pyoblib Documentation

SunSpec Alliance

Feb 15, 2019

Contents

The Orange Button Python Library, also called, pyoblib, provides functions to interact and work with the SunSpec Orange Button Taxonomy and provides capabilities that simplify working with Orange Button data. This project is being actively developed and is not yet ready for production use.

CHAPTER 1

Overview

The Orange Button Python Library, also called, pyoblib, provides functions to interact and work with the SunSpec Orange Button Taxonomy and provides capabilities that simplify working with Orange Button data.

This project is being actively developed and is not yet ready for production use.

The pyoblib library leverages the Python standard library to the extent possible to minimize required dependencies. pyoblib is Open Sourced and is maintained by the Orange Button Open Source community. The source code is available on GitHub - pyoblib. The code is licensed under Apache 2.0.

The SunSpec Orange Button Taxonomy is also published as open source on GitHub - solar-taxonomy.

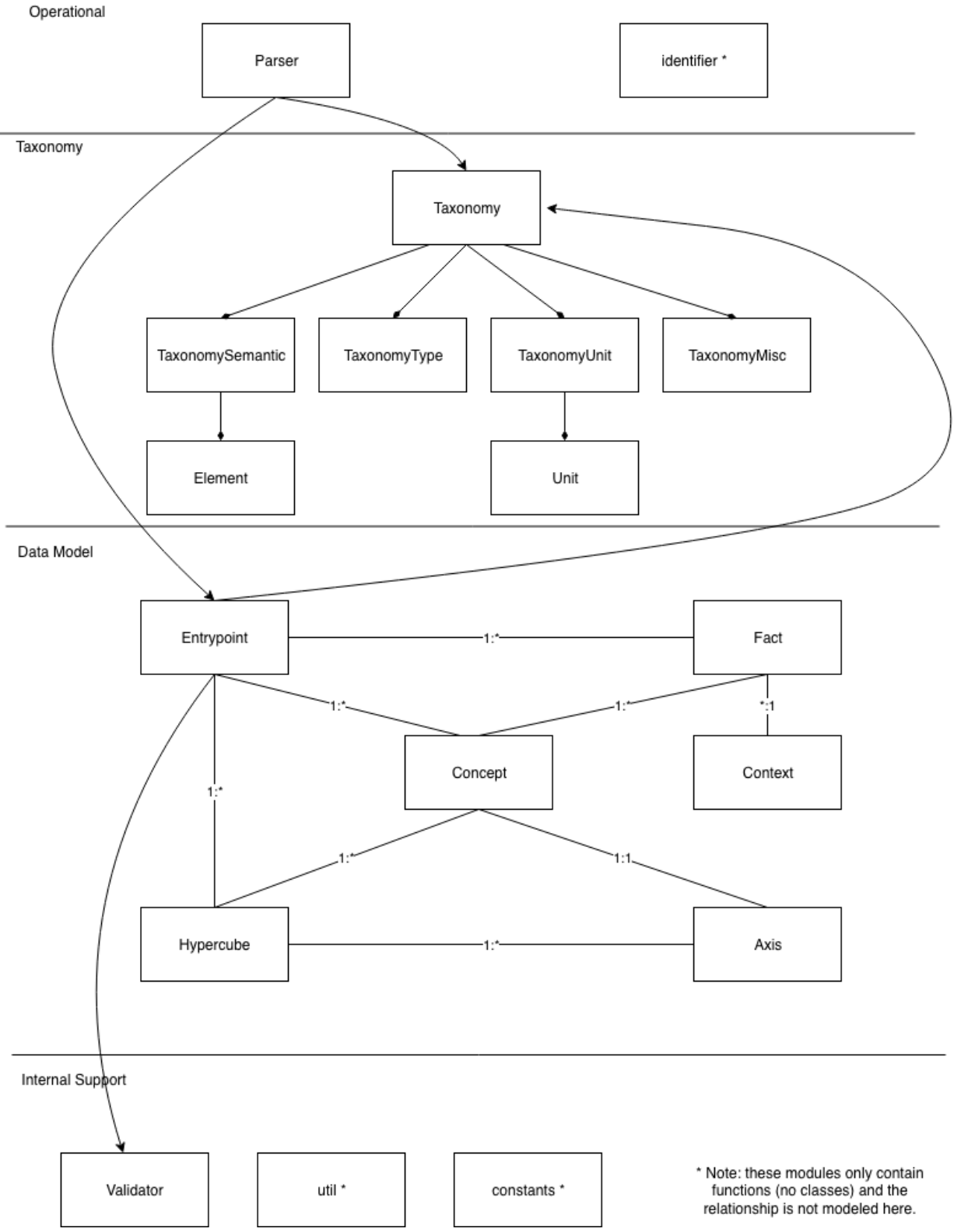
1.1 Features

- Includes an in-memory data model
- Includes in-memory meta-data about the SunSpec Orange Button Taxonomy
- Provides support for XML/JSON input-output
- Provides support for data conversion and data validation
- Supports identifier generation and validation

