pySBOL3

Release 1.1

Bryan Bartley and Tom Mitchell

Apr 11, 2023

CONTENTS

pySBOL3 is a Python module for reading, writing, and constructing genetic designs according to the standardized specifications of the Synthetic Biology Open Language (SBOL).

CHAPTER

ONE

INTRODUCTION

pySBOL3 provides Python interfaces and their implementation for Synthetic Biology Open Language (SBOL). The current version of pySBOL3 implements SBOL 3.0. The module provides an API to work with SBOL objects, and the ability to read and write SBOL version 3 files in a variety of RDF formats. pySBOL3 supports Python 3.7 or later. pySBOL3 does not support Python 2. pySBOL3 is made freely available under the Apache 2.0 license.

To install, go to the installation page.

- The source code of pySBOL3 is available on GitHub.
- Any problems or feature requests for pySBOL2 should be reported on the GitHub issue tracker.

pySBOL3 is brought to you by Bryan Bartley, Tom Mitchell, and SBOL Developers.

Development has been supported by the NSF through the Synthetic Biology Open Language Resource collaborative award.

Development has been supported under the DARPA Synergistic Discovery & Design (SD2) Program under Air Force Research Laboratory (AFRL) contract #FA875017CO184.

CHAPTER TWO

INSTALLATION

pySBOL3 is a pure Python package and is available on any platform that supports Python 3. pySBOL3 requires Python version 3.7 or higher, and can be installed using using pip

Note: Python 2 is not supported.

2.1 Install the current release

To install the latest release of pySBOL3 using *pip*, execute the following line in a console or terminal:

```
pip install sbol3
```

If you encounter permission errors, you may want to install pySBOL3 to your user site-packages directory as follows:

```
pip install --user sbol3
```

Or alternatively, you may install as a super-user (on Unix-like platforms):

```
sudo pip install sbol3
```

To update pySBOL3 using pip, run:

```
pip install -U sbol3
```

2.2 Install the latest from GitHub

You can also install the latest version from GitHub. This might be appropriate for you if you need a specific feature or bug fix that has been fixed in git and not yet been incorporated into a release. Please exercise caution though, the latest version might also contain new bugs.

```
python3 -m pip install git+https://github.com/synbiodex/pysbol3
```

2.3 For developers

1. Clone the repository using git:

```
$ git clone --recurse-submodules https://github.com/synbiodex/pysbol3.git
```

2. Install pySBOL3 using the `setup.py` file:

```
$ cd pysbol3
$ python3 setup.py install
```

3. Test the installation by importing it in a Python interpreter:

```
$ python3
Python 3.8.5 (v3.8.5:580fbb018f, Jul 20 2020, 12:11:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sbol3
RDFLib Version: 5.0.0
```

4. Optionally run the unit test suite:

```
$ python3 -m unittest discover -s test
```

2.4 Installing on macOS

Macs do not ship with Python 3 so it is necessary to download and install Python 3 before installing pySBOL3. You can download the latest Python 3 release from python.org. After Python 3 is installed please follow the instructions above to install pySBOL3.

2.5 Using PyPy

PyPy is "a fast, compliant alternative implementation of Python." PyPy uses a just-in-time compiler (JIT), which can make certain programs faster.

pySBOL3 uses RDFlib which can be slow for reading and writing SBOL files when compared to a C implementation like Raptor .

Programs that might benefit from PyPy's JIT are programs that have longer runtimes and repeat tasks. A program that iterates over a directory reading each SBOL file, modifying the contents, and then writing the file is a good example of a program that might benefit from PyPy's JIT. On the other hand a program that reads or writes a single file is an example of a program that would probably *not* benefit from PyPy because the JIT complier doesn't have a chance to optimize the code.

pySBOL3 is compatible with PyPy. The installation and use of PyPy is out of scope for this document. Please see the PyPy documentation if you want to try using PyPy with pySBOL3.

CHAPTER

THREE

GETTING STARTED WITH PYSBOL3

This beginner's guide introduces the basic principles of pySBOL3 for new users. The examples discussed in this guide are excerpted from the Jupyter notebook (pySBOL3/examples/getting_started.ipynb). The objective of this documentation is to familiarize users with the basic patterns of the API. For more comprehensive documentation about the API, refer to documentation about specific classes and methods.

The class structure and data model for the API is based on the Synthetic Biology Open Language. For more detail about the SBOL standard, visit sbolstandard.org or refer to the specification document. This document provides diagrams and description of all the standard classes and properties that comprise SBOL.

3.1 Creating an SBOL Document

In a previous era, engineers might sit at a drafting board and draft a design by hand. The engineer's drafting sheet in pySBOL2 is called a Document. The Document serves as a container, initially empty, for SBOL data objects which represent elements of a biological design. Usually the first step is to construct a Document in which to put your objects. All file I/O operations are performed on the Document. The Document read and write methods are used for reading and writing files in SBOL format.

```
>>> import sbol3
>>> doc = sbol3.Document()
>>> doc.read('simple_library.nt')
>>> doc.write('simple_library_out.nt')
```

Reading a Document will wipe any existing contents clean before import.

A Document may contain different types of SBOL objects, including ComponentDefinitions, ModuleDefinitions, Sequences, and Models. These objects are collectively referred to as TopLevel objects because they can be referenced directly from a Document. The total count of objects contained in a Document is determined using the `len` function. To view an inventory of objects contained in the Document, simply `print` it.

```
>>> len(doc)
67
>>> print(doc)
Collection....................4
CombinatorialDerivation.......6
Component....................33
Sequence.....................24
---
Total: .........................67
```

Each SBOL object in a Document is uniquely identified by a special string of characters called a Uniform Resource Identifier (URI). A URI is used as a key to retrieve objects from the Document. To see the identities of objects in a Document, iterate over them using a Python iterator.

```
>>> for obj in doc.objects:
...     print(obj.identity)
...
http://sbolstandard.org/testfiles/All_FPs
http://sbolstandard.org/testfiles/FPs_small
http://sbolstandard.org/testfiles/FPs_small_ins
.
.
```

These URIs are said to be **sbol-compliant**. An sbol-compliant URI consists of a namespace, an optional local path, and a display ID (`display_id`). In this tutorial, we use URIs of the type `http://sbolstandard.org/testfiles/my_obj`, where the namespace is `http://sbolstandard.org/testfiles`, and the display ID is `my_object`.

Based on our inspection of objects contained in the Document above, we can see that these objects were all created in the namespace `http://sbolstandard.org/testfiles`. Thus, in order to take advantage of SBOL-compliant URIs, we set an environment variable that configures this namespace as the default.

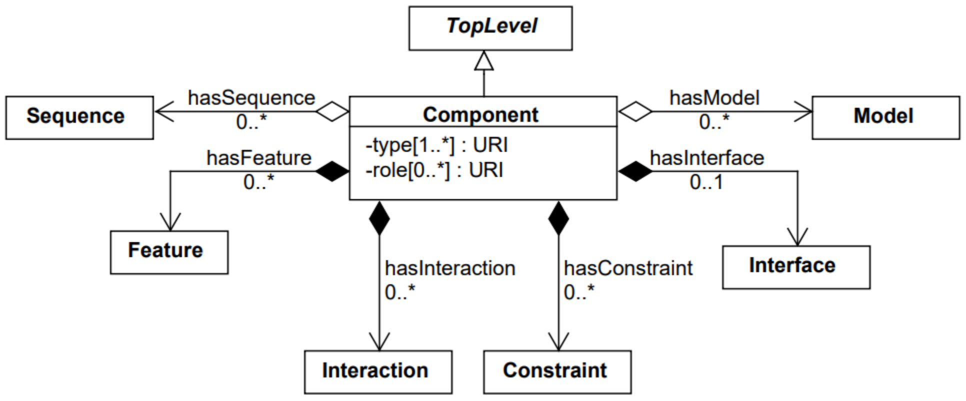
```
>>> sbol3.set_namespace('http://sbolstandard.org/testfiles')
```

Setting the namespace has several advantages. It simplifies object creation and retrieval from Documents. In addition, it serves as a way for a user to claim ownership of new objects. Generally users will want to specify a namespace that corresponds to their organization's web domain.

3.2 Creating SBOL Data Objects

Biological designs can be described with SBOL data objects, including both structural and functional features. The principle classes for describing the structure and primary sequence of a design are `Component`, `Sequence`, and `Feature`. The principle classes for describing the function of a design are `Component`, `Feature`, `Interaction`, and `Participation`.

In the SBOL specification document, classes and their properties are represented as box diagrams. Each box represents an SBOL class and its attributes. Following is an example of the diagram for the `Component` class which will be referred to in later sections. These class diagrams follow conventions of the Unified Modeling Language.



As introduced in the previous section, SBOL objects are identified by a uniform resource identifier (URI). When a new object is constructed, the user must assign a unique identity. The identity is ALWAYS the first argument supplied to the constructor of an SBOL object.

Constructors for SBOL objects follow a predictable pattern. The first argument is an identifier, which can be either a full URI, a universally unique identifier (UUID), or a display ID (possibly with a local path). If the first argument to the constructor is a valid URI or UUID, the object is created with the URI or UUID as its `identity`. Otherwise, the object is created with an `identity` composed of the first argument appended to the configured namespace (set using `sbol3.set_namespace()`). Constructors can take additional arguments, depending on whether the SBOL class has required attributes. Attributes are required if the specification says they are. In a UML diagram, required attributes are indicated as properties with a cardinality of 1 or more. For example, a `Component` (see the UML diagram above) has only one required attribute, `types`, which specifies one or more molecular types for a component. Required attributes MUST be specified when calling a constructor.

The following code creates a protein component (`types` set to `SBO_PROTEIN`).

```
>>> cas9 = sbol3.Component('Cas9', sbol3.SBO_PROTEIN)
```

The following code creates a DNA component (`types` set to `SBO_DNA`).

```
>>> target_promoter = sbol3.Component('target_promoter', sbol3.SBO_DNA, roles=[sbol3.SO_
↪PROMOTER])
```

The following code creates a DNA component with a local path (`/promoters/`), and another DNA component with a different namespace.

```
>>> # Include a local path in addition to a display_id
>>> second_promoter = sbol3.Component('promoters/second_promoter', sbol3.SBO_DNA)
>>>
>>> # Use a namespace different from the configured default namespace
>>> third_promoter = sbol3.Component('http://sbolstandard.org/other_namespace/third_
↪promoter', sbol3.SBO_DNA)
```

For examples of how the first argument of the SBOL object constructor is used to assign the object's `identity` and `display_id`, compare the following:

```
>>> target_promoter.identity
'http://sbolstandard.org/testfiles/target_promoter'
>>> target_promoter.display_id
'target_promoter'
>>> second_promoter.identity
'http://sbolstandard.org/testfiles/promoters/second_promoter'
>>> second_promoter.display_id
'second_promoter'
>>> third_promoter.identity
'http://sbolstandard.org/other_namespace/third_promoter'
>>> third_promoter.display_id
'third_promoter'
```

3.3 Using Ontology Terms for Attribute Values

Notice the `Component.types` attribute is specified using predefined constants (`sbol3.SBO_PROTEIN` and `sbol3.SBO_DNA` in the examples above). The `Component.types` property is one of many SBOL attributes that uses ontology terms as property values. The `Component.types` property uses the Systems Biology Ontology (SBO) to be specific. Ontologies are standardized, machine-readable vocabularies that categorize concepts within a domain of scientific study. The SBOL 3.0 standard unifies many different ontologies into a high-level, object-oriented model.

Ontology terms also take the form of Uniform Resource Identifiers. Many commonly used ontological terms are built-in to pySBOL3 as predefined constants. If an ontology term is not provided as a built-in constant, its URI can often be found by using an ontology browser tool online. Browse Sequence Ontology terms here and Systems Biology Ontology terms here. While the SBOL specification often recommends particular ontologies and terms to be used for certain attributes, in many cases these are not rigid requirements. The advantage of using a recommended term is that it ensures your data can be interpreted or visualized by other applications that support SBOL. However in many cases an application developer may want to develop their own ontologies to support custom applications within their domain.

The following example illustrates how the URIs for ontology terms can be easily constructed, assuming they are not already part of pySBOL3's built-in ontology constants.

```
>>> SO_ENGINEERED_FUSION_GENE = tyto.SO.engineered_fusion_gene
>>> SO_ENGINEERED_FUSION_GENE
'https://identifiers.org/SO:0000288'
>>> SBO_DNA_REPLICATION = tyto.SBO.DNA_replication
>>> SBO_DNA_REPLICATION
'https://identifiers.org/SBO:0000204'
```

For more information on using ontology terms with pySBOL3, see: Using Ontology Terms.

3.4 Adding, Finding, and Getting Objects from a Document

In some cases a developer may want to use SBOL objects as intermediate data structures in a computational biology workflow. In this case, the user is free to manipulate objects independently of a Document. However, if the user wishes to write out a file with all the information contained in their object, they must first add it to the Document. This is done using the `add` method.

```
>>> doc.add(target_promoter)
>>> doc.add(cas9)
```

Objects can be found and retrieved from a Document by using the `find` method. This method can take either the object's `identity` (i.e., full URI) or `display_id` (local identifier) as an argument.

```
>>> cas9.identity
'http://sbolstandard.org/testfiles/Cas9'
>>> found_obj = doc.find('http://sbolstandard.org/testfiles/Cas9')
>>> found_obj.identity
'http://sbolstandard.org/testfiles/Cas9'
>>> cas9.display_id
'Cas9'
>>> found_obj = doc.find('Cas9')
>>> found_obj.identity
'http://sbolstandard.org/testfiles/Cas9'
```

It is possible to have multiple SBOL objects with the same `display_id` (but different `identity`) in the same document. In that case, if the `find` method is called with the `display_id` as the argument, it will return the matching object that was added to the document first.

```
>>> cas9a = sbol3.Component('http://sbolstandard.org/other_namespace/Cas9', sbol3.SBO_
↪PROTEIN)
>>> cas9a.identity
'http://sbolstandard.org/other_namespace/Cas9'
>>> cas9a.display_id
'Cas9'
>>> doc.add(cas9a)
>>> found_obj = doc.find('Cas9')
>>> found_obj.identity
'http://sbolstandard.org/testfiles/Cas9'
>>> found_obj = doc.find('http://sbolstandard.org/other_namespace/Cas9')
>>> found_obj.identity
'http://sbolstandard.org/other_namespace/Cas9'
```

3.5 Getting, Setting, and Editing Attributes

The attributes of an SBOL object can be accessed like other Python class objects, with a few special considerations. For example, to get the values of the `display_id` and `identity` properties of any object :

```
>>> print(cas9.display_id)
Cas9
>>> print(cas9.identity)
http://sbolstandard.org/testfiles/Cas9
```

Note that `display_id` gives only the shorthand, local identifier for the object, while the `identity` property gives the full URI.

