 1

class plantpredict.enumerations.**WeatherDataProviderEnum**
Weather Data Provider

NREL = 1

AWS = 2

WIND_LOGICS = 3

METEONORM = 4

THREE_TIER = 5

CLEAN_POWER_RESEARCH = 6

GEO_MODEL_SOLAR = 7

GEO_SUN_AFRICA = 8

SODA = 9

HELIO_CLIM = 10

SOLAR_RESOURCE_ASSESSMENT = 11

ENERGY_PLUS = 12

OTHER = 13

CUSTOMER = 14

SOLAR_PROSPECTOR = 15

GLOBAL_FED = 16

NSRDB = 17

WHITE_BOX_TECHNOLOGIES = 18

SOLARGIS = 19

NASA = 20

THREE_TIER_VAISALA = 21

SOLCAST = 22

class plantpredict.enumerations.**WeatherDataTypeEnum**

Weather Data Type

SYNTHETIC_MONTHLY = 0

SATELLITE = 1

GROUND_CORRECTED = 2

MEASURED = 3

TMY3 = 4

TGY = 5

TMY = 6

PSM = 7

SUNY = 8

MTS2 = 9

CZ2010 = 10

class plantpredict.enumerations.**WeatherFileColumnTypeEnum**

Weather File Column Type

GHI = 1

DNI = 2

DHI = 3

TEMP = 4

WINDSPEED = 5

RELATIVE_HUMIDITY = 6

PWAT = 7

RAIN = 8

PRESSURE = 9

DEWPOINT_TEMP = 10

WIND_DIRECTION = 11

SOILING_LOSS = 12

POAI = 13

REAR_POAI = 14

class plantpredict.enumerations.**WeatherPLevelEnum**
Weather P-Level

P50 = 0

P90 = 1

P95 = 3

P99 = 4

NA = 2

P75 = 5

class plantpredict.enumerations.**WeatherSourceTypeAPIEnum**
Weather Source Type API (web-service downloadable vendors). This Enum is used when calling *download()*.

UNKNOWN = 0

METEONORM = 1

CPR_SOLAR_ANYWHERE = 2

NSRDB_PSM = 3

NSRDB_SUNY = 4

NSRDB_MTS2 = 5

SOLAR_GIS = 6

NASA = 7

class plantpredict.enumerations.**WeatherTimeResolution**
Weather Time Resolution

UNKNOWN = 0

HALF_HOUR = 1

HOUR = 2

MINUTE = 3

1.4 Example Usage

The code snippets below are practical examples of useful tasks accomplished via PlantPredict's API. All of the code used in the examples below is available via the source code on Github. Feel free to use and modify the code in your local environment.

Every example assumes that you first `import plantpredict` and authenticate with *Api* as shown in Step 3 of *API Authentication*.

1.4.1 Create Project and Prediction from scratch.

This is one example of how to build a project, prediction, and attach a power plant. There are a variety of optional settings for every component that can't be captured in a single example. Please refer to the documentation for *Project*, *Prediction*, and *PowerPlant* for more information.

Instantiate a local instance of *Project*, assigning `name`, `latitude`, and `longitude`.

```
project = api.project(name="Grand Canyon Power Plant", latitude=36.099, longitude=-
↪112.112)
```

Assign location attributes with helper method *assign_location_attributes()*, and create as the local instance of *Project* a new entity in the PlantPredict database.

```
project.assign_location_attributes()
project.create()
```

Instantiate a local instance of *Prediction*, assigning `project_id` (from the newly created project) and `name`.

```
prediction = api.prediction(project_id=project.id, name="Grand Canyon - Contracted")
```

Assign the `weather_id` corresponding to the weather file you want to use (assuming it already exists in the PlantPredict database).

```
prediction.weather_id = 13628
```

Instantiate and retrieve the weather file, and ensure that the two pairs of prediction start/end attributes match those of the weather file.

```
weather = api.weather(id=prediction.weather_id)
weather.get()
prediction.start_date = weather.start_date
prediction.end_date = weather.end_date
prediction.start = weather.start_date
prediction.end = weather.end_date
```

Import all of the enumeration files relevant to prediction settings. Set ALL of the following model options on the prediction using the enumerations library in *enumerations* similar to the code below, but to your preferences.

```
from plantpredict.enumerations import PredictionStatusEnum, TranspositionModelEnum,␣
↪SpectralShiftModelEnum, \
    DiffuseDirectDecompositionModelEnum, ModuleTemperatureModelEnum,␣
↪IncidenceAngleModelTypeEnum, \
    AirMassModelTypeEnum, DirectBeamShadingModelEnum, SoilingModelTypeEnum,␣
↪DegradationModelEnum, \
    TrackingTypeEnum, BacktrackingTypeEnum, DiffuseShadingModelEnum
```