1.2 pyoblib Class and Module Structure

1.3 Requirements

- Python 2.7, 3.4-3.6


Operational
Parser
identifier *
Taxonomy
Taxonomy
TaxonomySemantic
TaxonomyType
TaxonomyUnit
TaxonomyMisc
Element
Unit
Data Model
Entrypoint
1:*
Fact
1:*
1:*
*:1
Concept
Context
1:*
1:*
1:1
Hypercube
1:*
Axis
Internal Support
Validator
util *
constants *
* Note: these modules only contain functions (no classes) and the relationship is not modeled here.

1.4 Installation for Development (From GitHub)

A series of Bash (Mac/Linux) shell scripts are available to assist with development and packaging. Their state is preliminary but they can be used to get started.

- cli.sh: Runs the CLI before it is packaged.
- dist-cli.sh: Packages the CLI into a single file executable.
- docs.sh: Creates the documentation (currently requires some manual work).
- setup-dev.sh: Downloads the solar-taxonomy, us-gaap taxonomy, and Units registry.
- tests.sh Runs the python tests.
- tests-cli.sh Runs the CLI test suite.

All scripts must be run from the root directory (i.e. "scripts/tests.sh" is the correct usage). Run "scripts/setup-dev.sh" before usage of other scripts.

Usage on Windows is a future feature (contributions are welcome!). It should be possible to use on Windows with manual setup at this point in time.

1.4.1 Running the Tests

In order to run the tests run the following scripts:

- tests.sh - Runs the python tests.
- tests-cli.sh - Runs the CLI test suite.

CHAPTER 2

API Reference

2.1 Subpackages

2.1.1 Test Cases API Reference

Submodules

oblib.tests.test_data_model module

class oblib.tests.test_data_model.**TestDataModelEntrypoint**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_can_write_concept()

test_concepts_can_type_check()

test_concepts_load_details()

test_conversion_to_json()

test_conversion_to_xml()

test_decimals_and_precision()

test_facts_stored_with_context()

test_get_table_for_concept()

test_hypercube_can_identify_axis_domains()

test_hypercube_rejects_context_with_unwanted_axes()

test_hypercube_rejects_out_of_domain_axis_values()

test_hypercube_store_context()

test_ids_in_xml_and_json()

test_input_ids()

test_instantiate_empty_entrypoint()

test_is_unit_method()

test_json_fields_are_strings()

test_optional_namespaces_included()

test_reject_invalid_datatype()

test_reject_missing_or_invalid_units()

test_set_context_arg()

test_set_default_context_values()

test_set_default_multiple_times()

test_set_raises_exception()

test_set_separate_dimension_args()

test_sufficient_context_instant_vs_duration()

test_tableless_facts()

test_validate_context_axes()

test_validate_values_for_enumerated_solar_data_types()

oblib.tests.test_identifier module

class oblib.tests.test_identifier.**TestIdentifier**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_invalid_identifier_bad_version()

test_invalid_identifier_format()

test_valid_identifiers()

oblib.tests.test_taxonomy module

class oblib.tests.test_taxonomy.**TestTaxonomy**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_element()

test_relationship()

test_taxonomy()

test_unit()

oblib.tests.test_taxonomy_misc module

oblib.tests.test_taxonomy_semantic module

oblib.tests.test_taxonomy_types module

oblib.tests.test_taxonomy_units module

oblib.tests.test_util module

class oblib.tests.test_util.**TestUtil**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_convert_taxonomy_xsd_bool()

test_convert_taxonomy_xsd_date()

oblib.tests.test_validator module

Module contents

2.2 Submodules

2.3 oblib.constants module

Sets pyoblib constants. SOLAR_TAXONOMY_DIR : path to solar taxonomy.