The attributes above return singleton values. Some attributes, like `Component.roles` and `Component.types` support multiple values. Generally these attributes have plural names. If an attribute supports multiple values, then it will return a list. If the attribute has not been assigned any values, it will return an empty list.

```
>>> cas9.types
['https://identifiers.org/SBO:0000252']
>>> cas9.roles
[]
```

Setting an attribute follows the ordinary convention for assigning attribute values:

```
>>> cas9.description = 'This is a Cas9 protein'
```

To set multiple values:

```
>>> plasmid = sbol3.Component('pBB1', sbol3.SBO_DNA)
>>> plasmid.roles = [ sbol3.SO_DOUBLE_STRANDED, sbol3.SO_CIRCULAR ]
```

Properties such as `types` and `roles` behave like Python lists, and list operations like `append` and `extend` will work directly on these kind of attributes:

```
>>> plasmid.roles = [ sbol3.SO_DOUBLE_STRANDED ]
>>> plasmid.roles.append( sbol3.SO_CIRCULAR )

>>> plasmid.roles = []
>>> plasmid.roles.extend( [sbol3.SO_DOUBLE_STRANDED, sbol3.SO_CIRCULAR] )

>>> plasmid.roles = [ sbol3.SO_DOUBLE_STRANDED ]
>>> plasmid.roles += [ sbol3.SO_CIRCULAR ]
```

To clear all values from an attribute, set it to an empty list:

```
>>> plasmid.roles = []
```

3.6 Creating and Adding Child Objects

Some SBOL objects can be composed into hierarchical parent-child relationships. In the specification diagrams, these relationships are indicated by black diamond arrows. In the UML diagram above, the black diamond indicates that Components are parents of Features. In pySBOL3, properties of this type are created as subcomponents and then added to the appropriate list attribute of the parent component. The constructor for the `SubComponent` class takes a `Component` as its only required argument. In this usage, the `Component` is "… analogous to a blueprint or specification sheet for a biological part…" Whereas the `SubComponent` "… represents the specific occurrence of a part…" within a larger design (SBOL version 3.0.0 specification document). For example, to add a promoter to a circuit design, first define the promoter and circuit as SBOL `Component` objects, then define a `SubComponent` as an instance of the promoter and add that `SubComponent` to the circuit's `features` attribute:

```
>>> ptet = sbol3.Component('pTetR', sbol3.SBO_DNA, roles=[sbol3.SO_PROMOTER])

>>> circuit = sbol3.Component('circuit', sbol3.SBO_DNA, roles=[sbol3.SO_ENGINEERED_
↪REGION])

>>> ptet_sc = sbol3.SubComponent(ptet)
>>> circuit.features += [ptet_sc]
```

3.7 Creating and Editing Reference Properties

Some SBOL objects point to other objects by way of URI references. For example, Components point to their corresponding Sequences by way of a URI reference. These kind of properties correspond to white diamond arrows in UML diagrams, as shown in the figure above. Attributes of this type contain the URI of the related object.

```
>>> gfp = sbol3.Component('GFP', sbol3.SBO_DNA)
>>> doc.add(gfp)
>>> gfp_seq = sbol3.Sequence('GFPSequence', elements='atgnnntaa', encoding=sbol3.IUPAC_
↪DNA_ENCODING)
>>> doc.add(gfp_seq)
>>> gfp.sequences = [ gfp_seq ]
>>> print(gfp.sequences)
['http://sbolstandard.org/testfiles/GFPSequence']
>>> # Look up the sequence via the document
>>> seq2 = gfp.sequences[0].lookup()
```

(continues on next page)

(continued from previous page)

```
>>> seq2 == gfp_seq
True
```

Note that assigning the `gfp_seq` object to the `gfp.sequences` actually results in assignment of the object's URI. An equivalent assignment is as follows:

```
>>> gfp.sequences = [ gfp_seq.identity ]
>>> print(gfp.sequences)
['http://sbolstandard.org/testfiles/GFPSequence']
>>> seq2 = gfp.sequences[0].lookup()
>>> seq2 == gfp_seq
True
```

Also note that the DNA sequence information is saved as the `elements` attribute of the `Sequence` object, as per the SBOL 3 specification:

```
>>> gfp_seq.elements
'atgnnntaa'
```

3.8 Iterating and Indexing List Properties

Some SBOL object properties can contain multiple values or objects. You may iterate over those list properties as with normal Python lists:

```
>>> # Iterate through objects (black diamond properties in UML)
>>> for feat in circuit.features:
...     print(f'{feat.display_id}, {feat.identity}, {feat.instance_of}')
...
SubComponent1, http://sbolstandard.org/testfiles/circuit/SubComponent1, http://
↪sbolstandard.org/testfiles/pTetR
SubComponent2, http://sbolstandard.org/testfiles/circuit/SubComponent2, http://
↪sbolstandard.org/testfiles/op1
SubComponent3, http://sbolstandard.org/testfiles/circuit/SubComponent3, http://
↪sbolstandard.org/testfiles/RBS1
.
.
```

```
>>> # Iterate through references (white diamond properties in UML)
>>> for seq in gfp.sequences:
...     print(seq)
...
http://sbolstandard.org/testfiles/GFPSequence
```

Numerical indexing of list properties works as well:

```
>>> for n in range(0, len(circuit.features)):
...     print(circuit.features[n].display_id)
...
SubComponent1
SubComponent2
```

(continues on next page)

(continued from previous page)

```
SubComponent3
.
.
```

3.9 Copying Documents and Objects

Copying a Document can result in a few different ends, depending on the user's goal. The first option is to create a simple copy of the original Document with Document.copy. After copying, the object in the Document clone has the same identity as the object in the original Document.

```
>>> import sbol3
>>> sbol3.set_namespace('https://example.org/pysbol3')
>>> doc = sbol3.Document()
>>> cd1 = sbol3.Component('cd1', types=[sbol3.SBO_DNA])
>>> doc.add(cd1)
<sbol3.component.Component object at 0x7fb7d805b9a0>
>>> for o in doc:
...     print(o)
...
<Component https://example.org/pysbol3/cd1>
>>> doc2 = doc.copy()
>>> for o in doc2:
...     print(o)
...
<Component https://example.org/pysbol3/cd1>
>>> cd1a = doc2.find('cd1')
>>>
>>> # The two objects point to different locations in memory, they are different objects␣
↪with the same name.
>>>
>>> cd1a
<sbol3.component.Component object at 0x7fb7c83e7c40>
>>> cd1
<sbol3.component.Component object at 0x7fb7d805b9a0>
```

The sbol3.copy function is a more powerful way to copy or clone objects. Document.copy is built on sbol3.copy. The sbol3.copy function lets a user copy objects as above. It also lets the user change object namespaces and add the new documents to an existing Document.

For example, if a user wants to copy objects and change the namespace of those objects, a user can use the into_namespace argument to sbol3.copy. Following on from the example above:

```
>>> objects = sbol3.copy(doc, into_namespace='https://example.org/foo')
>>> len(objects)
1
>>> for o in objects:
...     print(o)
...
<Component https://example.org/foo/cd1>
>>>
```

Finally, if a user wants to construct a new set of objects and add them to an existing `Document` they can do so using the `into_document` argument to `sbol3.copy`. Again, following on from the example above:

```
>>> doc3 = sbol3.Document()
>>> len(doc3)
0
>>> # Any iterable of TopLevel can be passed to sbol3.copy:
>>> sbol3.copy([cd1], into_namespace='https://example.org/bar', into_document=doc3)
[<sbol3.component.Component object at 0x7fb7d844aa60>]
>>> len(doc3)
1
>>> for o in doc3:
...     print(o)
...
<Component https://example.org/bar/cd1>
>>>
```

CHAPTER

FOUR

SBOL EXTENSIONS

The Synthetic Biology Open Language is an extensible data representation. This means that users may add new classes to the core data model or add new properties to existing SBOL classes. These are referred to as "extensions" or "custom annotations". Extension data can be serialized to and parsed from an SBOL file, enabling exchange of the data with other users. This capability is often helpful because applications may produce application-specific data (for example, visual layout information for a visualization tool), or there may be special types of scientific data that the user may want to associate with SBOL data. The extensibility of the SBOL format is another of its advantages compared to traditional bioinformatics formats. This help page provides examples of how to create, access, and exchange extension data.

4.1 Defining Extension Properties

Extension data may be added or retrieved from an SBOL object using special SBOL property interfaces. Currently the following property interfaces are supported:

```
TextProperty
URIProperty
IntProperty
FloatProperty
DateTimeProperty
ReferencedObject
OwnedObject
```

Each interface is specialized for a specific data type. For example *TextProperty*, *IntProperty*, and *FloatProperty* contain string, integer, and float values, respectively. The *URIProperty* is used whenever an ontology term is specified. Some properties include special validation rules. For example, *DateTimeProperty* will validate whether its string value conforms to XML Schema Date/Time format. *OwnedObject* is used to define parent-child compositional relationships between objects, while the *ReferencedObject* is used to define associative links between objects.

A property can be created by calling a constructor of the same name. Property constructors follow a general pattern. The first argument indicates the object to which the property will be bound. The second argument is a URI that indicates how the data property will appear when serialized into the contents of an SBOL file. In XML, such a URI is called a "qualified name" or QName. Property constructors also include a cardinality lower and upper bound. Typical values for a lower bound are either 0 (if the field is optional) or 1 (if the field requires a value). Typical upper bound values are either 1 (if the field can contain only a single value) or *math.inf* (if the field can contain an arbitrary length list of values).

Once a property is initialized, it can be assigned and accessed like any other property, as the following example demonstrates. This example associates an x and y coordinate with the Component for layout rendering.

```
>>> import sbol3
RDFLib Version: 5.0.0
```

(continues on next page)

(continued from previous page)

```
>>> c1 = sbol3.Component('http://example.org/sbol3/c1', sbol3.SBO_DNA)
>>> c1.x_coordinate = sbol3.IntProperty(c1, 'http://example.org/my_vis#x_coordinate', 0,␣
↪1)
>>> c1.y_coordinate = sbol3.IntProperty(c1, 'http://example.org/my_vis#y_coordinate', 0,␣
↪1)
>>> c1.x_coordinate = 150
>>> c1.y_coordinate = 175
>>> doc = sbol3.Document()
>>> doc.add(c1)
>>> print(doc.write_string(sbol3.SORTED_NTRIPLES).decode())

<http://example.org/sbol3/c1> <http://example.org/my_vis#x_coordinate> "150"^^<http://
↪www.w3.org/2001/XMLSchema#integer> .
<http://example.org/sbol3/c1> <http://example.org/my_vis#y_coordinate> "175"^^<http://
↪www.w3.org/2001/XMLSchema#integer> .
<http://example.org/sbol3/c1> <http://sbols.org/v3#displayId> "c1" .
<http://example.org/sbol3/c1> <http://sbols.org/v3#type> <https://identifiers.org/
↪SBO:0000251> .
<http://example.org/sbol3/c1> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
↪sbols.org/v3#Component> .

>>>
```

As you can see, when the file is serialized, the extension data are integrated with core SBOL data.

The example above demonstrates how to write extension data. The following example demonstrates how to recover extension data upon loading a file. This code block simply takes the output from above and reads it into a new *Document*. Once the *IntProperty* interfaces are initialized, the extension data becomes accessible.

```
>>> doc2 = sbol3.Document()
>>> doc2.read_string(doc.write_string(sbol3.NTRIPLES), sbol3.NTRIPLES)
>>> c1a = doc2.find('http://example.org/sbol3/c1')
>>> c1a.display_id
'c1'

# See that x_coordinate is not known
>>> c1a.x_coordinate
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/path/to/sbol3/object.py", line 35, in __getattribute__
    result = object.__getattribute__(self, name)
AttributeError: 'Component' object has no attribute 'x_coordinate'

# Now define the extension properties
>>> c1a.x_coordinate = sbol3.IntProperty(c1, 'http://example.org/my_vis#x_coordinate', 0,
↪ 1)
>>> c1a.y_coordinate = sbol3.IntProperty(c1, 'http://example.org/my_vis#y_coordinate', 0,
↪ 1)
>>> c1a.x_coordinate
150
>>> c1a.y_coordinate
175
```

While in many cases a user knows in advance whether or not a file contains certain types of extension data, it may not always be obvious. Therefore it is possible to inspect the data fields contained in an object using the *properties* attribute. This attribute contains the URIs of the properties associated with an object. Most of the properties listed will be core properties, especially those in the *http://sbols.org*, *http://www.w3.org/ns/prov*, and *http://purl.org/dc/terms* namespaces. If any URIs are listed in a namespace that is not one of these, then it is likely custom extension data.

```
>>> import pprint
>>> pprint.pprint(sorted(c1a.properties))
['http://example.org/my_vis#x_coordinate',
 'http://example.org/my_vis#y_coordinate',
 'http://sbols.org/v3#description',
 'http://sbols.org/v3#displayId',
 'http://sbols.org/v3#hasAttachment',
 'http://sbols.org/v3#hasModel',
 'http://sbols.org/v3#hasSequence',
 'http://sbols.org/v3#name',
 'http://sbols.org/v3#role',
 'http://sbols.org/v3#type',
 'http://www.w3.org/ns/prov#wasDerivedFrom',
 'http://www.w3.org/ns/prov#wasGeneratedBy']
```

4.2 Extension Classes

Extension classes are classes that are derived from SBOL classes. Using extension classes, the data model can be expanded *ad hoc* to represent a wider domain of synthetic biology knowledge. Extension classes allow a user to define an explicit specification for the types of annotation data it contains. This is advantageous when a user wants to efficiently share extension data with other users. A user can share the Python files containing the extension class definition, and other users will have instant access to the extension data.