(continues on next page)

(continued from previous page)

```
prediction.diffuse_direct_decomp_model = DiffuseDirectDecompositionModelEnum.NONE
prediction.transposition_model = TranspositionModelEnum.PEREZ
prediction.mod_temp_model = ModuleTemperatureModelEnum.HEAT_BALANCE
prediction.inc_angle_model = IncidenceAngleModelTypeEnum.TABULAR_IAM
prediction.spectral_shift_model = SpectralShiftModelEnum.TWO_PARAM_PWAT_AND_AM
prediction.air_mass_model = AirMassModelTypeEnum.BIRD_HULSTROM
prediction.direct_beam_shading_model = DirectBeamShadingModelEnum.LINEAR
prediction.diffuse_shading_model = DiffuseShadingModelEnum.SCHAAR_PANCHULA
prediction.soiling_model = SoilingModelTypeEnum.CONSTANT_MONTHLY
prediction.monthly_factors = [
    {"month": 1, "month_name": "Jan", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 2, "month_name": "Feb", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 3, "month_name": "Mar", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 4, "month_name": "Apr", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 5, "month_name": "May", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 6, "month_name": "Jun", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 7, "month_name": "Jul", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 8, "month_name": "Aug", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 9, "month_name": "Sep", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 10, "month_name": "Oct", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 11, "month_name": "Nov", "albedo": 0.2, "soiling_loss": 2.0},
    {"month": 12, "month_name": "Dec", "albedo": 0.2, "soiling_loss": 2.0},
]
prediction.diffuse_direct_decomp_model_executed = True
prediction.use_meteo_dni = False
prediction.use_meteo_poai = False
prediction.degradation_model = DegradationModelEnum.LINEAR_DC
prediction.linear_degradation_rate = 0.5
prediction.first_year_degradation = False
prediction.year_repeater = 3
```

Create the prediction in the PlantPredict database.

```
prediction.create()
```

Change the prediction's status to `plantpredict.enumerations.PredictionStatusEnum.DRAFT-SHARED` to make it accessible to other members of your team (or to another relevant status).

```
prediction.change_status(new_status=PredictionStatusEnum.DRAFT_SHARED, note="Changed␣
↪for tutorial.")
```

Instantiate a local instance of *PowerPlant*, assigning its `project_id` and `prediction_id`.

```
powerplant = api.powerplant(project_id=project.id, prediction_id=prediction.id)
```

Add a fixed tilt block, array, inverter, and dc field using *add_block()*, *add_array()*, *add_inverter()* and *add_dc_field()*, respectively. In this example, not all optional fields are used in this method. Refer to each method's documentation for information on what other power plant attributes can be configured. Additionally, refer to the PlantPredict User Guide for documentation on power plant hierarchy.

```
fixed_tilt_block_name = powerplant.add_block()
fixed_tilt_array_name = powerplant.add_array(
    block_name=fixed_tilt_block_name,
    transformer_enabled=False,
    repeater=3,
```

(continues on next page)

(continued from previous page)

```
    description="Arrays in north eastern section of plant."
)
fixed_tilt_inverter_name = powerplant.add_inverter(
    block_name=fixed_tilt_block_name,
    array_name=fixed_tilt_array_name,
    inverter_id=619,
    setpoint_kw=720.0,
    repeater=2
)
```

Assuming there is one DC field on the inverter, the field DC power can be calculated from a DC AC ratio. If there were two identical DC fields on a single inverter, you would use half of the number of strings. For irregular configurations, perform a custom calculation for number of strings in parallel and field dc power. Additionally, the post to post spacing can be calculated from GCR and some information about the module being used in the DC field. Use the helpers to prepare field DC power and post to post spacing, and then add the fixed tilt DC field.

```
field_dc_power = powerplant.calculate_field_dc_power_from_dc_ac_ratio(dc_ac_ratio=1.2,
↪ inverter_setpoint=720.0)
post_to_post_spacing = powerplant.calculate_post_to_post_spacing_from_gcr(ground_
↪coverage_ratio=0.40, module_id=298,
                                                                        modules_
↪high=4)

fixed_tilt_dc_field_name = powerplant.add_dc_field(
    block_name=fixed_tilt_block_name,
    array_name=fixed_tilt_array_name,
    inverter_name=fixed_tilt_inverter_name,
    module_id=298,
    tracking_type=TrackingTypeEnum.FIXED_TILT,
    modules_high=4,
    modules_wired_in_series=10,
    post_to_post_spacing=post_to_post_spacing,
    number_of_rows=10,
    field_dc_power=field_dc_power,
    module_tilt=30
)
```