2.4 oblib.data_model module

Orange Button data model.

Consists of

- *OBInstance*, representing an XBRL Instance Document
- *Concept*, representing a XBRL concept
- *Fact*, representing a XBRL fact
- *Context*, representing the XBRL context for a fact
- *Axis* and *Hypercube* to represent tables within Instance Documents.

If you are writing Orange Button, the typical usage is to create an *OBInstance* document `doc = OBInstance()` and use `doc.set` to add data to the document, and `doc.to_XML_string` or `doc.to_JSON_string` to export to the desired format.

Example:

```
from oblib.taxonomy import getTaxonomy, PeriodType
from oblib.data_model import OBInstance
taxonomy = getTaxonomy()
mor_document = OBInstance("MonthlyOperatingReport", taxonomy)
mor_document.set_default_context({"entity": "My Company Name",
```

(continues on next page)

(continued from previous page)

```
                                    PeriodType.duration: "forever"})
mor_document.set("solar:MonthlyOperatingReportEffectiveDate",
                 date(year=2018, month=12, day=1))
mor_document.set("solar:MeasuredEnergy", "1246.25", unit_name="kWh")
xml = mor_document.to_XML_string()
```

If you are reading Orange Button data, the typical usage is to read the source JSON or XML file with `oblib.Parser` to create an `OBInstance` document, then use the document's `.get` method to read data from the document.

class oblib.data_model.**Axis**(*taxonomy*, *concept_name*)
Bases: *oblib.data_model.Concept*

A table axis. All Axes are concepts, but not all concepts are Axes, so this class is a subclass of Concept. In addition to the fields of a Concept, an Axis may also have a Domain and a finite set of allowed Domain Members.

get_domain()

Returns: the domain of the axis (a string, naming another concept)

class oblib.data_model.**Concept**(*taxonomy*, *concept_name*)
Bases: object

Represents metadata about concepts and their relationships: instances of this class are nodes in a tree data structure to keep track of which concepts are parents/children of other concepts in the schema hierarchy. Also stores concept metadata derived from the schema.

add_child(*new_child*)
Adds a child concept (in the tree structure) to this concept. Args:

new_child: Concept instance new_child becomes a child of this concept, this concept becomes parent of child

get_ancestors()
Gets all of the concept's ancestors (in the tree structure) Returns:

flat list of Concept instances, including the concept's parent, its parent's parent, etc. recursively up to the root of the tree.

get_details(*field_name*)

Args:

field_name: string name of the concept metadata field to be queried. Accepted field_names are: 'period_type' 'nillable' 'abstract' 'id' 'name' 'substitution_group' 'type_name' 'period_independent' 'typed_domain_ref' (only present for dimension concepts)

Returns: concept metadata value for the named field, or None if there is no value for that field.

set_parent(*new_parent*)
Sets the parent concept (in the tree structure) of this concept Args:

new_parent: Concept instance new_parent becomes parent of this concept, this concept becomes child of parent

validate_datatype(*value*)
Checks whether the given value is a valid one for this concept, given this concept's data type. Please note: this method is slated to be moved to the validator module. Args:

value: a float, integer, string, boolean, or date

Returns: True if the given value matches the expected type of this concept. e.g. integer, string, decimal, boolean, or complex enumerated type. False otherwise.

class oblib.data_model.**Context**(***kwargs*)
Bases: object

Represents the context for one or more facts. The context tells us when the fact applies, who is the entity reporting the fact, and also provides values for all of the table axes (aka Dimensions) needed to place the fact within a table (aka Hypercube). A fact cannot be reported without a context.

equals_context(*other_context*)

Args: other_context: a Context object

Returns: True if all my fields are the same as the fields in other_context.

get_id()

Returns: This context's ID (a string)

set_id(*hypercube*, *new_id*)
Adds this context to a hypercube and sets its ID. (A context ID is only meaningful within a certain hypercube) Args:

hypercube: reference to a Hypercube instance the given hypercube instance becomes the parent of this context

new_id: string this context's ID becomes the given ID.