In the following examples, an extension class includes a class definition containing attributes with SBOL property interfaces, as described in the preceding example. Each class definition must have a builder function. The pySBOL parser invokes the builder function when it encounters the RDF type of an object in the SBOL file.

4.2.1 Example 1: Override a Core Class

The following example illustrates this concept. It defines a *ComponentExtension* class which, like the example in the preceding section, includes *x_coordinate* and *y_coordinate* properties. However, in this case, the user does not need to define the property interface, because the extension class definition already does this. The user can simply import the class definition into their code base and access the additional annotation data.

In this example, overriding the core class has the effect that any *Component* that is accessed in a Document after file I/O is now represented as a *ComponentExtension* rather than a *Component*.

Listing 1: examples/compext.py

```
import sbol3

X_COORDINATE_URI = 'http://example.org/my_vis#x_coordinate'
Y_COORDINATE_URI = 'http://example.org/my_vis#y_coordinate'


class ComponentExtension(sbol3.Component):
```

(continues on next page)

(continued from previous page)

```
    """Override sbol3.Component to add two fields
    for visual display.
    """

    def __init__(self, identity, types,
                 *, type_uri=sbol3.SBOL_COMPONENT):
        super().__init__(identity=identity, types=types, type_uri=type_uri)
        self.x_coordinate = sbol3.IntProperty(self, X_COORDINATE_URI, 0, 1,
                                              initial_value=0)
        self.y_coordinate = sbol3.IntProperty(self, Y_COORDINATE_URI, 0, 1,
                                              initial_value=0)


def build_component_extension(*, identity, type_uri):
    """A builder function to be called by the SBOL3 parser
    when it encounters a Component in an SBOL file.
    """

    # `types` is required and not known at build time.
    # Supply a missing value to the constructor, then clear
    # the missing value before returning the built object.
    obj = ComponentExtension(identity=identity,
                             types=[sbol3.PYSBOL3_MISSING],
                             type_uri=type_uri)
    # Remove the placeholder value
    obj.clear_property(sbol3.SBOL_TYPE)
    return obj


# Register the builder function so it can be invoked by
# the SBOL3 parser to build objects with a Component type URI
sbol3.Document.register_builder(sbol3.SBOL_COMPONENT,
                                build_component_extension)
```

In a Python interpreter with the *ComponentExtension* class loaded you can now use the *x_coordinate* and *y_coordinate* properties automatically. Here is an example that creates a *ComponentExtension*, sets the new properties, then saves it to file. The saved file is then loaded and the new properties are preserved.

```
>>> c = ComponentExtension('http://example.org/sbol3/c1', sbol3.SBO_DNA)
>>> c.x_coordinate
0
>>> c.y_coordinate
0
>>> c.x_coordinate = 150
>>> c.y_coordinate = 100
>>>
>>> # Round trip the document
>>> doc = sbol3.Document()
>>> doc.add(c)
>>> doc2 = sbol3.Document()
>>> doc2.read_string(doc.write_string(sbol3.NTRIPLES), sbol3.NTRIPLES)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Fetch the Component out of the new document
>>> c2 = doc2.find('c1')
>>> c2.x_coordinate
150
>>> c2.y_coordinate
100
>>> type(c2).__name__
'ComponentExtension'
```

4.2.2 Example 2: Extend a Core Class

Instead of annotating a class like in the above example, it is also possible to extend a core class, creating a new class that has all the properties of the core class as well as additional properties. The key to doing this extension properly is to use multiple inheritance to extend both the desired core class as well as the appropriate *Custom* class, either *CustomTopLevel* or *CustomIdentified*. The choice of which Custom class is dictated by the class you are choosing to extend. If the core class is a *TopLevel*, choose *CustomTopLevel*. Otherwise choose *CustomIdentified*.

Here is an example class definition that uses multiple inheritance to extend *CombinatorialDerivation*:

```
SAMPLE_SET_URI = 'http://bioprotocols.org/opil/v1#SampleSet'


class SampleSet(sbol3.CombinatorialDerivation, sbol3.CustomTopLevel):

    def __init__(self, identity: str, template: Union[sbol3.Component, str],
                 *, type_uri: str = SAMPLE_SET_URI):
        super().__init__(identity=identity, template=template,
                         type_uri=type_uri)
        # Add additional properties here


def build_sample_set(identity: str,
                     *, type_uri: str = SAMPLE_SET_URI):
    template = sbol3.PYSBOL3_MISSING
    return SampleSet(identity=identity, template=template, type_uri=type_uri)


sbol3.Document.register_builder(SAMPLE_SET_URI, build_sample_set)
```

4.2.3 Example 3: Define a New Class

In the above annotation example (Example 1), the extension class overrides the core *ComponentDefinition* class, allowing the user to extend the core class definition with extra properties. In other cases, a user may want to extend the SBOL data model with an entirely new class. In this case, the user defines a new class derived from *TopLevel*. The definition of this extension class differs from the example above in one important respect. It now becomes necessary to specify an RDF type for the new class. The RDF type is a URI represented by the *type_uri* parameter passed to the constructor. The *type_uri* dictates that the object will now be serialized as an entirely new class. The following example defines a custom *Analysis* extension class.

Listing 2: examples/analysisext.py

```
import sbol3


class Analysis(sbol3.CustomTopLevel):

    TYPE_URI = 'http://examples.org#Analysis'
    FITTED_MODEL_URI = 'http://example.org/sbol3#fit'

    def __init__(self, identity=None, model=None,
                 *, type_uri=TYPE_URI):
        # Override the default type_uri that is used when serializing
        super().__init__(identity=identity, type_uri=type_uri)
        self.fittedModel = sbol3.ReferencedObject(self,
                                                  Analysis.FITTED_MODEL_URI,
                                                  0, 1,
                                                  initial_value=model)


# Register the constructor with the parser
sbol3.Document.register_builder(Analysis.TYPE_URI, Analysis)
```

Extension classes that do not override a core SBOL class can be accessed from a *Document* through general *add* and *find* methods.

```
>>> doc = sbol3.Document()
>>> a = Analysis('http://example.org/sbol3/a1')
>>> doc.add(a)
>>> also_a = doc.find(a.identity)
>>> also_a is a
True
```

4.2.4 Example 4: Composing Extension Objects

It is also possible to create extension classes that have a parent-child compositional relationship. In this case the child class should be defined to inherit from *CustomIdentified*, while the parent class inherits from *CustomTopLevel*. The child class is referenced through an *OwnedObject* interface. The following example introduces the *DataSheet* class which can now be referenced through the parent *Analysis* class.

Listing 3: examples/datasheetext.py

```
import sbol3


class DataSheet(sbol3.CustomIdentified):
    TYPE_URI = 'http://example.org/sbol3#DataSheet'
    RATE_URI = 'http://example.org/sbol3#txRate'

    def __init__(self, rate=None):
        super().__init__(type_uri=DataSheet.TYPE_URI)
        self.transcription_rate = sbol3.FloatProperty(self,
```

(continues on next page)

(continued from previous page)

```
                                                       DataSheet.RATE_URI,
                                                       0, 1,
                                                       initial_value=rate)


class Analysis(sbol3.CustomTopLevel):
    TYPE_URI = 'http://example.org/sbol3#Analysis'
    MODEL_URI = 'http://example.org/sbol3#fittedModel'
    DATA_SHEET_URI = 'http://example.org/sbol3#dataSheet'

    def __init__(self, identity=None, model=None):
        super().__init__(identity=identity,
                         type_uri=Analysis.TYPE_URI)
        self.fitted_model = sbol3.ReferencedObject(self,
                                                    Analysis.MODEL_URI,
                                                    0, 1,
                                                    initial_value=model)
        self.data_sheet = sbol3.OwnedObject(self,
                                             Analysis.DATA_SHEET_URI,
                                             0, 1,
                                             type_constraint=DataSheet)


# Register the constructor with the parser
sbol3.Document.register_builder(DataSheet.TYPE_URI, DataSheet)
sbol3.Document.register_builder(Analysis.TYPE_URI, Analysis)
```

```
>>> doc = sbol3.Document()
>>> a = Analysis('http://example.org/sbol3/a1')
>>> doc.add(a)
>>> a.data_sheet = DataSheet()
>>> a.data_sheet.transcription_rate = 96.3
>>> a.data_sheet.transcription_rate
96.3
```

CHAPTER

FIVE

VALIDATION IN PYSBOL3

5.1 Validating Documents

The most common use of validation will be validating entire documents. After objects have been loaded, created, or manipulated, a programmer can invoke *validate* on the *Document* to get a report of any errors or warnings. If the length of the report is 0, or if the report evaluates to boolean False, there are no validation issues. If there are validation issues it is possible to iterate over the validation errors and warnings as show in the next section.

Here is an example that validates a newly loaded document:

```
>>> import sbol3
RDFLib Version: 5.0.0
>>> doc = sbol3.Document()
>>> doc.read('combine2020.ttl')
>>> report = doc.validate()
>>> len(report)
0
>>> bool(report)
False
```

5.2 Validating Objects

The pySBOL3 library includes a capability to generate a validation report for an SBOL3 object or an SBOL3 Document. The report can be used to check your work or fix issues with your document or object.

Here is a short example of how to validate an object. We intentionally create a Range with end before start, which is invalid SBOL3. This generates a validation error in the ValidationReport:

```
>>> import sbol3
RDFLib Version: 5.0.0
>>> seq_uri = 'https://github.com/synbiodex/pysbol3/sequence'
>>> start = 10
>>> end = 1
>>> r = sbol3.Range(seq_uri, start, end)
>>> report = r.validate()
>>> len(report)
1
>>> bool(report)
True
>>> for error in report.errors:
```

(continues on next page)

(continued from previous page)

```
...     print(error.message)
...
sbol3-11403: Range.end must be >= start
```

Validating an object automatically validates any child objects. Invoking *validate()* on a document will validate all objects contained in that document.

5.3 Extending Validation

If you are building extension classes and want to add custom validation to those objects you can extend the pySBOL3 validation in your custom classes. To do so, define your own *validate* method, call the super method, then perform your own validation, adding warnings or errors to the validation report. Finally, your *validate* method must return the *ValidationReport* to the caller. This new method will automatically get called when a *Document* is validated or when the an instance of this class is validated.

Here is an example:

```
def validate(self, report: ValidationReport = None) -> ValidationReport:

    # Invoke the super method, and hold on to the resulting report
    report = super().validate(report)

    # Run my own validation here
    if self.x >= self.x2:
        report.addError(self.identity, None, 'X must be less than X2')
    if self.x > 100:
        report.addWarning(self.identity, None, 'X values over 100 do not work well')

    # Return the report to the caller
    return report
```

CHAPTER

SIX

USING VISITORS

The Visitor Pattern is a well known and commonly used pattern for performing an operation on the elements of an object structure. There are many online resources for learning about the visitor pattern.

The implementation of the Visitor Pattern in pySBOL3 is very simple. When a pySBOL3 object's *accept* method is called, a visitor is passed as the only argument. The *accept* method, in turn, invokes *visit_type* on the visitor, passing the pySBOL3 object as the only argument.

Traversal of the pySBOL3 object graph is left to the visitor itself. When a *visit_type* method is called, the visitor must then invoke *accept* on any child objects it might want to visit. See the code sample below for an example of this traversal.

6.1 Resources

- https://sourcemaking.com/design_patterns/visitor
- https://refactoring.guru/design-patterns/visitor
- https://www.informit.com/store/design-patterns-elements-of-reusable-object-oriented-9780201633610

6.2 Example Code

The program below will visit each top level object in the document as well as visiting any features on the top level components. Note that the visitor must direct the traversal of the features, as discussed above.

```python
import sbol3


class MyVisitor:

    def visit_component(self, c: sbol3.Component):
        print(f'Component {c.identity}')
        for f in c.features:
            f.accept(self)

    def visit_sequence(self, s: sbol3.Component):
        print(f'Sequence {s.identity}')

    def visit_sub_component(self, sc: sbol3.Component):
        print(f'SubComponent {sc.identity}')
```

(continues on next page)

(continued from previous page)

```
doc = sbol3.Document()
doc.read('test/SBOLTestSuite/SBOL3/BBa_F2620_PoPSReceiver/BBa_F2620_PoPSReceiver.ttl')
visitor = MyVisitor()
for obj in doc.objects:
    obj.accept(visitor)
```

6.3 Visit Methods

The table below lists each class that has an accept method and the corresponding method that is invoked on the visitor passed to the accept method.

Class	Visit Method
Activity	visit_activity
Agent	visit_agent
Association	visit_association
Attachment	visit_attachment
BinaryPrefix	visit_binary_prefix
Collection	visit_collection
CombinatorialDerivation	visit_combinatorial_derivation
Component	visit_component
ComponentReference	visit_component_reference
Constraint	visit_constraint
Cut	visit_cut
Document	visit_document(self)
EntireSequence	visit_entire_sequence
Experiment	visit_experiment
ExperimentalData	visit_experimental_data
ExternallyDefined	visit_externally_defined
Implementation	visit_implementation
Interaction	visit_interaction
Interface	visit_interface
LocalSubComponent	visit_local_sub_component
Measure	visit_measure
Model	visit_model
Participation	visit_participation
Plan	visit_plan
PrefixedUnit	visit_prefixed_unit
Range	visit_range
SIPrefix	visit_si_prefix
Sequence	visit_sequence
SequenceFeature	visit_sequence_feature
SingularUnit	visit_singular_unit
SubComponent	visit_sub_component
UnitDivision	visit_unit_division
UnitExponentiation	visit_unit_exponentiation
UnitMultiplication	visit_unit_multiplication

continues on next page

Table 1 – continued from previous page

Class	Visit Method
Usage	visit_usage
VariableFeature	visit_variable_feature

CHAPTER

SEVEN

USING ONTOLOGY TERMS

SBOL leans heavily on a variety of ontologies for terminology. Examples include PROV-O for provenance terms, and Ontology of units of Measure for defining and using measures and units. The most commonly used terms are defined as pySBOL3 constants. These only scratch the surface of what is available.