You can continue to add new blocks, or even add arrays to blocks, inverters to arrays, etc. The code below is an example of adding a block with a DC field that uses single-axis tracking.

```
tracker_block_name = powerplant.add_block()
tracker_array_name = powerplant.add_array(
    block_name=tracker_block_name,
    transformer_enabled=False,
)
tracker_inverter_name = powerplant.add_inverter(
    block_name=tracker_block_name,
    array_name=tracker_array_name,
    inverter_id=619,
    setpoint_kw=720.0
)
```

Prepare the field DC power and post to post spacing for the tracker DC field, and then add it to the inverter.

```
field_dc_power = powerplant.calculate_field_dc_power_from_dc_ac_ratio(dc_ac_ratio=1.1,
↪ inverter_setpoint=720.0)
```

(continues on next page)

(continued from previous page)

```
post_to_post_spacing = powerplant.calculate_post_to_post_spacing_from_gcr(ground_
↪coverage_ratio=0.20, module_id=298,
                                                                          modules_
↪high=1)

tracker_dc_field_name = powerplant.add_dc_field(
    block_name=tracker_block_name,
    array_name=tracker_array_name,
    inverter_name=tracker_inverter_name,
    module_id=298,
    tracking_type=TrackingTypeEnum.HORIZONTAL_TRACKER,
    modules_high=1,
    modules_wired_in_series=10,
    post_to_post_spacing=post_to_post_spacing,
    number_of_rows=10,
    field_dc_power=field_dc_power,
    tracking_backtracking_type=BacktrackingTypeEnum.TRUE_TRACKING
)
```

Create the local instance of *PowerPlant* as a new entity in the PlantPredict database. Since the id's of the project and prediction created previously were assigned to the PowerPlant, it will automatically attach to the prediction in PlantPredict.

```
powerplant.create()
```

The prediction can now be run.

```
prediction.run()
```

1.4.2 Model System-Level of Power Plant (Transformer, Transmission, etc.)

This tutorial details how to model Total System Capacity, Transformers and Transmission Lines for a power plant/energy prediction. This can be done upon initial creation of a prediction from scratch (see the example for *Create Project and Prediction from scratch.*), but for the sake of example, we will consider the case of updating an existing power plant.

Instantiate a *PowerPlant*, specifying its `project_id` and `prediction_id` (visible in the URL of that prediction in a web browser ... `/projects/{project_id}/prediction/{id}`).

```
project_id = 13161   # CHANGE TO YOUR PROJECT ID
prediction_id = 147813   # CHANGE TO YOUR PREDICTION ID
powerplant = api.powerplant(project_id=project_id, prediction_id=prediction_id)
```

Retrieve the power plant's attributes.

```
powerplant.get()
```

Set the system `availability_loss` on the *PowerPlant* instance in units `[%]`.

```
powerplant.availability_loss = 1.7
```

Set the plant output (LGIA) limit in units `[MWac]`.

```
powerplant.lgia_limitation = 0.8
```

Add `transformers` and `transmission_lines`, specifying the `ordinal` (1-indexed) such that they are in the desired order (where 1 is closest to the physical output of the plant).

```
powerplant.add_transformer(rating=0.6, high_side_voltage=600, no_load_loss=1.1, full_
↪load_loss=1.7, ordinal=1)
powerplant.add_transmission_line(length=3, resistance=0.1, number_of_conductors_per_
↪phase=1, ordinal=2)
```

Call the *update()* method on the instance of *PowerPlant* to persist these changes to PlantPredict.

```
powerplant.update()
```

1.4.3 Download nodal data.

First, set up a dictionary containing the nodal data export options. Set the values to True according to which nodes in the *PowerPlant* hierarchy you are interested in exporting nodal data. For each block in `block_export_options`, specify the block number (using the field `name`). You can add export options for multiple blocks, but in this example we just do one.

```
export_options = {
    'export_system': True,
    'block_export_options': [{
        "name": 1,
        "export_block": False,
        "export_arrays": True,
        "export_inverters": False,
        "export_dc_fields": True
    }]
}
```