Returns: None

class oblib.data_model.**Fact**(*concept_name*, *context*, *unit*, *value*, *decimals=None*, *precision=None*, *id=None*)
Bases: object

Represents a single XBRL Fact, linked to a context, that can be exported as either XML or JSON. A Fact provides a value for a certain concept within a certain context, and can optionally provide units and a precision.

set_id(*new_id*)

Args:

new_id: string The ID of this Fact becomes the new_id.

class oblib.data_model.**Hypercube**(*ob_instance*, *table_name*)
Bases: object

Data structure representing a table (aka a Hypercube) within a document. The constructor uses the taxonomy to figure out what axes the table has, what line items are allowed within the table, what are the domains of each axis, etc. The constructed table is still empty until it is populated by storing Context objects, which act as keys to locate facts within the table.

get_axes()

Returns: A list of strings which are the names of the table's axes.

get_domain(*dimensionName*)

Args:

dimensionName: string name of a dimension (aka an Axis concept) which is an axis of this table

Returns: The domain name (a string, name of a Concept) corresponding to the named dimension, if that dimension is a typed dimension; otherwise returns None.

get_name()

Returns: The name of the table, a string

get_valid_values_for_axis(*dimensionName*)

Args:

dimensionName: string name of a dimension (aka an Axis concept) which is an axis of this table

Returns: if this axis is restricted to certain values (an enumerated type), returns a list of strings where each string is a valid value (the name of a Member-type concept) Otherwise, returns empty list.

has_line_item(*line_item_name*)

Args:

line_item_name: string Name of a concept which may or may not be a line item

Returns: True if the given line_item_name is a line itme which can be stored in this table.

is_axis_value_within_domain(*dimensionName*, *dimensionValue*)

Args:

dimensionName: string name of a dimension (aka an Axis concept) which is an axis of this table

dimensionValue: a string containing a proposed value for that dimension

Returns: True if the given value is allowed in the given dimension, False otherwise.

is_typed_dimension(*dimensionName*)

Args:

dimensionName: string name of a dimension (aka an Axis concept) which is an axis of this table

Returns: True if the dimensionName (a string) is a dimension with a defined domain, as opposed to an explicit dimension.

lookup_context(*old_context*)

Args: old_context: a Context instance

Returns: If there is a matching Context stored in the table already, returns that one; otherwise returns None.

store_context(*new_context*)

De-duplicates a Context to avoid storing duplicate contexts in the table. Args:

new_context: a Context instance

Returns: A Context object with an ID that client code should use. If there is a matching Context stored in the table already, returns that one and makes no change to the table. Otherwise, a unique ID is assigned to the new context and it's both stored and returned.

validate_context(*context*)

Args:

context: a Context instance the context to be validated.

Returns: None

Raises: an OBContextException if any axis is missing, or has an out-of-bound value, or if an axis is given that doesn't belong in this table.

exception oblib.data_model.**OBConceptException**(*message*)
Bases: *oblib.data_model.OBException*

Raised if we try to set a concept that can't be set in the current Entrypoint

exception oblib.data_model.**OBContextException**(*message*)
Bases: *oblib.data_model.OBException*

Raised if we try to set a concept without sufficient Context fields

exception oblib.data_model.**OBException**(*message*)
Bases: Exception

Base class for Orange Button data validity exceptions.

class oblib.data_model.**OBInstance**(*entrypoint_name*, *taxonomy*, *dev_validation_off=False*)
Bases: object

Data structure representing an Orange Button Instance document. (Apologies if the name is confusing: the name of this thing in XBRL nomenclature is an "instance", not to be confused with "an instance of a class" in Python.) You can think of an XBRL Instance as a kind of abstract document that stores Facts in a format-agnostic way. It doesn't become a physical document until it's exported as a particular data format.

Each Fact provides a value for a certain Concept within a certain Context. An OBInstance is more than just a list of facts, however – the facts and contexts may be grouped into one or more Hypercubes (tables).

An OBInstance usually has a single "entrypoint" defining what Concepts it can hold. For example, if the entrypoint is "MonthlyOperatingReport", that means this instance document represents a monthly operating report, and is restricted to storing the Concepts that the Orange Button schema allows in a Monthly Operating Report. (There is not always a single entrypoint, though – the spec supports a multiple-entrypoint Instance or an Instance with no entrypoint. These are not implemented yet.)

get(*concept_name*, *context=None*)
Looks up the value of a fact given its concept name and context. Args:

concept_name: string Name of the concept to get the value for.

context: Context instance The context identifying the fact to look up.