TYTO is a Python module that automates the lookup of ontology terms so that you do not have to remember long, sometimes meaningless URIs. Here is an example of ontology lookup using TYTO:

```
>>> import tyto
RDFLib Version: 5.0.0
>>> tyto.SO.promoter
'https://identifiers.org/SO:0000167'
>>> tyto.SBO.systems_biology_representation
'https://identifiers.org/SBO:0000000'
```

TYTO and pySBOL3 will happily coexist and work together. TYTO can be used to look up some of the same terms that pySBOL3 defines as constants. For example:

```
>>> import sbol3
RDFLib Version: 5.0.0
>>> import tyto
>>> sbol3.SBO_DNA == tyto.SBO.deoxyribonucleic_acid
True
>>> sbol3.SBO_RNA == tyto.SBO.ribonucleic_acid
True
>>> sbol3.SO_PROMOTER == tyto.SO.promoter
True
```

7.1 TYTO Installation

TYTO can be installed using pip, Python’s package installer.

```
pip install tyto
```

CHAPTER

EIGHT

API REFERENCE

This page contains auto-generated API reference documentation[1].

8.1 `sbol3`

8.1.1 Submodules

`sbol3.attachment`

Module Contents

Classes

`Attachment`	The purpose of the Attachment class is to serve as a general

Functions

`build_attachment`(→ SBOLObject)

`class Attachment`(*identity: str*, *source: str*, ***, *format: Optional[str] = None*, *size: int = None*, *hash: str = None*, *hash_algorithm: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = SBOL_ATTACHMENT*)

Bases: `TopLevel`

The purpose of the Attachment class is to serve as a general container for data files, especially experimental data files. It provides a means for linking files and metadata to SBOL designs.

`accept`(*visitor: Any*) → Any

Invokes *visit_attachment* on *visitor* with *self* as the only argument.

Parameters

`visitor` (*Any*) – The visitor instance

[1] Created with sphinx-autoapi

Raises
`AttributeError` – If visitor lacks a visit_attachment method

Returns
Whatever *visitor.visit_attachment* returns

Return type
Any

`validate`(*report: ValidationReport = None*) → *ValidationReport*

`build_attachment`(*identity: str, *, type_uri: str = SBOL_COMPONENT*) → *SBOLObject*

`sbol3.boolean_property`

Module Contents

Classes

`BooleanListProperty`
`BooleanPropertyMixin`
`BooleanSingletonProperty`

Functions

`BooleanProperty`(→ sbol3.Property)

`class` **`BooleanListProperty`**(*property_owner: Any, property_uri: str, lower_bound: int, upper_bound: int, validation_rules: Optional[List] = None, initial_value: Optional[List[bool]] = None*)

Bases: *`BooleanPropertyMixin`*, `sbol3.ListProperty`

`BooleanProperty`(*property_owner: Any, property_uri: str, lower_bound: int, upper_bound: Union[int, float], validation_rules: Optional[List] = None, initial_value: Optional[Union[bool, List[bool]]] = None*) → sbol3.Property

`class` **`BooleanPropertyMixin`**

`from_user`(*value: Any*) → Union[None, rdflib.Literal]

`to_user`(*value: Any*) → bool

`class` **`BooleanSingletonProperty`**(*property_owner: Any, property_uri: str, lower_bound: int, upper_bound: int, validation_rules: Optional[List] = None, initial_value: Optional[bool] = None*)

Bases: *`BooleanPropertyMixin`*, `sbol3.SingletonProperty` `sbol3.collection`

Module Contents

Classes

Collection	The Collection class is a class that groups together a set of
Experiment	The purpose of the Experiment class is to aggregate

class Collection(*identity: str*, ***, *members: List[str] = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = SBOL_COLLECTION*)

Bases: `TopLevel`

The Collection class is a class that groups together a set of TopLevel objects that have something in common.

Some examples of Collection objects:

- Results of a query to find all Component objects in a repository that function as promoters.
- A set of Component objects representing a library of genetic logic gates.
- A "parts list" for Component with a complex design, containing both that component and all of the Component, Sequence, and Model objects used to provide its full specification.

accept(*visitor: Any*) → Any

Invokes *visit_collection* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_collection method

Returns
Whatever *visitor.visit_collection* returns

Return type
Any

class Experiment(*identity: str*, ***, *members: List[str] = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = SBOL_EXPERIMENT*)

Bases: *Collection*

The purpose of the Experiment class is to aggregate ExperimentalData objects for subsequent analysis, usually in accordance with an experimental design. Namely, the member properties of an Experiment MUST refer to ExperimentalData objects.

accept(*visitor: Any*) → Any

Invokes *visit_experiment* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_experiment method

Returns
Whatever *visitor.visit_experiment* returns

Return type
Any

sbol3.combderiv

Module Contents

Classes

CombinatorialDerivation	The purpose of the CombinatorialDerivation class is to specify

Functions

build_combinatorial_derivation(identity, *[, type_uri])

class CombinatorialDerivation(*identity: str*, *template: Union[sbol3.Component, str]*, *, *strategy: Optional[str] = None*, *variable_features: List[str] = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = SBOL_COMBINATORIAL_DERIVATION*)

Bases: `sbol3.TopLevel`

The purpose of the CombinatorialDerivation class is to specify combinatorial biological designs without having to specify every possible design variant. For example, a CombinatorialDerivation can be used to specify a library of reporter gene variants that include different promoters and RBSs without having to specify a Component for every possible combination of promoter, RBS, and CDS in the library. Component objects that realize a CombinatorialDerivation can be derived in accordance with the class properties template, hasVariableFeature, and strategy.

accept(*visitor: Any*) → Any

Invokes *visit_combinatorial_derivation* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_combinatorial_derivation method

Returns
Whatever *visitor.visit_combinatorial_derivation* returns

Return type
Any

validate(*report: sbol3.ValidationReport = None*) → sbol3.ValidationReport

build_combinatorial_derivation(*identity: str*, ***, *type_uri: str = SBOL_COMBINATORIAL_DERIVATION*)

`sbol3.component`

Module Contents

Classes

Component

Functions

build_component(→ SBOLObject)

class Component(*identity: str*, *types: Optional[Union[str, Sequence[str]]]*, ***, *roles: Optional[Union[str, Sequence[str]]] = None*, *sequences: Optional[refobj_list_arg] = None*, *features: Union[Feature, Sequence[Feature]] = None*, *constraints: Union[Constraint, Sequence[Constraint]] = None*, *interactions: Union[Interaction, Sequence[Interaction]] = None*, *interface: Union[Interface, Sequence[Interface]] = None*, *models: Optional[refobj_list_arg] = None*, *namespace: Optional[str] = None*, *attachments: Optional[refobj_list_arg] = None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *derived_from: Optional[Union[str, Sequence[str]]] = None*, *generated_by: Optional[refobj_list_arg] = None*, *measures: Optional[ownedobj_list_arg] = None*, *type_uri: str = SBOL_COMPONENT*)

Bases: `TopLevel`

accept(*visitor: Any*) → Any

Invokes *visit_component* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_component method

Returns
Whatever *visitor.visit_component* returns

Return type
Any

build_component(*identity: str*, ***, *type_uri: str = SBOL_COMPONENT*) → *SBOLObject* `sbol3.compref`

Module Contents

Classes

ComponentReference	ComponentReference can be used to reference Features within

Functions

build_component_reference(→ SBOLObject)	

class ComponentReference(*in_child_of: Union[SubComponent, str], refers_to: Union[sbol3.feature.Feature, str], *, roles: List[str] = None, orientation: str = None, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[SBOLObject] = None, identity: str = None, type_uri: str = SBOL_COMPONENT_REFERENCE*)

Bases: *sbol3.feature.Feature*

ComponentReference can be used to reference Features within SubComponents.

accept(*visitor: Any*) → Any

Invokes *visit_component_reference* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_component_reference method

Returns
Whatever *visitor.visit_component_reference* returns

Return type
Any

validate(*report: ValidationReport = None*) → *ValidationReport*

build_component_reference(*identity: str, *, type_uri: str = SBOL_COMPONENT_REFERENCE*) → *SBOLObject*

sbol3.config

Module Contents

Functions

get_namespace(→ Optional[str])	
set_defaults(→ None)	Restores configuration to factory settings.
set_namespace(→ None)	

Attributes

SBOL3_NAMESPACE

SBOL3_NAMESPACE

get_namespace() → Optional[str]

set_defaults() → None

Restores configuration to factory settings.

set_namespace(*namespace: Optional[str]*) → None

sbol3.constants

Module Contents

CHEBI_EFFECTOR

CHEBI_NS = 'https://identifiers.org/CHEBI:'

INCHI_ENCODING = 'https://identifiers.org/edam:format_1197'

IUPAC_DNA_ENCODING = 'https://identifiers.org/edam:format_1207'

IUPAC_PROTEIN_ENCODING = 'https://identifiers.org/edam:format_1208'

IUPAC_RNA_ENCODING = 'https://identifiers.org/edam:format_1207'

JSONLD = 'json-ld'

NTRIPLES = 'nt11'

OM_ALTERNATIVE_LABEL

OM_ALTERNATIVE_SYMBOL

OM_BINARY_PREFIX

OM_COMMENT

OM_HAS_BASE

OM_HAS_DENOMINATOR

OM_HAS_EXPONENT

OM_HAS_FACTOR

OM_HAS_NUMERATOR

OM_HAS_NUMERICAL_VALUE

OM_HAS_PREFIX

OM_HAS_TERM1

OM_HAS_TERM2

OM_HAS_UNIT

OM_LABEL

OM_LONG_COMMENT

OM_MEASURE

OM_NS = 'http://www.ontology-of-units-of-measure.org/resource/om-2/'

OM_PREFIXED_UNIT

OM_SINGULAR_UNIT

OM_SI_PREFIX

OM_SYMBOL

OM_UNIT_DIVISION

OM_UNIT_EXPONENTIATION

OM_UNIT_MULTIPLICATION

PROV_ACTIVITY

PROV_AGENT

PROV_AGENTS

PROV_ASSOCIATION

PROV_DERIVED_FROM

PROV_ENDED_AT_TIME

PROV_ENTITY

PROV_GENERATED_BY

PROV_NS = 'http://www.w3.org/ns/prov#'

PROV_PLAN

PROV_PLANS

PROV_QUALIFIED_ASSOCIATION

PROV_QUALIFIED_USAGE

PROV_ROLES

PROV_STARTED_AT_TIME

PROV_USAGE

PYSBOL3_DEFAULT_NAMESPACE = 'http://sbols.org/unspecified_namespace/'

PYSBOL3_MISSING

PYSBOL3_NS = 'https://github.com/synbiodex/pysbol3#'

RDF_NS = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'

RDF_TYPE

RDF_XML = 'xml'

SBOL1_NS = 'http://sbols.org/v1#'

SBOL2_NS = 'http://sbols.org/v2#'

SBOL3_NS = 'http://sbols.org/v3#'

SBOL_ATTACHMENT

SBOL_BUILD

SBOL_BUILT

SBOL_CARDINALITY

SBOL_COLLECTION

SBOL_COMBINATORIAL_DERIVATION

SBOL_COMPONENT

SBOL_COMPONENT_REFERENCE

SBOL_CONSTRAINT

SBOL_CONSTRAINTS

SBOL_CONTAINS

SBOL_COVERS

SBOL_CUT

SBOL_DEFINITION

SBOL_DESCRIPTION

SBOL_DESIGN

SBOL_DIFFERENT_FROM

SBOL_DISPLAY_ID

SBOL_ELEMENTS

SBOL_ENCODING

SBOL_END

SBOL_ENTIRE_SEQUENCE

SBOL_ENUMERATE

SBOL_EQUALS

SBOL_EXPERIMENT

SBOL_EXPERIMENTAL_DATA

SBOL_EXTERNALLY_DEFINED

SBOL_FEATURES

SBOL_FINISHES

SBOL_FORMAT

SBOL_FRAMEWORK

SBOL_HASH

SBOL_HASH_ALGORITHM

SBOL_HAS_ATTACHMENT

SBOL_HAS_MEASURE

SBOL_IDENTIFIED

SBOL_IMPLEMENTATION

SBOL_INLINE

SBOL_INPUT

SBOL_INSTANCE_OF

SBOL_INTERACTION

SBOL_INTERACTIONS

SBOL_INTERFACE

SBOL_INTERFACES

SBOL_IN_CHILD_OF

SBOL_IS_DISJOINT_FROM

SBOL_LANGUAGE

SBOL_LEARN

SBOL_LOCAL_SUBCOMPONENT

SBOL_LOCATION

SBOL_LOGGER_NAME = 'sbol3'

SBOL_MEETS

SBOL_MEMBER

SBOL_MODEL

SBOL_MODELS

SBOL_NAME

SBOL_NAMESPACE

SBOL_NONDIRECTIONAL

SBOL_OBJECT

SBOL_ONE

SBOL_ONE_OR_MORE

SBOL_OPPOSITE_ORIENTATION_AS

SBOL_ORDER

SBOL_ORIENTATION

SBOL_OUTPUT

SBOL_OVERLAPS

SBOL_PARTCIPATION

SBOL_PARTICIPANT

SBOL_PARTICIPATIONS

SBOL_PRECEDES

SBOL_RANGE

SBOL_REFERS_TO

SBOL_REPLACES

SBOL_RESTRICTION

SBOL_REVERSE_COMPLEMENT

SBOL_ROLE

SBOL_SAME_ORIENTATION_AS

SBOL_SAMPLE

SBOL_SEQUENCE

SBOL_SEQUENCES

SBOL_SEQUENCE_FEATURE

SBOL_SIZE

SBOL_SOURCE

SBOL_SOURCE_LOCATION

SBOL_START

SBOL_STARTS

SBOL_STRATEGY

SBOL_STRICTLY_CONTAINS

SBOL_STRICTLY_PRECEDES

SBOL_SUBCOMPONENT

SBOL_SUBJECT

SBOL_TEMPLATE

SBOL_TEST

SBOL_TOP_LEVEL

SBOL_TYPE

SBOL_VARIABLE

SBOL_VARIABLE_FEATURE

SBOL_VARIABLE_FEATURES

SBOL_VARIANT

SBOL_VARIANT_COLLECTION

SBOL_VARIANT_DERIVATION

SBOL_VARIANT_MEASURE

SBOL_VERIFY_IDENTICAL

SBOL_ZERO_OR_MORE

SBOL_ZERO_OR_ONE

SBO_BIOCHEMICAL_REACTION

SBO_CONTROL

SBO_DEGRADATION

SBO_DNA

SBO_FUNCTIONAL_ENTITY

SBO_GENETIC_PRODUCTION

SBO_INHIBITED

SBO_INHIBITION

SBO_INHIBITOR

SBO_MODIFIED

SBO_MODIFIER

SBO_NON_COVALENT_BINDING

SBO_NON_COVALENT_COMPLEX

SBO_NS = 'https://identifiers.org/SBO:'