Instantiate a new prediction using the *Prediction* class, specifying its `id` and `project_id` (visible in the URL of that prediction in a web browser ... `/projects/{project_id}/prediction/{id}/`).

```
project_id = 13161   # CHANGE TO YOUR PROJECT ID
prediction_id = 147813   # CHANGE TO YOUR PREDICTION ID
prediction = api.prediction(id=prediction_id, project_id=project_id)
```

Run the prediction.

```
prediction.run(export_options=export_options)
```

Retrieve the nodal data of Array 1 (in Block 1) and DC Field 1 (in Block 1 –> Array 1 –> Inverter A). Note that the lowest node (power plant hierarchy-wise) in the input dictionary specifies the nodal data returned.

```
nodal_data_array = prediction.get_nodal_data(params={
    'block_number': 1,
    'array_number': 1,
})

nodal_data_dc_field = prediction.get_nodal_data(params = {
    'block_number': 1,
    'array_number': 1,
    'inverter_name': 'A',
    'dc_field_number': 1
})
```

For system-level nodal data, call the method with no inputs.

```
nodal_data_system = prediction.get_nodal_data()
```

The nodal data returned will be returned as JSON serializable data, as detailed in the documentation for *get_nodal_data()*.

1.4.4 Download Specific Nodal Data Outputs.

The *get_nodal_data()* also supports an optional parameter for the requested output fields. For example, the DCField nodal data request from above can be modified to only return a subset of the available fields. The example below will result in only the specified output values being returned from the PlantPredict API. This can be very useful when dealing with large datasets or filtering out erroneous outputs. For a complete list of available parameters, visit our API documentation.

```
nodal_data_dc_field = prediction.get_nodal_data(params = {
    'block_number': 1,
    'array_number': 1,
    'inverter_name': 'A',
    'dc_field_number': 1,
    'nodal_parameters': 'SunZenithAngle,SunAzimuthAngle,ModuleTrackerAngle,
↪ModuleAzimuthAngle,IncidenceAngle'
})
```

1.4.5 Get Prediction Result Summary.

Instantiate the prediction you wish to retrieve results for by using the *Prediction* class, specifying its `id` and `project_id` (visible in the URL of that prediction in a web browser ... `/projects/{project_id}/prediction/{id}/`).

```
project_id = 132509 # CHANGE TO YOUR PROJECT ID
prediction_id = 707667 # CHANGE TO YOUR PREDICTION ID
prediction = api.prediction(id=prediction_id, project_id=project_id)
```

When retrieving the results, you can provide a parameter to the `get_results_summary()` function of `negate_losses=True` if you wish to see the corrected loss factors values (which are shown in the PlantPredictUI). If you wish to get the raw, un-corrected loss data, either omit this parameter or specify a value of *False*.

```
results = prediction.get_results_summary(negate_losses=True)
```

1.4.6 Clone a prediction.

Instantiate the prediction you wish to clone using the *Prediction* class, specifying its `id` and `project_id` (visible in the URL of that prediction in a web browser ... `/projects/{project_id}/prediction/{id}/`).

```
project_id = 13161    # CHANGE TO YOUR PROJECT ID
prediction_id = 147813    # CHANGE TO YOUR PREDICTION ID
prediction_to_clone = api.prediction(id=prediction_id, project_id=project_id)
```

Clone the prediction, passing in a name for the new prediction. This will create a new prediction within the same project that is an exact copy (other than the name) of the original prediction.

```
new_prediction_id = prediction_to_clone.clone(new_prediction_name='Cloned Prediction')
```

If you wish to change something about the new prediction, instantiate a new *Prediction* with the returned prediction ID, change an attribute, and call the *update()* method.

```
new_prediction = api.prediction(id=new_prediction_id, project_id=project_id)
new_prediction.get()
from plantpredict.enumerations import TranspositionModelEnum    # import at the top␣
↪of the file
new_prediction.transposition_model = TranspositionModelEnum.HAY
new_prediction.update()
```

1.4.7 Change the module in a power plant.

Instantiate the powerplant of the prediction of interest using the *PowerPlant* class, specifying the `project_id` and `prediction_id` (visible in the URL of that prediction in a web browser … `/projects/{project_id}/prediction/{id}/`).

```
project_id = 13161   # CHANGE TO YOUR PROJECT ID
prediction_id = 147813   # CHANGE TO YOUR PREDICTION ID
powerplant = api.powerplant(prediction_id=prediction_id, project_id=project_id)
```