Returns: The value of the fact previously set, if a match is found for concept_name and context. None if no match is found.

get_all_facts()

Returns: a list of Fact. All facts are returned in a single list, regardless of which table or context they belong to.

get_all_writable_concepts()

Returns: list of strings. Strings are the names of all concepts that can be written to this instance document as allowed by the entrypoint.

get_concept(*concept_name*)

Args:

concept_name: string name of a concept

Returns: Concept instance matching concept_name, if it's a concept allowed in this instance document by the entrypoint.

get_table(*table_name*)

Args:

table_name: string name of a table (Hypercube)

Returns: Hypercube instance matching the given table_name string, if it's a table allowed in this instance document by the entrypoint.

get_table_for_concept(*concept_name*)

Args:

concept_name: string name of a concept that can be written to this instance document

Returns: Hypercube instance – the table where the named concept belongs, if the named concept belongs on a table in this instance document. Note there are some concepts that belong in the instance document but not in any table. In that case, returns a placeholder table identified by the constant UNTABLE.

get_table_names()
Args: None Returns:

A list of strings identifying all table (hypercubes) allowed in this instance document by the entrypoint.

is_complete()
(Placeholder).

Returns: True if no required facts are missing, i.e. if there is a value for all concepts with nillable=False

is_concept_writable(*concept_name*)

Args:

concept_name: string name of a concept

Returns: True if concept_name is a writeable concept within this document. False for concepts not in this document or concepts that are only abstract parents of writeable concepts. e.g. you can't write a value to an "Abstract" or a "LineItem".

is_valid()
(Placeholder).

Returns: true if all of the facts in the document validate. i.e. they have allowed data types, allowed units, anything that needs to be in a table has all the required axis values to identify its place in that table, etc.

set(*concept_name*, *value*, ***kwargs*)
Adds a fact to the document. Stores a Fact that sets the named concept to the given value within the given context.

If concept_name and context are identical to a previous call, the old fact will be overwritten. Otherwise, a new fact is created.

The context can be provided in one of two ways: either a Context object passed in using the 'context=' keyword arg, OR the duration/instant, entity, and extra axes that define a context can all be passed in as separate keyword args. (Not both!)

Args:

concept_name: string name of a concept that can be written to this instance document

value: string, float, int, boolean, or date value to set for the concept

Keyword Args:

context: a Context instance context for the fact being set. Required unless supplying duration/instant entity/axes separately.

unit_name: string required if value is a numeric type. Specifies the unit in which the value is counted.

precision: integer number of significant digits of precision (for decimal values only)

decimals: integer number of places past the decimal point to be considered precise. (For decimal values only. Only one of precision or decimals is accepted, not both. Defaults to decimals=2.)

fact_id: string Optional ID for a fact. If not passed in it is auto-generated.

instant: datetime instant value for the context, if "context" is not given

duration: a dict with start and end fields {"start": <date>, "end": <date>} duration value for the context, if "context" is not given

entity: string entity value for the context, if "context" is not given

<axis name (*Axis)>: <axis value> as a convenience, instant/duration, entity, and <axis name> can be given directly as keyword args instead of constructing and passing a Context argument. These should only be passed in if the "context" keyword arg is not used. See the Context class constructor for more details – usage is identical.

Returns: None

Raises: OBConceptException: if the concept is not writable in this document OBContextException: if the context is not correct for the concept OBUnitException: if the unit given is wrong for the concept OBTypeException: if the value given is the wrong type for the concept

set_default_context (*dictionary*)

Set default values for context entity, instant/duration, and/or axes. The default values are used to fill in any fields that are missing from any contexts passed into set().

For example: document.set_default_context({"entity": "MyCompanyName"})

sets "MyCompanyName" as the default "entity" of this document. From then on, you can call document.set() without providing an entity in the context, and "MyCompanyName" will be used as the entity. You can set a default for an axis and it will simply be ignored by any contexts that do not require that axis.

Args:

dictionary: a python dict that can have the following keys: "entity": string, entity name to set as default for all contexts. PeriodType.instant: date to set as default for all instant-period

contexts.

PeriodType.duration: either a dict with keys "start" and "end" whose values are dates, OR the literal string "forever". This will be set as default for all duration-period contexts.

<*Axis>: If the key is the name of an Axis on one of the document's tables, and the value is a valid value for that axis, then the value will be used as default value for that axis for all contexts that require it.

to_JSON (*filename*)

Exports the document as JSON-formatted XBRL to the given filename Note: this method is slated to be moved to Parser. To ensure future support use the method with the same name and functionality in Parser.