SBO_PRODUCT

SBO_PROMOTER

SBO_PROTEIN

SBO_REACTANT

SBO_RNA

SBO_SIMPLE_CHEMICAL

SBO_STIMULATED

SBO_STIMULATION

SBO_STIMULATOR

SBO_TEMPLATE

SMILES_ENCODING = 'https://identifiers.org/edam:format_1196'

SORTED_NTRIPLES = 'sorted nt'

SO_CDS

SO_CIRCULAR

SO_DOUBLE_STRANDED

SO_ENGINEERED_GENE

SO_ENGINEERED_REGION

SO_FORWARD

SO_GENE

SO_LINEAR

`SO_MRNA`

`SO_NS = 'https://identifiers.org/SO:'`

`SO_OPERATOR`

`SO_PROMOTER`

`SO_RBS`

`SO_REVERSE`

`SO_SINGLE_STRANDED`

`SO_TERMINATOR`

`SO_TRANSCRIPTION_FACTOR`

`TURTLE = 'ttl'`

`sbol3.constraint`

Module Contents

Classes

`Constraint`	The Constraint class can be used to assert restrictions on the

Functions

`build_constraint`(→ SBOLObject)	

class `Constraint`(*restriction: str*, *subject: Union[Identified, str]*, *object: Union[Identified, str]*, ***, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *identity: Optional[str] = None*, *type_uri: str = SBOL_CONSTRAINT*)

Bases: `Identified`

The Constraint class can be used to assert restrictions on the relationships of pairs of Feature objects contained by the same parent Component. Uses of this class include expressing containment (e.g., a plasmid transformed into a chassis strain), identity mappings (e.g., replacing a placeholder value with a complete definition), and expressing relative, sequence-based positions (e.g., the ordering of features within a template). Each Constraint includes the subject, object, and restriction properties.

`accept`(*visitor: Any*) → Any

Invokes *visit_constraint* on *visitor* with *self* as the only argument.

Parameters

`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_constraint method

Returns
Whatever *visitor.visit_constraint* returns

Return type
Any

validate(*report: ValidationReport = None*) → *ValidationReport*

build_constraint(*identity: str, type_uri: str = SBOL_CONSTRAINT*) → *SBOLObject*

sbol3.custom

Module Contents

Classes

CustomIdentified	
CustomTopLevel	

class CustomIdentified(*type_uri: str = None, *, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[SBOLObject] = None, identity: str = None, sbol_type_uri: str = SBOL_IDENTIFIED*)

Bases: `Identified`

validate(*report: ValidationReport = None*) → *ValidationReport*

class CustomTopLevel(*identity: str = None, type_uri: str = None, *, namespace: str = None, attachments: List[str] = None, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[SBOLObject] = None, sbol_type_uri: str = SBOL_TOP_LEVEL*)

Bases: `TopLevel`

validate(*report: ValidationReport = None*) → *ValidationReport*

sbol3.datetime_property

Module Contents

Classes

DateTimePropertyMixin	
DateTimeSingletonProperty	Helper class that provides a standard way to create an ABC using

Functions

DateTimeProperty(→ sbol3.Property)	

DateTimeProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[int, List[int]]] = None*) → sbol3.Property

class DateTimePropertyMixin

from_user(*value*)

to_user(*value*)

class DateTimeSingletonProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[str] = None*)

Bases: *DateTimePropertyMixin*, `sbol3.SingletonProperty`

Helper class that provides a standard way to create an ABC using inheritance.

sbol3.document

Module Contents

Classes

Document

Functions

copy(→ List[TopLevel])	Copy SBOL objects, optionally changing their namespace and
data_path(→ str)	Expand path based on module installation directory.

class Document

__contains__(*item*)

__iter__()

Iterate over the top level objects in this document.

```
>>> import sbol3
>>> doc = sbol3.Document()
>>> doc.read('some_path.ttl')
>>> for top_level in doc:
>>>     print(top_level.identity)
```

Returns
An iterator over the top level objects

`__len__()`

Get the total number of objects in the Document.

(Returns the same thing as size())

Returns
The total number of objects in the Document.

`__str__()`

Produce a string representation of the Document.

Returns
A string representation of the Document.

`accept`(*visitor: Any*) → Any

Invokes *visit_document* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_document method

Returns
Whatever *visitor.visit_document* returns

Return type
Any

`add`(*objects: Union[*TopLevel*,* Sequence*[*TopLevel*]]*) → Union[*TopLevel*, *Sequence*[*TopLevel*]]

`addNamespace`(*namespace: str*, *prefix: str*) → None

Document.addNamespace is deprecated. Replace with Document.bind.

Document.addNamespace existed in pySBOL2 and was commonly used.

Document.addNamespace(namespace, prefix) should now be Document.bind(prefix, namespace). Note the change of argument order.

`bind`(*prefix: str*, *uri: str*) → None

Bind a prefix to an RDF namespace in the written RDF document.

These prefixes make the written RDF easier for humans to read. These prefixes do not change the semantic meaning of the RDF document in any way.

`builder`(*type_uri: str*) → Callable[[str, str], *Identified*]

Lookup up the builder callable for the given type_uri.

The builder must have been previously registered under this type_uri via Document.register_builder().

Raises
ValueError if the type_uri does not have an associated builder.

`static change_object_namespace`(*top_levels: Iterable[*TopLevel*]*, *new_namespace: str*, *update_references: Iterable[*TopLevel*] = None*) → Any

Change the namespace of all TopLevel objects in *top_levels* to new_namespace, regardless of the previous value, while maintaining referential integrity among all the top level objects in top_levels, including their dependents. The namespace change is "in place". No new objects are allocated.

Note: this operation can result in an invalid Document if the change in namespace creates a naming collision. This method does not check for this case either before or after the operation. It is up to the caller to decide whether this operation is safe.

Parameters

- `top_levels` – objects to change
- `new_namespace` – new namespace for objects
- `update_references` – objects that should have their references updated without changing their namespace

Returns
Nothing

`clear()` → None

`clone()` → List[*TopLevel*]

Clone the top level objects in this document.

Returns
A list of cloned TopLevel objects

`copy()` → *Document*

Make a copy of this document.

Returns
A new document containing a new set of objects that are identical to the original objects.

`static file_extension(`*file_format: str*`)` → str

Return standard extensions when provided the document's file format

Parameters
`file_format` – The format of the file

Returns
A file extension, including the leading '.'

`find(`*search_string: str*`)` → Optional[*Identified*]

Find an object by identity URI or by display_id.

Parameters
`search_string` (*str*) – Either an identity URI or a display_id

Returns
The named object or `None` if no object was found

`find_all(`*predicate: Callable[[*Identified*], bool]*`)` → List[*Identified*]

Executes a predicate on every object in the document tree, gathering the list of objects to which the predicate returns true.

`graph()` → rdflib.Graph

Convert document to an RDF Graph.

The returned graph is a snapshot of the document and will not be updated by subsequent changes to the document.

`join_lines(`*lines: List[Union[bytes, str]]*`)` → Union[bytes, str]

Join lines for either bytes or strings. Joins a list of lines together whether they are bytes or strings. Returns a bytes if the input was a list of bytes, and a str if the input was a list of str.

`migrate`(*top_levels: Iterable[TopLevel]*) → Any

Migrate objects to this document.

No effort is made to maintain referential integrity. The burden of referential integrity lies with the caller of this method.

Parameters
`top_levels` – The top levels to migrate to this document

Returns
Nothing

`static open`(*location: Union[pathlib.Path, str]*, *file_format: str = None*) → *Document*

`parse_shacl_graph`(*shacl_graph: rdflib.Graph*, *report:* ValidationReport) → *ValidationReport*

Convert SHACL violations and warnings into a pySBOL3 validation report.

Parameters

- `shacl_graph` (*`rdflib.Graph`*) – The output graph from pyshacl
- `report` (`ValidationReport`) – The ValidationReport to be populated

Returns
report

Return type
ValidationReport

`read`(*location: Union[pathlib.Path, str]*, *file_format: str = None*) → None

`read_string`(*data: str*, *file_format: str*) → None

`static register_builder`(*type_uri: str*, *builder: Callable[[str, str],* Identified*]*) → None

A builder function will be called with an identity and a keyword argument type_uri.

builder(identity_uri: str, type_uri: str = None) -> SBOLObject

`remove`(*objects: Iterable[TopLevel]*)

`remove_object`(*top_level:* TopLevel)

Removes the given TopLevel from this document. No referential integrity is updated, and the TopLevel object is not informed that it has been removed, so it may still have a pointer to this document. No errors are raised and no value is returned.

N.B. You probably want to use *remove* instead of *remove_object*.

Parameters
`top_level` – An object to remove

Returns
Nothing

`size`()

Get the total number of objects in the Document.

Returns
The total number of objects in the Document.

`summary`()

Produce a string representation of the Document. :return: A string representation of the Document.

traverse(*func: Callable[[Identified], None]*)

Enable a traversal of the entire object hierarchy contained in this document.

validate(*report: ValidationReport = None*) → *ValidationReport*

Validate all objects in this document.

validate_shacl(*report: Optional[ValidationReport] = None*) → *ValidationReport*

Validate this document using SHACL rules.

write(*fpath: Union[pathlib.Path, str]*, *file_format: str = None*) → None

Write the document to file.

If file_format is None the desired format is guessed from the extension of fpath. If file_format cannot be guessed a ValueError is raised.

write_string(*file_format: str*) → str

copy(*top_levels: Iterable[TopLevel]*, *into_namespace: Optional[str] = None*, *into_document: Optional[Document] = None*) → List[*TopLevel*]

Copy SBOL objects, optionally changing their namespace and optionally adding them to a document. Referential integrity among the group of provided TopLevel objects is maintained.

If *new_namespace* is provided, the newly created objects will have the provided namespace and will maintain the rest of their identities, including the local path and diplay ID.

If *new_document* is provided, the newly created objects will be added to the provided Document.

Parameters

- **top_levels** – Top Level objects to be copied
- **into_namespace** – A namespace to be given to the new objects
- **into_document** – A document to which the newly created objects will be added

Returns

A list of the newly created objects

data_path(*path: str*) → str

Expand path based on module installation directory.

Parameters

path –

Returns

sbol3.error

Module Contents

exception Error

Bases: Exception

Base class for exceptions in this module.

Initialize self. See help(type(self)) for accurate signature.

exception NamespaceError(*message: str*)

Bases: *Error*

Exception raised for namespace errors.

Attributes:
message – explanation of the error

Initialize self. See help(type(self)) for accurate signature.

exception SBOLError(*message: str*)

Bases: *Error*

Base class for exceptions in this module.

Initialize self. See help(type(self)) for accurate signature.

sbol3.expdata

Module Contents

Classes

ExperimentalData	The purpose of the ExperimentalData class is to aggregate links to

class ExperimentalData(*identity: str*, ***, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = SBOL_EXPERIMENTAL_DATA*)

Bases: `TopLevel`

The purpose of the ExperimentalData class is to aggregate links to experimental data files. An ExperimentalData is typically associated with a single sample, lab instrument, or experimental condition and can be used to describe the output of the test phase of a design-build-test-learn workflow.

accept(*visitor: Any*) → Any

Invokes *visit_experimental_data* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_experimental_data method

Returns
Whatever *visitor.visit_experimental_data* returns

Return type
Any

`sbol3.extdef`

Module Contents

Classes

ExternallyDefined	The ExternallyDefined class has been introduced so that

Functions

build_externally_defined(→ SBOLObject)	

class ExternallyDefined(*types: Union[str, list[str]], definition: str, *, roles: List[str] = None, orientation: str = None, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[*SBOLObject*] = None, identity: str = None, type_uri: str = SBOL_EXTERNALLY_DEFINED*)

Bases: *sbol3.feature.Feature*

The ExternallyDefined class has been introduced so that external definitions in databases like ChEBI or UniProt can be referenced.

accept(*visitor: Any*) → Any

Invokes *visit_externally_defined* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_externally_defined method

Returns
Whatever *visitor.visit_externally_defined* returns

Return type
Any

validate(*report:* ValidationReport *= None*) → *ValidationReport*

build_externally_defined(*identity: str, *, type_uri: str = SBOL_EXTERNALLY_DEFINED*) → *SBOLObject*

`sbol3.feature`

Module Contents

Classes

Feature	Feature is an abstract base class.

class Feature(*identity: str*, *type_uri: str*, ***, *roles: Optional[str, list[str]] = None*, *orientation: Optional[str] = None*, *name: str = None*, *description: str = None*, *derived_from: sbol3.typing.List[str] = None*, *generated_by: sbol3.typing.List[str] = None*, *measures: sbol3.typing.List[sbol3.SBOLObject] = None*)

Bases: `sbol3.Identified`, `abc.ABC`

Feature is an abstract base class.