Retrieve all of its attributes.

```
powerplant.get()
```

Specify the `id` of the module you want to replace the power plant's current module with (visible in the URL of that module in a web browser … `/module/{id}/`). Retrieve the module.

```
new_module_id = 3047
new_module = api.module(id=new_module_id)
new_module.get()
```

In order to change the module in Block 1 –> Array 1 –> Inverter A –> DC Field 1, replace the previous module's data structure, replace the module id, and update the power plant with the the *update()* method.

```
powerplant.blocks[0]['arrays'][0]['inverters'][0]['dc_fields'][0]['module'] = new_
↪module.__dict__
powerplant.blocks[0]['arrays'][0]['inverters'][0]['dc_fields'][0]['module_id'] = new_
↪module_id
powerplant.update()
```

Change various power plant properties

The SDK supports direct JSON modifcation with the power plant entity as an alternative way to adjust your power plant configuration. Below is an example of a very simple way to get a power plant configuration, modify the desired fields, and save those changes.

```
project_id = 132771 # CHANGE TO YOUR PROJECT ID
prediction_id = 706864 # CHANGE TO YOUR PREDICTION ID
power_plant = api.powerplant(project_id=project_id, prediction_id=prediction_id).get_
↪json()
power_plant['blocks'][0]['arrays'][0]['inverters'][0]['dcFields'][0]['postHeight'] =␣
↪2.55
```

(continues on next page)

(continued from previous page)

```
power_plant['blocks'][0]['arrays'][0]['inverters'][0]['dcFields'][0]['moduleId'] =␣
↪42169
power_plant['blocks'][0]['arrays'][0]['inverters'][0]['dcFields'][0]['rowSpacing'] =␣
↪10.5
power_plant['blocks'][0]['arrays'][0]['inverters'][0]['dcFields'][0]['moduleAzimuth']␣
↪= 270
api.powerplant(project_id=project_id, prediction_id=prediction_id).update_from_
↪json(power_plant)
```

1.4.8 Change a prediction's weather file.

Instantiate the prediction of interest using the *Prediction* class, specifying its `id` and `project_id` (visible in the URL of that prediction in a web browser … `/projects/{project_id}/prediction/{id}/`). Do the same for the project of interest using the *Project* class.

```
project_id = 13161   # CHANGE TO YOUR PROJECT ID
prediction_id = 147813   # CHANGE TO YOUR PREDICTION ID
prediction = api.prediction(id=prediction_id, project_id=project_id)
project = api.project(id=project_id)
```

Retrieve the project and prediction's attributes.

```
prediction.get()
project.get()
```

In this particular case, let's say you are looking for the most recent Meteonorm weather file within a 5-mile radius of the project site. Search for all weather files within a 5 mile radius of the project's `latitude`/`longitude` coordinates.

```
w = api.weather()
weathers = w.search(project.latitude, project.longitude, search_radius=5)
```

Filter the results by only Meteonorm weather files.

```
from plantpredict.enumerations import WeatherDataProviderEnum  # should import at the␣
↪top of your file
weathers_meteo = [weather for weather in weathers if int(weather['data_provider']) ==␣
↪WeatherDataProviderEnum.METEONORM]
```

If there is a weather file that meets the criteria, used the most recently created weather file's `id`. If no weather file meets the criteria, download a new Meteonorm (or whatever type you are working with) weather file and use that `id`.

```
from plantpredict.enumerations import WeatherSourceTypeAPIEnum
if weathers_meteo:
    created_dates = [w['created_date'] for w in weathers_meteo]
    created_dates.sort()
    idx = [w['created_date'] for w in weathers_meteo].index(created_dates[-1])
    weather_id = weathers_meteo[idx]['id']
else:
    weather = api.weather()
    response = weather.download(project.latitude, project.longitude,␣
↪provider=WeatherSourceTypeAPIEnum.METEONORM)
    weather_id = weather.id
```

Instantiate weather using the weather `id` and retrieve all of its attributes.

```
weather = api.weather(id=weather_id)
weather.get()
```

Ensure that the prediction `start`/`end` attributes match those of the weather file.

```
prediction.start_date = weather.start_date
prediction.end_date = weather.end_date
prediction.start = weather.start_date
prediction.end = weather.end_date
```

Change the `weather_id` of the prediction and update the prediction.

```
prediction.weather_id = weather_id
prediction.update()
```