Args:

filename: string filesystem path of a location to write the document to.

to_JSON_string()
Exports the document as JSON-formatted XBRL string Note: this method is slated to be moved to Parser. To ensure future support use the method with the same name and functionality in Parser.

Returns: String containing entire document as JSON-formatted XBRL.

to_XML(*filename*)
Exports the document as XML-formatted XBRL to the given filename Note: this method is slated to be moved to Parser. To ensure future support use the method with the same name and functionality in Parser.

Args:

filename: string filesystem path of a location to write the document to.

to_XML_string()
Exports the document as XML-formatted XBRL string Note: this method is slated to be moved to Parser. To ensure future support use the method with the same name and functionality in Parser.

Returns: String containing entire document as XML-formatted XBRL.

validate_context(*concept_name*, *context*)

Args:

concept_name: string name of a concept that can be written to this instance document

context: Context instance context to validate.

Returns: True if the given Context object contains all the information needed to provide full context for the named concept. i.e. if it provides either a valid duration or a valid instant, whichever the concept requires, and it also provides valid values for each and every dimension (Axis) required by the table where the concepr resides.

Raises: OBContextException explaining what is wrong, if some needed information is missing or invalid.

exception oblib.data_model.**OBNotFoundException**(*message*)
Bases: *oblib.data_model.OBException*

Raised if we refer to a name that's not found in the taxonomy

exception oblib.data_model.**OBTypeException**(*message*)
Bases: *oblib.data_model.OBException*

Raised if we try to set a concept to a value with an invalid data type

exception oblib.data_model.**OBUnitException**(*message*)
Bases: *oblib.data_model.OBException*

Raised if we try to set a concept to a value with incorrect units

2.5 oblib.identifier module

Handles Orange Button identifiers.

oblib.identifier.**identifier**()
Return valid UUID for Orange Button identifiers.

Returns: A string containing a valid UUID.

oblib.identifier.**validate**(*inp*)
Validate a UUID string.

Args: inp (string): Identifier to validate.

Returns: True if the input string is a valid UUID, False otherwise.

2.6 oblib.taxonomy module

Handles Orange button taxonomy.

class oblib.taxonomy.**BaseStandard**
Bases: enum.Enum

Legal values for base standards.

customary = 'Customary'

iso4217 = 'ISO4217'

non_si = 'Non-SI'

si = 'SI'

xbrl = 'XBRL'

class oblib.taxonomy.**ConceptDetails**
Bases: object

ConceptDetails models a data element within a Taxonomy Concept.

Attributes:

abstract: boolean False for standard concepts, True for concepts that model relationships but don't hold data.

id: str ID with the namespace (solar:, us-gaap:, dei:) for the concept.

name: str Name of the concept, usually identical to the ID without the namespace.

nillable: boolean True if the concept can be set to None, False if it must have a value set.

period_independent: boolean True if the concept is period independent, false otherwise.

substitution_group: SubstitutionGroup The type of substitution group.

type_name: str XBRL data type.

period_type: PeriodType Duration if the period models a period of time (including forever), instance if it is a point in time.

class oblib.taxonomy.**PeriodType**
Bases: enum.Enum

Legal values for period types.

duration = 'duration'

instant = 'instant'

class oblib.taxonomy.**Relationship**
Bases: object

Relationship holds a taxonomy relationship record.

Attributes:

role: str XBRL Arcrole

from_: str Models a relationship between two concepts in conjunction with to.

to: str Models a relationship between two concpets in conjunction with **from_**.

order: int The order of the relationshps for a single entrypoint.

class oblib.taxonomy.**RelationshipRole**
Bases: enum.Enum

Legal values for Relationship roles.

dimension_all = 'all'

dimension_default = 'dimension-default'

dimension_domain = 'dimension-domain'

domain_member = 'domain-member'

hypercube_dimension = 'hypercube-dimension'

class oblib.taxonomy.**SubstitutionGroup**
Bases: enum.Enum

Legal values for substitution groups.

dimension = 'xbrldt:dimensionItem'

hypercube = 'xbrldt:hypercubeItem'

item = 'xbrli:item'

class oblib.taxonomy.**Taxonomy**
Bases: object

Parent class for Taxonomy.

Use this class to load and access all elements of the Taxonomy. Taxonomy supplies a single import location and is better than loading a portion of the Taxonomy unless there is a specific need to save memory.

class oblib.taxonomy.**Unit**
Bases: object

Unit holds the definition of a Unit from the Unit Registry.