Parameters

- **identity** – this object's Uniform Resource Identifier (URI). this URI MUST be globally unique among all other Identified object URIs. See SBOL 3.0.1 specification section 5.1. This can also be a *displayId*, which will be concatenated to a default namespace automatically.
- **type_uri** – the concrete type of this object, specified as a URI. These are typically in the SBOL3 namespace, like *http://sbols.org/v3#Sequence* or *http://sbols.org/v3#Component*. This can also be the type URI of an extension class.
- **name** – A human-readable name for this object, for display purposes.
- **description** – Per the SBOL 3.0.1 specification, "a more thorough text description" of this object.
- **derived_from** – The URIs of one or more SBOL or non-SBOL objects from which this object was derived. This property is defined by the PROV-O ontology.
- **generated_by** – The URIs of one or more prov:Activity objects that describe how this object was generated. This property is defined by the PROV-O ontology.
- **measures** – The URIs of one or more om:Measure objects, each of which refers to a om:Measure object that describes measured parameters for this object. om:Measure objects are defined by the OM ontology

validate(*report: sbol3.ValidationReport = None*) → sbol3.ValidationReport

`sbol3.float_property`

Module Contents

Classes

FloatPropertyMixin	
FloatSingletonProperty	Helper class that provides a standard way to create an ABC using

Functions

FloatProperty(→ sbol3.Property)	

FloatProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[float, List[float]]] = None*) → sbol3.Property

class FloatPropertyMixin

from_user(*value: Union[float, int, None]*) → Union[None, rdflib.Literal]

to_user(*value: Any*) → float

class FloatSingletonProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Union[float, int, None] = None*)

Bases: *FloatPropertyMixin*, `sbol3.SingletonProperty`

Helper class that provides a standard way to create an ABC using inheritance.

sbol3.identified

Module Contents

Classes

Identified	All SBOL-defined classes are directly or indirectly derived from

Functions

extract_display_id(→ Union[None, str])	Determine the display ID for an object with the given identity.
is_valid_display_id(→ bool)	Determine if a given display ID is valid according to the SBOL specification.

class Identified(*identity: str*, *type_uri: str*, ***, *name: Optional[str] = None*, *description: Optional[str] = None*, *derived_from: Optional[Union[str, Sequence[str]]] = None*, *generated_by: Optional[refobj_list_arg] = None*, *measures: Optional[ownedobj_list_arg] = None*)

Bases: `sbol3.SBOLObject`

All SBOL-defined classes are directly or indirectly derived from the Identified abstract class. This inheritance means that all SBOL objects are uniquely identified using URIs that uniquely refer to these objects within an SBOL document or at locations on the World Wide Web.

Parameters

- `identity` – this object's Uniform Resource Identifier (URI). this URI MUST be globally unique among all other Identified object URIs. See SBOL 3.0.1 specification section 5.1. This can also be a *displayId*, which will be concatenated to a default namespace automatically.
- `type_uri` – the concrete type of this object, specified as a URI. These are typically in the SBOL3 namespace, like *http://sbols.org/v3#Sequence* or *http://sbols.org/v3#Component*. This can also be the type URI of an extension class.
- `name` – A human-readable name for this object, for display purposes.
- `description` – Per the SBOL 3.0.1 specification, "a more thorough text description" of this object.
- `derived_from` – The URIs of one or more SBOL or non-SBOL objects from which this object was derived. This property is defined by the PROV-O ontology.
- `generated_by` – The URIs of one or more prov:Activity objects that describe how this object was generated. This property is defined by the PROV-O ontology.
- `measures` – The URIs of one or more om:Measure objects, each of which refers to a om:Measure object that describes measured parameters for this object. om:Measure objects are defined by the OM ontology

`property display_id`

`property display_name`

`property document: Union[sbol3.Document, None]`

`property properties`

`property type_uri`

`__str__()`

`accept(visitor: Any)` → Any

An abstract method for concrete classes to override. This method is part of the visitor pattern implementation.

Parameters
`visitor` – Ignored

Returns
Unspecified

Raises
NotImplementedError if not overridden

`clear_property(uri)`

Clears the internal storage of a property based on the URI. This is for advanced usage only, and may cause inconsistent objects and/or graphs.

USE WITH CARE.

`counter_value(type_name: str)` → int

`remove_from_document()`

`serialize(graph: rdflib.Graph)`

`traverse`(*func: Callable[[*Identified*], None]*)

Enable a traversal of this object and all of its children by invoking the passed function on all objects.

`validate`(*report: sbol3.ValidationReport = None*) → sbol3.ValidationReport

`extract_display_id`(*identity: str*) → Union[None, str]

Determine the display ID for an object with the given identity.

The display ID is the final component of the path of a URL. If the provided identity is not a URL, then no display ID can be determined and None is returned.

Parameters
`identity` – An identity for an SBOL object

Returns
a valid display id or None of no display ID can be extracted

`is_valid_display_id`(*display_id: str*) → bool

Determine if a given display ID is valid according to the SBOL specification.

The SBOL3 specification states, "A display id [. . .] value MUST be composed of only alphanumeric or underscore characters and MUST NOT begin with a digit." (Section 6.1)

Parameters
`display_id` – The display ID to check for validity

Returns
True if the display ID is valid, False otherwise.

`sbol3.implementation`

Module Contents

Classes

`Implementation`	An Implementation represents a realized instance of a Component,

`class Implementation`(*identity: str*, ***, *built: Union[*Component*, str] = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[*SBOLObject*] = None*, *type_uri: str = SBOL_IMPLEMENTATION*)

Bases: `TopLevel`

An Implementation represents a realized instance of a Component, such a sample of DNA resulting from fabricating a genetic design or an aliquot of a specified reagent. Importantly, an Implementation can be associated with a laboratory sample that was already built, or that is planned to be built in the future. An Implementation can also represent virtual and simulated instances. An Implementation may be linked back to its original design using the prov:wasDerivedFrom property inherited from the Identified superclass. An Implementation may also link to a Component that specifies its realized structure and/or function.

`accept`(*visitor: Any*) → Any

Invokes *visit_implementation* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_implementation method

Returns
Whatever *visitor.visit_implementation* returns

Return type
Any

`sbol3.int_property`

Module Contents

Classes

IntListProperty	Helper class that provides a standard way to create an ABC using
IntPropertyMixin	
IntSingletonProperty	Helper class that provides a standard way to create an ABC using

Functions

IntProperty(→ sbol3.Property)	

class IntListProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[List[int]] = None*)

Bases: *IntPropertyMixin*, `sbol3.ListProperty`

Helper class that provides a standard way to create an ABC using inheritance.

IntProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[int, List[int]]] = None*) → sbol3.Property

class IntPropertyMixin

from_user(*value: Any*) → Union[None, rdflib.Literal]

to_user(*value: Any*) → int

class IntSingletonProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[int] = None*)

Bases: *IntPropertyMixin*, `sbol3.SingletonProperty`

Helper class that provides a standard way to create an ABC using inheritance.

`sbol3.interaction`

Module Contents

Classes

Interaction	The Interaction class provides more detailed description of how the

Functions

build_interaction(→ SBOLObject)	

class Interaction(*types: Union[str, list[str]]*, ***, *participations: List[*Participation*] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[*SBOLObject*] = None*, *identity: str = None*, *type_uri: str = SBOL_INTERACTION*)

Bases: `Identified`

The Interaction class provides more detailed description of how the Feature objects of a Component are intended to work together. For example, this class can be used to represent different forms of genetic regulation (e.g., transcriptional activation or repression), processes from the central dogma of biology (e.g. transcription and translation), and other basic molecular interactions (e.g., non-covalent binding or enzymatic phosphorylation). Each Interaction includes type properties that refer to descriptive ontology terms and hasParticipation properties that describe which Feature objects participate in which ways in the Interaction.

accept(*visitor: Any*) → Any

Invokes *visit_interaction* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_interaction method

Returns
Whatever *visitor.visit_interaction* returns

Return type
Any

build_interaction(*identity: str*, ***, *type_uri: str = SBOL_INTERACTION*) → *SBOLObject* `sbol3.interface`

Module Contents

Classes

`Interface`	The Interface class is a way of explicitly specifying the interface

Functions

`build_interface`(→ SBOLObject)	

`class Interface`(**, inputs: List[str] = None, outputs: List[str] = None, nondirectionals: List[str] = None, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[*SBOLObject*] = None, identity: str = None, type_uri: str = SBOL_INTERFACE*)

Bases: `Identified`

The Interface class is a way of explicitly specifying the interface of a Component.

`accept`(*visitor: Any*) → Any

Invokes *visit_interface* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*`Any`*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_interface method

Returns
Whatever *visitor.visit_interface* returns

Return type
Any

`build_interface`(*identity: str, *, type_uri: str = SBOL_INTERFACE*) → *SBOLObject*

`sbol3.localsub`

Module Contents

Classes

`LocalSubComponent`	LocalSubComponent serves as a way to create a placeholder in more

Functions

`build_local_subcomponent`(→ sbol3.SBOLObject)	

class LocalSubComponent(*types: sbol3.typing.Union[str, list[str]], *, locations: sbol3.typing.Union[sbol3.Location, sbol3.typing.List[sbol3.Location]] = None, roles: sbol3.typing.List[str] = None, orientation: str = None, name: str = None, description: str = None, derived_from: sbol3.typing.List[str] = None, generated_by: sbol3.typing.List[str] = None, measures: sbol3.typing.List[sbol3.SBOLObject] = None, identity: str = None, type_uri: str = SBOL_LOCAL_SUBCOMPONENT*)

Bases: `sbol3.feature.Feature`

LocalSubComponent serves as a way to create a placeholder in more complex Components, such as a variable to be filled in later or a composite that exists only within the context of the parent Component.

Parameters

- **identity** – this object's Uniform Resource Identifier (URI). this URI MUST be globally unique among all other Identified object URIs. See SBOL 3.0.1 specification section 5.1. This can also be a *displayId*, which will be concatenated to a default namespace automatically.
- **type_uri** – the concrete type of this object, specified as a URI. These are typically in the SBOL3 namespace, like *http://sbols.org/v3#Sequence* or *http://sbols.org/v3#Component*. This can also be the type URI of an extension class.
- **name** – A human-readable name for this object, for display purposes.
- **description** – Per the SBOL 3.0.1 specification, "a more thorough text description" of this object.
- **derived_from** – The URIs of one or more SBOL or non-SBOL objects from which this object was derived. This property is defined by the PROV-O ontology.
- **generated_by** – The URIs of one or more prov:Activity objects that describe how this object was generated. This property is defined by the PROV-O ontology.
- **measures** – The URIs of one or more om:Measure objects, each of which refers to a om:Measure object that describes measured parameters for this object. om:Measure objects are defined by the OM ontology

accept(*visitor: Any*) → Any

Invokes *visit_local_sub_component* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_local_sub_component method

Returns
Whatever *visitor.visit_local_sub_component* returns

Return type
Any

build_local_subcomponent(*identity: str, *, type_uri: str = SBOL_LOCAL_SUBCOMPONENT*) → sbol3.SBOLObject `sbol3.location`

Module Contents

Classes

Cut	The Cut class has been introduced to enable the specification of a
EntireSequence	The EntireSequence class does not have any additional
Location	The Location class is used to represent the location of Features
Range	A Range object specifies a region via discrete, inclusive start and

Functions

build_cut(identity[, type_uri])	Used by Document to construct a Cut when reading an SBOL file.
build_entire_sequence(identity[, type_uri])	Used by Document to construct an EntireSequence when reading an SBOL file.
build_range(identity[, type_uri])	Used by Document to construct a Range when reading an SBOL file.

Attributes

int_property

class Cut(*sequence: Union[*Sequence*, str]*, *at: int*, ***, *orientation: str = None*, *order: int = None*, *identity: str = None*, *type_uri: str = SBOL_CUT*)

Bases: *Location*

The Cut class has been introduced to enable the specification of a region between two discrete positions. This specification is accomplished using the at property, which specifies a discrete position that corresponds to the index of a character in the elements String of a Sequence (except in the case when at is equal to zero).

accept(*visitor: Any*) → Any

Invokes *visit_cut* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_cut method

Returns
Whatever *visitor.visit_cut* returns

Return type
Any

validate(*report: ValidationReport = None*) → *ValidationReport*

class EntireSequence(*sequence: Union[Sequence, str]*, ***, *orientation: str = None*, *order: int = None*, *identity: str = None*, *type_uri: str = SBOL_ENTIRE_SEQUENCE*)

Bases: `Location`

The EntireSequence class does not have any additional properties. Use of this class indicates that the linked Sequence describes the entirety of the Component or Feature parent of this Location object.

accept(*visitor: Any*) → Any

Invokes *visit_entire_sequence* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_entire_sequence method

Returns
Whatever *visitor.visit_entire_sequence* returns

Return type
Any

class Location(*sequence: Union[Sequence, str]*, *identity: str*, *type_uri: str*, ***, *orientation: Optional[str] = None*, *order: int = None*)

Bases: `Identified`, `abc.ABC`

The Location class is used to represent the location of Features within Sequences. This class is extended by the Range, Cut, and EntireSequence classes

validate(*report: ValidationReport = None*) → *ValidationReport*

class Range(*sequence: Union[Sequence, str]*, *start: int*, *end: int*, ***, *orientation: str = None*, *order: int = None*, *identity: str = None*, *type_uri: str = SBOL_RANGE*)

Bases: `Location`

A Range object specifies a region via discrete, inclusive start and end positions that correspond to indices for characters in the elements String of a Sequence.

Note that the index of the first location is 1, as is typical practice in biology, rather than 0, as is typical practice in computer science.

accept(*visitor: Any*) → Any

Invokes *visit_range* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_range method

Returns
Whatever *visitor.visit_range* returns

Return type
Any

validate(*report: ValidationReport = None*) → *ValidationReport*

build_cut(*identity: str*, *type_uri: str = SBOL_CUT*)

Used by Document to construct a Cut when reading an SBOL file.

build_entire_sequence(*identity: str*, *type_uri: str = SBOL_ENTIRE_SEQUENCE*)

Used by Document to construct an EntireSequence when reading an SBOL file.

build_range(*identity: str*, *type_uri: str = SBOL_RANGE*)

Used by Document to construct a Range when reading an SBOL file.

int_property

sbol3.model

Module Contents

Classes

Model	The purpose of the Model class is to serve as a placeholder for an

Functions

build_model(identity[, type_uri])	Used by Document to construct a Range when reading an SBOL file.

class Model(*identity: str*, *source: str*, *language: str*, *framework: str*, ***, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = SBOL_MODEL*)

Bases: `sbol3.TopLevel`

The purpose of the Model class is to serve as a placeholder for an external computational model and provide additional meta-data to enable better reasoning about the contents of this model. In this way, there is minimal duplication of standardization efforts and users of SBOL can elaborate descriptions of Component function in the language of their choice.

accept(*visitor: Any*) → Any

Invokes *visit_model* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_model method

Returns
Whatever *visitor.visit_model* returns

Return type
Any

validate(*report: sbol3.ValidationReport = None*) → sbol3.ValidationReport

build_model(*identity: str*, *type_uri: str = SBOL_MODEL*)

Used by Document to construct a Range when reading an SBOL file.

sbol3.object

Module Contents

Classes

SBOLObject

Functions

replace_namespace(old_uri, target_namespace, rdf_type)

Attributes

BUILDER_REGISTER

BUILDER_REGISTER: Dict[str, Callable[[str, str], ***SBOLObject*****]]**

class SBOLObject(*name: str*)

property identity: str

__getattribute__(*name*)

Return getattr(self, name).