1.4.9 Change the status of a prediction, weather, module, inverter object.

In order to change the status of a weather, module or inverter object, one must call a separate "update_status" endpoint. For example:

```
from plantpredict.enumerations import LibraryStatusEnum
prediction.update_status(LibraryStatusEnum.DRAFT_SHARED)
```

1.4.10 Upload raw weather data.

Whether you are starting with an Excel file, CSV file, SQL query, or other data format, the first step is to get your data into a JSON-like format. That format is represented in Python as a list of dictionaries, where each dictionary represents a timestamp of weather data. Depending on the initial data format, you can utilize any of Python's open-source data tools such as the native csv library or pandas. This tutorial skips that step and loads pre-processed data from `this JSON file`.

```
import json
with open('weather_details.json', 'rb') as json_file:
    weather_details = json.load(json_file)
```

Using the known latitude and longitude of the weather data location, call *`get_location_info()`* query crucial location info necessary to populate the weather file's metadata.

```
latitude = 35.0
longitude = -119.0
geo = api.geo(latitude=latitude, longitude=longitude)
location_info = geo.get_location_info()
```

Initialize the *`Weather`* entity and populate with the minimum fields required by *`create()`*. Note that the weather details time series data loaded in the first step is assigned to `weather_details` at this point.

```
from plantpredict.enumerations import WeatherDataProviderEnum
weather = api.weather()
weather.name = "Python SDK Test Weather"
weather.latitude = 35.0
weather.longitude = -119.0
weather.country = location_info['country']
weather.country_code = location_info['country_code']
```

(continues on next page)

(continued from previous page)

```
weather.data_provider = WeatherDataProviderEnum.METEONORM
weather.weather_details = weather_details
```

Assign additional metadata fields.

```
weather.elevation = round(geo.get_elevation()["elevation"], 2)
weather.locality = location_info['locality']
weather.region = location_info['region']
weather.state_province = location_info['state_province']
weather.state_province_code = location_info['state_province_code']
weather.time_zone = geo.get_time_zone()['time_zone']
weather.status = LibraryStatusEnum.DRAFT_PRIVATE
weather.data_type = WeatherDataTypeEnum.MEASURED
weather.p_level = WeatherPLevelEnum.P95
weather.time_interval = 60  # minutes
weather.global_horizontal_irradiance_sum = round(
    sum([w['global_horizontal_irradiance'] for w in weather_details])/1000, 2
)
weather.diffuse_horizontal_irradiance_sum = round(
    sum([w['diffuse_horizontal_irradiance'] for w in weather_details])/1000, 2
)
weather.direct_normal_irradiance_sum = round(
    sum([w['direct_normal_irradiance'] for w in weather_details])/1000, 2
)
weather.average_air_temperature = np.round(np.mean([w['temperature'] for w in weather_
↪details]), 2)
weather.average_relative_humidity = np.round(np.mean([w['relative_humidity'] for w in␣
↪weather_details]), 2)
weather.average_wind_speed = np.round(np.mean([w['windspeed'] for w in weather_
↪details]), 2)
weather.max_air_temperature = np.round(max([w['temperature'] for w in weather_
↪details]), 2)
```

Create the weather file in PlantPredict with `create()`.

```
weather.create()
```

1.4.11 Generate a module file.

Instantiate a `Module` object.

```
module = api.module()
```

Assign basic module parameters from the manufacturer's datasheet or similar data source.

```
from plantpredict.enumerations import CellTechnologyTypeEnum, PVModelTypeEnum
module.cell_technology_type = CellTechnologyTypeEnum.CDTE
module.number_of_cells_in_series = 264
module.pv_model = PVModelTypeEnum.ONE_DIODE_RECOMBINATION
module.reference_temperature = 25
module.reference_irradiance = 1000
module.stc_max_power = 430.0
module.stc_short_circuit_current = 2.54
module.stc_open_circuit_voltage = 219.2
module.stc_mpp_current = 2.355
```

(continues on next page)

(continued from previous page)

```
module.stc_mpp_voltage = 182.55
module.stc_power_temp_coef = -0.32
module.stc_short_circuit_current_temp_coef = 0.04
module.stc_open_circuit_voltage_temp_coef = -0.28
```

Generate single diode parameters using the default algorithm/assumptions.

```
module.generate_single_diode_parameters_default()
```