Attributes:

id: str ID for this unit (sample is u00020), not normally used but preserved for completeness.

unit_id: str Unit ID for this unit (for example MMBoe), usually the main lookup value.

unit_name: str Spells out Unit ID (for example MMBoe == Millions of Barrels of Oil Equivalent)

ns_unit: str Namespace, not normally used in processing.

item_type: str XBRL item type associated with the unit.

item_type_date: datetime.datetime Date the item type was set.

symbol: str Symbol used on presentation - may be same as unit_id.

definition: str Definition of unit, may be same as name or may elaborate.

base_standard: BaseStandard Base standard for a unit.

status: UnitStatus Unit status for a unit.

version_date: datetime.datetime Date for a unit

to_dict()
Convert Unit to dict.

class oblib.taxonomy.**UnitStatus**
Bases: enum.Enum

Legal values for unit registry entry status. Please note that UnitStatus is referred to as just status in the actual entries. The name has been expanded here since status is generic.

cr = 'CR'

rec = 'REC'

oblib.taxonomy.**getTaxonomy**()
Return the taxonomy.

2.7 oblib.taxonomy_misc module

Miscellaneous taxonomy functions.

class oblib.taxonomy_misc.**TaxonomyDocumentation**
Bases: object

Loads the documentation strings for each concept from solar_2018-03-31_r01_lab.xml

get_all_concepts_documentation()
Used to lookup all docstrings.

Returns: A map of all docstrings with a value of an array of two elements; array element 0 is a xlink:label and array element 1 is a xlink:role.

get_concept_documentation(*concept*)
Used to load

Args: concept (str): A concept name to lookup.

Returns: The documentation for the concept or None if not found.

class oblib.taxonomy_misc.**TaxonomyGenericRoles**
Bases: object

Represents Generic Roles portion of the taxonomy. Generally speaking this is rarely used.

get_all_generic_roles()
Used to access a list of all generic roles.

Returns: list of all generic roles (strings).

is_generic_role(*generic_role*)
Used to check if a generic role is valid.

Args: generic_role (string): Generic role to check for validity.

Returns: True if the generic role is valid, false otherwise.

class oblib.taxonomy_misc.**TaxonomyNumericTypes**
Bases: object

Represents Miscellaneous Taxonomy Objects.

Represents objects that are not covered in the other classes. Generally speaking these are rarely used.

get_all_numeric_types()
Used to access a list of numeric types.

Returns: list of all numeric types (strings).

is_numeric_type(*numeric_type*)
Used to check if a numeric type is valid.

Args: numeric_type (string): Numeric type to check for validity.

Returns: True if the numeric type is valid, false otherwise.

class oblib.taxonomy_misc.**TaxonomyRefParts**
Bases: object

Represents the Referential Parts portion of the Taxonomy. Generally speaking this is rarely used.

get_all_ref_parts()
Used to access the a list of all ref parts.

Returns: A list of all ref parts (strings).

is_ref_part(*ref_part*)
Used to check if a ref part is valid.

Args: ref_part (string): Ref part to check or validity.

Returns: True if the ref part is valid, false otherwise.

2.8 oblib.taxonomy_semantic module

Semantic taxonomy classes.

class oblib.taxonomy_semantic.**TaxonomySemantic**
Bases: object

Manage semantic portions of the taxonomy including entrypoints, concepts, concepts_details, and relationships.

get_all_concepts(*details=False*)
Return all concepts in the taxonomy.

Args: details : boolean, default False

Returns: list of concept names if details=False dict of concept details if details=True

get_all_entrypoints()
Used to access a list of all entry points (data, documents, and processes) in the Taxonomy.

Returns: A list of entrypoint names (strings).

get_all_type_names()
Used to access all type names in elements of the taxonomy.

Returns: list of type names (strings).

get_concept_details(*concept*)
Return information on a single concept.

Args:

concept [str] concept name

Returns: dict containing concept attributes

Raises: KeyError if concept is not found

get_entrypoint_concepts(*entrypoint*, *details=False*)
Return a list of all concepts in the entrypoint, with concept details (optional).

Args:

entrypoint: str name of the entrypoint

details: boolean, default False if True return details for each concept

Returns:

concepts: list elements of the list are concept names

details: dict primary key is name from concepts, value is dict of concept details

get_entrypoint_relationships(*entrypoint*)
Used to find the relationships for an entrypoint.