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

copy(*target_doc=None*, *target_namespace=None*)

find(*search_string: str*) → Optional[*SBOLObject*]

identity_is_url() → bool

replace_namespace(*old_uri*, *target_namespace*, *rdf_type*) `sbol3.om_compound`

Module Contents

Classes

CompoundUnit	As adopted by SBOL, om:CompoundUnit is an abstract class that is
UnitDivision	The purpose of the om:UnitDivision class is to describe a unit of
UnitExponentiation	The purpose of the om:UnitExponentiation class is to describe a
UnitMultiplication	The purpose of the om:UnitMultiplication class is to describe a

Functions

build_unit_division(→ sbol3.SBOLObject)	
build_unit_exponentiation(→ sbol3.SBOLObject)	
build_unit_multiplication(→ sbol3.SBOLObject)	

class CompoundUnit(*identity: str, symbol: str, label: str, type_uri: str, *, alternative_symbols: List[str] = None, alternative_labels: List[str] = None, comment: str = None, long_comment: str = None, namespace: str = None, attachments: List[str] = None, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[sbol3.SBOLObject] = None*)

Bases: *sbol3.om_unit.Unit*, `abc.ABC`

As adopted by SBOL, om:CompoundUnit is an abstract class that is extended by other classes to describe units of measure that can be represented as combinations of multiple other units of measure.

class UnitDivision(*identity: str, symbol: str, label: str, numerator: Union[sbol3.om_unit.Unit, str], denominator: Union[sbol3.om_unit.Unit, str], *, alternative_symbols: List[str] = None, alternative_labels: List[str] = None, comment: str = None, long_comment: str = None, namespace: str = None, attachments: List[str] = None, name: str = None, description: str = None, derived_from: List[str] = None, generated_by: List[str] = None, measures: List[sbol3.SBOLObject] = None, type_uri: str = OM_UNIT_DIVISION*)

Bases: *CompoundUnit*

The purpose of the om:UnitDivision class is to describe a unit of measure that is the division of one unit of measure by another.

accept(*visitor: Any*) → Any

Invokes *visit_unit_division* on *visitor* with *self* as the only argument.

Parameters

visitor (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_unit_division method

Returns
Whatever *visitor.visit_unit_division* returns

Return type
Any

`class UnitExponentiation`(*identity: str*, *symbol: str*, *label: str*, *base: Union[*sbol3.om_unit.Unit*, str]*, *exponent: int*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = OM_UNIT_EXPONENTIATION*)

Bases: *`CompoundUnit`*

The purpose of the om:UnitExponentiation class is to describe a unit of measure that is raised to an integer power.

`accept`(*visitor: Any*) → Any

Invokes *visit_unit_exponentiation* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_unit_exponentiation method

Returns
Whatever *visitor.visit_unit_exponentiation* returns

Return type
Any

`class UnitMultiplication`(*identity: str*, *symbol: str*, *label: str*, *term1: Union[*sbol3.om_unit.Unit*, str]*, *term2: Union[*sbol3.om_unit.Unit*, str]*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = OM_UNIT_MULTIPLICATION*)

Bases: *`CompoundUnit`*

The purpose of the om:UnitMultiplication class is to describe a unit of measure that is the multiplication of two other units of measure.

`accept`(*visitor: Any*) → Any

Invokes *visit_unit_multiplication* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_unit_multiplication method

Returns
Whatever *visitor.visit_unit_multiplication* returns

Return type
Any

build_unit_division(*identity: str*, ***, *type_uri: str = OM_UNIT_DIVISION*) → sbol3.SBOLObject

build_unit_exponentiation(*identity: str*, ***, *type_uri: str = OM_UNIT_EXPONENTIATION*) → sbol3.SBOLObject

build_unit_multiplication(*identity: str*, ***, *type_uri: str = OM_UNIT_MULTIPLICATION*) → sbol3.SBOLObject

`sbol3.om_prefix`

Module Contents

Classes

BinaryPrefix	The purpose of the om:BinaryPrefix class is to describe standard
Prefix	As adopted by SBOL, om:Prefix is an abstract class that is extended
SIPrefix	The purpose of the om:SIPrefix class is to describe standard SI

Functions

build_binary_prefix(→ SBOLObject)	
build_si_prefix(→ SBOLObject)	

class BinaryPrefix(*identity: str*, *symbol: str*, *label: str*, *factor: float*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = OM_BINARY_PREFIX*)

Bases: *Prefix*

The purpose of the om:BinaryPrefix class is to describe standard binary prefixes such as “kibi,” “mebi,” “gibi,” etc. These prefixes commonly precede units of information such as “bit” and “byte.”

accept(*visitor: Any*) → Any

Invokes *visit_binary_prefix* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_binary_prefix method

Returns
Whatever *visitor.visit_binary_prefix* returns

Return type
Any

class Prefix(*identity: str*, *symbol: str*, *label: str*, *factor: float*, *type_uri: str*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*)

Bases: `CustomTopLevel`, `abc.ABC`

As adopted by SBOL, om:Prefix is an abstract class that is extended by other classes to describe factors that are commonly represented by standard unit prefixes. For example, the factor 10**3 is represented by the standard unit prefix "milli".

See Appendix A Section A.2 of the SBOL 3.0 specificiation.

validate(*report: ValidationReport = None*) → *ValidationReport*

class SIPrefix(*identity: str*, *symbol: str*, *label: str*, *factor: float*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = OM_SI_PREFIX*)

Bases: *Prefix*

The purpose of the om:SIPrefix class is to describe standard SI prefixes such as "milli," "centi," "kilo," etc.

accept(*visitor: Any*) → Any

Invokes *visit_si_prefix* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_si_prefix method

Returns
Whatever *visitor.visit_si_prefix* returns

Return type
Any

build_binary_prefix(*identity: str*, ***, *type_uri: str = OM_BINARY_PREFIX*) → *SBOLObject*

build_si_prefix(*identity: str*, ***, *type_uri: str = OM_SI_PREFIX*) → *SBOLObject*

sbol3.om_unit

Module Contents

Classes

Measure	The purpose of the om:Measure class is to link a numerical value to
PrefixedUnit	The purpose of the om:PrefixedUnit class is to describe a unit of
SingularUnit	The purpose of the om:SingularUnit class is to describe a unit of
Unit	As adopted by SBOL, om:Unit is an abstract class that is extended

Functions

`build_measure`(→ SBOLObject)
`build_prefixed_unit`(→ SBOLObject)
`build_singular_unit`(→ SBOLObject)

class Measure(*value: float*, *unit: str*, ***, *types: Optional[str, list[str]] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *identity: str = None*, *type_uri: str = OM_MEASURE*)

Bases: `CustomIdentified`

The purpose of the om:Measure class is to link a numerical value to a om:Unit.

accept(*visitor: Any*) → Any

Invokes *visit_measure* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_measure method

Returns
Whatever *visitor.visit_measure* returns

Return type
Any

class PrefixedUnit(*identity: str*, *symbol: str*, *label: str*, *unit: Union[Unit, str]*, *prefix: Union[sbol3.om_prefix.Prefix, str]*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = OM_PREFIXED_UNIT*)

Bases: `Unit`

The purpose of the om:PrefixedUnit class is to describe a unit of measure that is the multiplication of another unit of measure and a factor represented by a standard prefix such as “milli,” “centi,” “kilo,” etc.

accept(*visitor: Any*) → Any

Invokes *visit_prefixed_unit* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_prefixed_unit method

Returns
Whatever *visitor.visit_prefixed_unit* returns

Return type
Any

class SingularUnit(*identity: str*, *symbol: str*, *label: str*, ***, *unit: str = None*, *factor: float = None*, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *type_uri: str = OM_SINGULAR_UNIT*)

Bases: `Unit`

The purpose of the om:SingularUnit class is to describe a unit of measure that is not explicitly represented as a combination of multiple units, but could be equivalent to such a representation. For example, a joule is considered to be a om:SingularUnit, but it is equivalent to the multiplication of a newton and a meter.

accept(*visitor: Any*) → Any

Invokes *visit_singular_unit* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_singular_unit method

Returns
Whatever *visitor.visit_singular_unit* returns

Return type
Any

class Unit(*identity: str*, *symbol: str*, *label: str*, *type_uri: str*, ***, *alternative_symbols: List[str] = None*, *alternative_labels: List[str] = None*, *comment: str = None*, *long_comment: str = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*)

Bases: `CustomTopLevel`, `abc.ABC`

As adopted by SBOL, om:Unit is an abstract class that is extended by other classes to describe units of measure using a shared set of properties.

See Appendix A Section A.2 of the SBOL 3.0 specificiation.

build_measure(*identity: str*, ***, *type_uri: str = OM_MEASURE*) → *SBOLObject*

build_prefixed_unit(*identity: str*, ***, *type_uri: str = OM_PREFIXED_UNIT*) → *SBOLObject*

build_singular_unit(*identity: str*, ***, *type_uri: str = OM_SINGULAR_UNIT*) → *SBOLObject*

`sbol3.ownedobject`

Module Contents

Classes

OwnedObjectListProperty
OwnedObjectPropertyMixin
OwnedObjectSingletonProperty

Functions

`OwnedObject`(→ Property)	

`OwnedObject`(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[Identified, List[Identified]]] = None*, *type_constraint: Optional[Any] = None*) → *Property*

class `OwnedObjectListProperty`(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[str] = None*, *type_constraint: Optional[Any] = None*)

Bases: `OwnedObjectPropertyMixin`, `ListProperty`

`validate`(*name: str*, *report: ValidationReport*)

class `OwnedObjectPropertyMixin`(***kwargs*)

`from_user`(*value: Any*) → rdflib.Literal

`item_added`(*item: Any*) → None

`to_user`(*value: Any*) → str

`validate_type_constraint`(*name: str*, *report: ValidationReport*)

class `OwnedObjectSingletonProperty`(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[str] = None*, *type_constraint: Optional[Any] = None*)

Bases: `OwnedObjectPropertyMixin`, `SingletonProperty`

`validate`(*name: str*, *report: ValidationReport*)

`sbol3.participation`

Module Contents

Classes

`Participation`	Each Participation represents how a particular Feature behaves in

Functions

build_participation(→ SBOLObject)	

class Participation(*roles: Union[str, list[str]]*, *participant: Union[SBOLObject, str]*, ***, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *identity: str = None*, *type_uri: str = SBOL_PARTCIPATION*)

Bases: `Identified`

Each Participation represents how a particular Feature behaves in its parent Interaction.

accept(*visitor: Any*) → Any

Invokes *visit_participation* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_participation method

Returns
Whatever *visitor.visit_participation* returns

Return type
Any

validate(*report: ValidationReport = None*) → *ValidationReport*

build_participation(*identity: str*, ***, *type_uri: str = SBOL_PARTCIPATION*) → *SBOLObject*

sbol3.property_base

Module Contents

Classes

ListProperty	Helper class that provides a standard way to create an ABC using
Property	Helper class that provides a standard way to create an ABC using
SingletonProperty	Helper class that provides a standard way to create an ABC using

class ListProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*)

Bases: *Property*, `collections.abc.MutableSequence`, `abc.ABC`

Helper class that provides a standard way to create an ABC using inheritance.

__contains__(*item*) → bool

`__delitem__`(*key: Union[int, slice]*) → None

`__eq__`(*other*) → bool

Return self==value.

`__getitem__`(*key: Union[int, slice]*) → Any

`__len__`() → int

`__repr__`() → str

Return repr(self).

`__setitem__`(*key: Union[int, slice]*, *value: Any*) → None

`__str__`() → str

Return str(self).

`insert`(*index: int*, *value: Any*) → None

S.insert(index, value) – insert value before index

`set`(*value: Any*) → None

`class` **`Property`**(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*)

Bases: `abc.ABC`

Helper class that provides a standard way to create an ABC using inheritance.

`property` **`attribute_name:  Union[str, None]`**

Heuristically determine which attribute is associated with this property. If no attribute can be found for this property return None.

`abstract` **`from_user`**(*value: Any*)

`item_added`(*item: Any*) → None

Stub method for child classes to override if they have to do any additional processing on items after they are added. This method will be called on each individual item that was added to the list.

`abstract` **`set`**(*value: Any*) → None

`abstract` **`to_user`**(*value: Any*) → str

`validate`(*name: str*, *report: sbol3.ValidationReport*)

`class` **`SingletonProperty`**(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*)

Bases: `Property`, `abc.ABC`

Helper class that provides a standard way to create an ABC using inheritance.

`get`() → Any

`set`(*value: Any*) → None

`sbol3.provenance`

Module Contents

Classes

Activity	A generated prov:Entity is linked through a prov:wasGeneratedBy
Agent	Examples of agents are a person, organization, or software
Association	An prov:Association is linked to an prov:Agent through the
Plan	The prov:Plan entity can be used as a place holder to describe the
Usage	How different entities are used in an prov:Activity is specified

Functions

build_association(→ sbol3.SBOLObject)
build_usage(→ sbol3.SBOLObject)

class Activity(*identity: str*, ***, *types: Optional[str, list[str]] = None*, *start_time: Union[str, datetime.datetime] = None*, *end_time: Union[str, datetime.datetime] = None*, *usage: List[sbol3.Identified] = None*, *association: List[sbol3.Identified] = None*, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = PROV_ACTIVITY*)

Bases: `sbol3.CustomTopLevel`

A generated prov:Entity is linked through a prov:wasGeneratedBy relationship to an prov:Activity, which is used to describe how different prov:Agents and other entities were used. An prov:Activity is linked through a prov:qualifiedAssociation to prov:Associations, to describe the role of agents, and is linked through prov:qualifiedUsage to prov:Usages to describe the role of other entities used as part of the activity. Moreover, each prov:Activity includes optional prov:startedAtTime and prov:endedAtTime properties. When using prov:Activity to capture how an entity was derived, it is expected that any additional information needed will be attached as annotations. This may include software settings or textual notes. Activities can also be linked together using the prov:wasInformedBy relationship to provide dependency without explicitly specifying start and end times.

accept(*visitor: Any*) → Any

Invokes *visit_activity* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_activity method

Returns
Whatever *visitor.visit_activity* returns

Return type
Any

class Agent(*identity: str*, ***, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = PROV_AGENT*)

Bases: `sbol3.CustomTopLevel`

Examples of agents are a person, organization, or software tool. These agents should be annotated with additional information, such as software version, needed to be able to run the same prov:Activity again.

accept(*visitor: Any*) → Any

Invokes *visit_agent* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_agent method

Returns
Whatever *visitor.visit_agent* returns

Return type
Any

class Association(*agent: Union[str, sbol3.Identified]*, ***, *roles: Optional[str, list[str]] = None*, *plan: Union[sbol3.Identified, str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *identity: str = None*, *type_uri: str = PROV_ASSOCIATION*)

Bases: `sbol3.CustomIdentified`

An prov:Association is linked to an prov:Agent through the prov:agent relationship. The prov:Association includes the prov:hadRole property to qualify the role of the prov:Agent in the prov:Activity.

accept(*visitor: Any*) → Any

Invokes *visit_association* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_association method

Returns
Whatever *visitor.visit_association* returns

Return type
Any

class Plan(*identity: str*, ***, *namespace: str = None*, *attachments: List[str] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *type_uri: str = PROV_PLAN*)

Bases: `sbol3.CustomTopLevel`

The prov:Plan entity can be used as a place holder to describe the steps (for example scripts or lab protocols) taken when an prov:Agent is used in a particular prov:Activity.