At this point, the user could simply add the remaining required fields and save the new module. Alternatively, the user can tune the module's single diode parameters to achieve (close to) a desired effective irradiance response (EIR)/low-light performance. The first step is to define target relative efficiencies at specified irradiance.

```
module.effective_irradiance_response = [
    {'temperature': 25, 'irradiance': 1000, 'relative_efficiency': 1.0},
    {'temperature': 25, 'irradiance': 800, 'relative_efficiency': 1.0029},
    {'temperature': 25, 'irradiance': 600, 'relative_efficiency': 1.0003},
    {'temperature': 25, 'irradiance': 400, 'relative_efficiency': 0.9872},
    {'temperature': 25, 'irradiance': 200, 'relative_efficiency': 0.944}
]
```

How a user chooses to tune the module's performance is relatively open-ended, but a good place to start is using PlantPredict's Optimize Series Resistance" algorithm. This will automatically change the series resistance to generate an EIR closer to the target, and re-calculate all single-diode parameters dependent on series resistance.

```
module.optimize_series_resistance()
```

At any point the user can check the current model-calculated EIR to compare it to the target.

```
calculated_effective_irradiance_response = module.calculate_effective_irradiance_
↪response()
```

An IV curve can be generated for the module for reference.

```
iv_curve_at_stc = module.generate_iv_curve(num_iv_points=250)
```

The initial series resistance optimization might not achieve an EIR close enough to the target. the user can modify any parameter, re-optimize series resistance or just recalculate dependent parameters, and check EIR repeatedly. This is the open-ended portion of module file generation. Important Note: after modifying parameters, if the user does not re-optimize series resistance, *generate_single_diode_parameters_advanced()* must be called to re-calculate `saturation_current_at_stc`, `diode_ideality_factor_at_stc`, `light_generated_current`, `linear_temperature_dependence_on_gamma`, `maximum_series_resistance` and `maximum_recombination_parameter` (if applicable).

```
module.shunt_resistance_at_stc = 8000
module.dark_shunt_resistance = 9000
module.generate_single_diode_parameters_advanced()
new_eir = module.calculate_effective_irradiance_response()
```

Once the user is satisfied with the module parameters and performance, assign other required fields.

```
from plantpredict.enumerations import ConstructionTypeEnum
module.name = "Test Module"
module.model = "Test Module"
```

(continues on next page)

(continued from previous page)

```
module.manufacturer = "Solar Company"
module.length = 2009
module.width = 1232
module.heat_absorption_coef_alpha_t = 0.9
module.construction_type = ConstructionTypeEnum.GLASS_GLASS
```

Create a new *Module* in the PlantPredict database.

```
module.create()
```

1.4.12 Set a prediction's monthly factors (albedo, soiling loss, spectral loss).

Monthly albedo, soiling loss `[%]`, and spectral loss `[%]` can all be set for a prediction with the attribute `monthly_factors` (a py:data:*dict*). This can be done upon initial creation of a prediction from scratch (see the example for *Create Project and Prediction from scratch.*), but for the sake of example, we will consider the case of updating an existing prediction.

First instantiate the prediction of interest using the *Prediction* class, specifying its `id` and `project_id` (visible in the URL of that prediction in a web browser … `/projects/{project_id}/prediction/{id}/`).

```
project_id = 13161  # CHANGE TO YOUR PROJECT ID
prediction_id = 147813  # CHANGE TO YOUR PREDICTION ID
prediction = api.prediction(id=prediction_id, project_id=project_id)
```

Retrieve the prediction's attributes.

```
prediction.get()
```

This example assumes that the user wants to specify all 3 available `monthly_factors`, and enforce that the prediction use monthly soiling loss and spectral loss averages. (Alternatively, a user can choose to only specify albedo, or albedo and soiling loss, or albedo and spectral shift.)

Set the `monthly_factors` as such, where albedo is in units `[decimal]`, soiling loss in `[%]`, and spectral loss in `[%]`. (Note: for soiling loss and spectral loss, a negative number indicates a gain.) The values below should be replaced with those obtained from measurements or otherwise relevant to the project being modeled.

```
prediction.monthly_factors = [
    {"month": 1, "month_name": "Jan", "albedo": 0.4, "soiling_loss": 0.40, "spectral_
↪shift": 0.958},
    {"month": 2, "month_name": "Feb", "albedo": 0.3, "soiling_loss": 0.24, "spectral_
↪shift": 2.48},
    {"month": 3, "month_name": "Mar", "albedo": 0.2, "soiling_loss": 0.76, "spectral_
↪shift": 3.58},
    {"month": 4, "month_name": "Apr", "albedo": 0.2, "soiling_loss": 0.88, "spectral_
↪shift": 3.48},
    {"month": 5, "month_name": "May", "albedo": 0.2, "soiling_loss": 0.81, "spectral_
↪shift": 2.58},
    {"month": 6, "month_name": "Jun", "albedo": 0.2, "soiling_loss": 1.01, "spectral_
↪shift": 1.94},
    {"month": 7, "month_name": "Jul", "albedo": 0.2, "soiling_loss": 1.21, "spectral_
↪shift": 3.7},
    {"month": 8, "month_name": "Aug", "albedo": 0.2, "soiling_loss": 0.99, "spectral_
↪shift": 4.57},
    {"month": 9, "month_name": "Sep", "albedo": 0.2, "soiling_loss": 1.34, "spectral_
↪shift": 6.39},
```