Args: Entrypoint (string): Entrypoint name to lookup relationships for.

Returns: A list of all relationships in an entry point. Iif the concept exists but has no relationships an empty list is returned.

is_concept(*concept*)
Validate if a concept is present in the Taxonomy.

Args: concept (string): Concept id with the namespace required.

Return: True if concept is present, False otherwise.

is_entrypoint(*entrypoint*)
Validate if an entrypoint type is present in the Taxonomy.

Args: entrypoint (string): Entrypoint name to check for presence.

Return: True if the entrypoint is present, False otherwise.

2.9 oblib.taxonomy_types module

Taxonomy types.

class oblib.taxonomy_types.**TaxonomyTypes**
Bases: object

Represents Taxonomy Types.

Allows lookup of enumerated values for each Taxonomy Type.

Please note that in the implementation of this class the variable name "type" is never used although "_type" and "types" are in order to avoid confusion with the python "type" builtin.

get_all_types()
Used to lookup all types.

Returns: A map and sublists of types.

get_type_enum(*name*)
Used to lookup a type enumeration.

Returns: An enumeration given a type or None if the type does not exist in the taxonomy.

is_type(*name*)
Validates that a type is in the taxonomy.

Returns: True if the type is present, false otherwise.

2.10 oblib.taxonomy_units module

Taxonomy units.

class oblib.taxonomy_units.**TaxonomyUnits**
Bases: object

Represents Taxonomy Units.

Allows lookup of units in the taxonomy, and enumerated values for units.

get_all_units()
Used to lookup the entire list of units.

Returns: A dict of units with unit_id as primary key.

get_unit(*unit_str*, *attr=None*)
Returns the unit with unit_id, unit_name or id is given by unit_str.

The search for the unit can be restricted by specifying attr as one of 'unit_id', 'unit_name', or 'id'.

Args:

unit_str: str can be unit_id, unit_name or id

attr: str, default None checks only specified attribute, can be 'unit_id', 'unit_name', or 'id'

Returns: unit: dict

Raises: ValueError if unit_str is not a value for a unit in the taxonomy

is_unit(*unit_str*, *attr=None*)
Returns True if unit_str is the unit_id, unit_name or id of a unit in the taxonomy, False otherwise.

The search for the unit can be restricted by specifying attr as one of 'unit_id', 'unit_name', or 'id'.

Args:

unit_str: str can be unit_id, unit_name or id

attr: str, default None checks only specified attribute, can be 'unit_id', 'unit_name', or 'id'

Returns: boolean

Raises: ValueError if attr is not valid attribute

2.11 oblib.util module

Contains basic utility methods used throughout the library. Please note that this file is not exported as part of the module so if a method has an external signature place it somewhere else.

oblib.util.**convert_json_datetime**(*inp*)
Converts a JSON data time value based upon the XBRL JSON specification format.

Args: inp (string): String containning dates in Taxonomy XSD format.

Returns: datetime

oblib.util.**convert_taxonomy_xsd_bool**(*inp*)
Returns true/false given a string loaded from the Taxonomy. Values to check are based on observed values from within the Taxonomy. If the input is not valid this will always return false.

Args: inp (string): String containing booleans in Taxonomy XSD format.

Returns: True or False

oblib.util.**convert_taxonomy_xsd_date**(*inp*)

Returns a datetime representation of a date (in string format) loaded from the taxonomy. It is assumed that the input format will be YYYY-MM-DD based upon observed values from within the taxonomy.

If the input is not valid this will return none. At this point in time this function does not require MM and DD to be two digits since there does not appear to be any reason to reject data with these two validation errors.

Args: inp (string): String containning dates in Taxonomy XSD format.

Returns: True or False

2.12 oblib.validator module

Validation functions.

class oblib.validator.**Validator**(*taxonomy*)

Bases: object

Validates values for concepts.

Args: taxonomy (Taxonomy): initialized Taxonomy.

validate_concept_value(*concept_details*, *value*)

Validate a concept value.

Args: concept_details (ConceptDetails): concept details. value (*): value to be validated.

Returns: A tuple (*, list of str) containing original or converted value and list of errors (can be empty).

2.13 Module contents

Initializes the Orange Button package.

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

o

Index

H

I

L

N

O

P

R

S

T

U

V

X