(continues on next page)

(continued from previous page)

```
    {"month": 10, "month_name": "Oct", "albedo": 0.2, "soiling_loss": 0.54, "spectral_
↪shift": 4.16},
    {"month": 11, "month_name": "Nov", "albedo": 0.3, "soiling_loss": 0.52, "spectral_
↪shift": 0.758},
    {"month": 12, "month_name": "Dec", "albedo": 0.4, "soiling_loss": 0.33, "spectral_
↪shift": 0.886}
]
```

In order to enforce that the prediction use monthly average values (rather than soiling time series from a weather file, for instance), the attributes `soiling_model` and `spectral_shift_model` must be set with the following code (assuming that both soiling loss and spectral shift loss have been specified in `monthly factors`).

```
from plantpredict.enumerations import SoilingModelTypeEnum, SpectralShiftModelEnum
prediction.soiling_model = SoilingModelTypeEnum.CONSTANT_MONTHLY
prediction.spectral_shift_model = SpectralShiftModelEnum.MONTHLY_OVERRIDE
```

Call the *update()* method on the instance of *Prediction* to persist these changes to PlantPredict.

```
prediction.update()
```

1.4.13 Upload a module .PAN file.

The PlantPredict API has two related endpoints, the first which will parse the contents of the provided .PAN file and return a PlantPredict compatible JSON object. The second, which is the POST `/Module` endpoint where the module JSON can be sent in order to create the module within PlantPredict.

Option 1: One-Step Parse & Upload This approach only requires one function call and will request, process, and upload the parsed contents.

```
file_name = "FS-6455-P CdTe October2020_v687.PAN"
file_path = "python-sdk\\example_usage\\FS-6480-P CdTe January2022_v7211.PAN"
moduleRequest = api.module()
new_moduleId = moduleRequest.upload_pan_file(file_name, file_path)
```

Option 2: Two-Step Parse, Edit, & Upload This approach gives you the opportunity to view and modify the parsed contents prior to adding the module to PlantPredict

```
file_name = "FS-6455-P CdTe October2020_v687.PAN"
file_path = "python-sdk\\example_usage\\FS-6480-P CdTe January2022_v7211.PAN"
moduleRequest = api.module()
new_module_JSON = moduleRequest.parse_pan_file(file_name, file_path)
new_module_JSON['name'] += " SDK Example Name Edit"
moduleRequest.create_from_json(new_module_JSON)
```

1.4.14 Upload an inverter .OND file.

The PlantPredict API has two related endpoints, the first which will parse the contents of the provided .OND file and return a PlantPredict compatible JSON inverter. The second, which is the POST `/Inverter` endpoint where the inverter JSON can be sent in order to create the inverter within PlantPredict.

Option 1: One-Step Parse & Upload This approach only requires one function call and will request, process, and upload the parsed contents.

```
file_name = "Huawei_SUN2000-60KTL_M0_480V.OND"
file_path = "python-sdk\\example_usage\\Huawei_SUN2000-60KTL_M0_480V.OND"
inverter_request = api.inverter()
new_inverterId = inverter_request.upload_ond_file(file_name, file_path)
```

Option 2: Two-Step Parse, Edit, & Upload This approach gives you the opportunity to view and modify the parsed contents prior to adding the module to PlantPredict

```
file_name = "Huawei_SUN2000-60KTL_M0_480V.OND"
file_path = "python-sdk\\example_usage\\Huawei_SUN2000-60KTL_M0_480V.OND"
inverter_request = api.inverter()
new_inverter_JSON = inverter_request.parse_ond_file(file_name, file_path)
new_inverter_JSON['name'] += " SDK Example Name Edit"
inverter_request.create_from_json(new_inverter_JSON)
```

1.5 Release Notes

Please refer to the package's GitHub Releases page for detailed release notes.

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y