`accept`(*visitor: Any*) → Any

Invokes *visit_plan* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_plan method

Returns
Whatever *visitor.visit_plan* returns

Return type
Any

`class Usage`(*entity: str*, ***, *roles: Optional[str, list[str]] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *identity: str = None*, *type_uri: str = PROV_USAGE*)

Bases: `sbol3.CustomIdentified`

How different entities are used in an prov:Activity is specified with the prov:Usage class, which is linked from an prov:Activity through the prov:Usage relationship. A prov:Usage is then linked to an prov:Entity through the prov:entity property URI and the prov:hadRole property species how the prov:Entity is used. When the prov:wasDerivedFrom property is used together with the full provenance described here, the entity pointed at by the prov:wasDerivedFrom property MUST be included in a prov:Usage.

`accept`(*visitor: Any*) → Any

Invokes *visit_usage* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_usage method

Returns
Whatever *visitor.visit_usage* returns

Return type
Any

`build_association`(*identity: str*, ***, *type_uri: str = PROV_USAGE*) → sbol3.SBOLObject

`build_usage`(*identity: str*, ***, *type_uri: str = PROV_USAGE*) → sbol3.SBOLObject

`sbol3.refobj_property`

Module Contents

Classes

ReferencedObjectList	
ReferencedObjectMixin	
ReferencedObjectSingleton	
ReferencedURI	str(object='') -> str

Functions

ReferencedObject(→ Union[ReferencedURI, ...)	

ReferencedObject(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[Union[Identified, str], list[Union[Identified, str]]]] = None*) → Union[*ReferencedURI*, list[*ReferencedURI*], *Property*]

class ReferencedObjectList(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[list[Union[Identified, str]]] = None*)

Bases: *ReferencedObjectMixin*, `ListProperty`

class ReferencedObjectMixin

static from_user(*value: Any*) → rdflib.URIRef

maybe_add_to_document(*value: Any*) → None

to_user(*value: Any*) → str

class ReferencedObjectSingleton(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[Identified, str]] = None*)

Bases: *ReferencedObjectMixin*, `SingletonProperty`

set(*value: Any*) → None

class ReferencedURI

Bases: `str`

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

Initialize self. See help(type(self)) for accurate signature.

__eq__(*other*)

Return self==value.

`lookup()`

`sbol3.seqfeat`

Module Contents

Classes

SequenceFeature	

Functions

build_sequence_feature(→ SBOLObject)	

class SequenceFeature(*locations: Union[*sbol3.location.Location*, List[*sbol3.location.Location*]]*, *, *roles: List[str] = None*, *orientation: str = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[*SBOLObject*] = None*, *identity: str = None*, *type_uri: str = SBOL_SEQUENCE_FEATURE*)

Bases: *sbol3.feature.Feature*

accept(*visitor: Any*) → Any

Invokes *visit_sequence_feature* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_sequence_feature method

Returns
Whatever *visitor.visit_sequence_feature* returns

Return type
Any

build_sequence_feature(*identity: str*, *, *type_uri: str = SBOL_SEQUENCE_FEATURE*) → *SBOLObject*

`sbol3.sequence`

Module Contents

Classes

Sequence	

`class Sequence`(*identity: str*, ***, *elements: Optional[str] = None*, *encoding: Optional[str] = None*, *namespace: Optional[str] = None*, *attachments: Optional[refobj_list_arg] = None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *derived_from: Optional[Union[str, Sequence[str]]] = None*, *generated_by: Optional[refobj_list_arg] = None*, *measures: Optional[ownedobj_list_arg] = None*, *type_uri: str = SBOL_SEQUENCE*)

Bases: `TopLevel`

`accept`(*visitor: Any*) → Any

Invokes *visit_sequence* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_sequence method

Returns
Whatever *visitor.visit_sequence* returns

Return type
Any

`validate`(*report: ValidationReport = None*) → *ValidationReport*

`sbol3.subcomponent`

Module Contents

Classes

SubComponent	The SubComponent class can be used to specify structural

Functions

build_subcomponent(→ sbol3.Identified)	Used by Document to construct a SubComponent when reading an SBOL file.

`class SubComponent`(*instance_of: Union[sbol3.Identified, str]*, ***, *role_integration: Optional[str] = None*, *locations: Union[sbol3.Location, List[sbol3.Location]] = None*, *source_locations: Union[sbol3.Location, List[sbol3.Location]] = None*, *roles: List[str] = None*, *orientation: str = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[sbol3.SBOLObject] = None*, *identity: str = None*, *type_uri: str = SBOL_SUBCOMPONENT*)

Bases: *sbol3.feature.Feature*

The SubComponent class can be used to specify structural hierarchy. For example, the Component of a gene might contain four SubComponent objects: a promoter, RBS, CDS, and terminator, each linked to a Component that provides the complete definition. In turn, the Component of the promoter SubComponent might itself contain SubComponent objects defining various operator sites, etc.

Parameters

- **identity** – this object's Uniform Resource Identifier (URI). this URI MUST be globally unique among all other Identified object URIs. See SBOL 3.0.1 specification section 5.1. This can also be a *displayId*, which will be concatenated to a default namespace automatically.
- **type_uri** – the concrete type of this object, specified as a URI. These are typically in the SBOL3 namespace, like *http://sbols.org/v3#Sequence* or *http://sbols.org/v3#Component*. This can also be the type URI of an extension class.
- **name** – A human-readable name for this object, for display purposes.
- **description** – Per the SBOL 3.0.1 specification, "a more thorough text description" of this object.
- **derived_from** – The URIs of one or more SBOL or non-SBOL objects from which this object was derived. This property is defined by the PROV-O ontology.
- **generated_by** – The URIs of one or more prov:Activity objects that describe how this object was generated. This property is defined by the PROV-O ontology.
- **measures** – The URIs of one or more om:Measure objects, each of which refers to a om:Measure object that describes measured parameters for this object. om:Measure objects are defined by the OM ontology

accept(*visitor: Any*) → Any

Invokes *visit_sub_component* on *visitor* with *self* as the only argument.

Parameters
visitor (*Any*) – The visitor instance

Raises
AttributeError – If visitor lacks a visit_sub_component method

Returns
Whatever *visitor.visit_sub_component* returns

Return type
Any

build_subcomponent(*identity: str*, *type_uri: str = SBOL_SUBCOMPONENT*) → sbol3.Identified

Used by Document to construct a SubComponent when reading an SBOL file.

sbol3.text_property

Module Contents

Classes

TextListProperty
TextPropertyMixin
TextSingletonProperty

Functions

TextProperty(→ Union[str, list[str], Property])	

class TextListProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[str] = None*)

Bases: *TextPropertyMixin*, `ListProperty`

TextProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[str, List[str]]] = None*) → Union[str, list[str], *Property*]

class TextPropertyMixin

from_user(*value: Any*) → Union[None, rdflib.Literal]

to_user(*value: Any*) → str

class TextSingletonProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[str] = None*)

Bases: *TextPropertyMixin*, `SingletonProperty`

`sbol3.toplevel`

Module Contents

Classes

TopLevel	TopLevel is an abstract class that is extended by any Identified

Functions

make_erase_identity_traverser(→ Callable[[Identified], ...)
make_update_references_traverser(...)

class TopLevel(*identity: str*, *type_uri: str*, ***, *namespace: Optional[str] = None*, *attachments: Optional[refobj_list_arg] = None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *derived_from: Optional[Union[str, Sequence[str]]] = None*, *generated_by: Optional[refobj_list_arg] = None*, *measures: Optional[ownedobj_list_arg] = None*)

Bases: `Identified`

TopLevel is an abstract class that is extended by any Identified class that can be found at the top level of an SBOL document or file. In other words, TopLevel objects are not nested inside any other object via composite aggregation. Instead of nesting, composite TopLevel objects refer to subordinate TopLevel objects by their URIs using shared aggregation.

clone(*new_identity: str = None*) → *TopLevel*

copy(*target_doc=None*, *target_namespace=None*)

static default_namespace(*namespace: Union[None, str]*, *identity: str*) → str

remove_from_document()

set_identity(*new_identity: str*) → Any

Sets the identity of the object.

USE WITH EXTREME CAUTION!

This method can break a tree of objects, and invalid a document. Only use this method if you understand the ramifications of changing the identity of a top level object.

Parameters
new_identity – the new identity

Returns
Nothing

split_identity() → tuple[str, str, str]

Split this object's identity into three components: namespace, path, and display_id.

Returns
A tuple of namespace, path, and display_id

update_all_dependents(*identity_map: dict[str,* Identified*]*) → Any

Update all dependent objects based on the provided identity map.

Dependent objects are reparented and references are updated according to the identities and objects in *identity_map*.

USE WITH CAUTION!

Parameters
identity_map – map of identity to Identified

Returns
Nothing

validate(*report:* ValidationReport *= None*) → *ValidationReport*

validate_identity(*report:* ValidationReport) → None

make_erase_identity_traverser(*identity_map: Dict[str,* Identified*]*) → Callable[[*Identified*], None]

make_update_references_traverser(*identity_map: Dict[str,* Identified*]*) → Callable[[*Identified*], None]

sbol3.typing

Module Contents

ownedobj = 'Identified'

ownedobj_list

ownedobj_list_arg

`refobj`

`refobj_list`

`refobj_list_arg`

`uri_list`

`uri_singleton`

sbol3.uri_property

Module Contents

Classes

URIListProperty	Helper class that provides a standard way to create an ABC using
URIPropertyMixin	
URISingletonProperty	Helper class that provides a standard way to create an ABC using

Functions

URIProperty(→ Union[str, list[str], sbol3.Property])	

class URIListProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[str, list[str]]] = None*)

Bases: *URIPropertyMixin*, `sbol3.ListProperty`

Helper class that provides a standard way to create an ABC using inheritance.

URIProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: Union[int, float]*, ***, *validation_rules: Optional[List] = None*, *initial_value: Optional[Union[str, List[str]]] = None*) → Union[str, list[str], sbol3.Property]

class URIPropertyMixin

static from_user(*value: Any*) → rdflib.URIRef

static to_user(*value: Any*) → str

class URISingletonProperty(*property_owner: Any*, *property_uri: str*, *lower_bound: int*, *upper_bound: int*, *validation_rules: Optional[List] = None*, *initial_value: Optional[str] = None*)

Bases: *URIPropertyMixin*, `sbol3.SingletonProperty`

Helper class that provides a standard way to create an ABC using inheritance.

`sbol3.util`

Module Contents

Functions

string_to_display_id(→ str)	Convert a string to a valid display id.

string_to_display_id(*name: str*) → str

Convert a string to a valid display id.

The SBOL specification has rules about what constitutes a valid display id. Make an effort here to convert any string into a valid display id.

`sbol3.utils`

Module Contents

Functions

parse_class_name(uri)	Parse the class name in a URI.

parse_class_name(*uri: str*)

Parse the class name in a URI.

The input is expected to be of the form 'http://sbols.org/v3#Component' or 'http://example.com/ApplicationSpecificClass'. This function would return 'Component' and 'ApplicationSpecificClass' respectively.

`sbol3.validation`

Module Contents

Classes

ValidationError	A ValidationError is a violation of the SBOL specification.
ValidationIssue	Base class for ValidationError and ValidationWarning.
ValidationReport	
ValidationWarning	A ValidationWarning is a violation of an SBOL best practice.

class ValidationError(*object_id*, *rule_id*, *message*)

Bases: *ValidationIssue*

A ValidationError is a violation of the SBOL specification.

class ValidationIssue(*object_id*, *rule_id*, *message*)

Base class for ValidationError and ValidationWarning.

__str__()

Return str(self).

class ValidationReport

property errors: *Sequence*[*ValidationError*]

property warnings: *Sequence*[*ValidationWarning*]

__iter__()

__len__()

__str__()

Return str(self).

addError(*object_id*, *rule_id*, *message*)

addWarning(*object_id*, *rule_id*, *message*)

class ValidationWarning(*object_id*, *rule_id*, *message*)

Bases: `ValidationIssue`

A ValidationWarning is a violation of an SBOL best practice.

`sbol3.varcomp`

Module Contents

Classes

VariableFeature	As described above, the VariableComponent Variable-Feature class

Functions

build_variable_feature(identity[, type_uri])	Used by Document to construct a VariableFeature when reading an SBOL file.

class VariableFeature(*cardinality: str*, *variable: Union[Identified, str]*, ***, *variants: List[Union[Identified, str]] = None*, *variant_collections: Union[Union[Identified, str], List[Union[Identified, str]]] = None*, *variant_derivations: List[Union[Identified, str]] = None*, *variant_measures: List[Union[Identified, str]] = None*, *name: str = None*, *description: str = None*, *derived_from: List[str] = None*, *generated_by: List[str] = None*, *measures: List[SBOLObject] = None*, *identity: str = None*, *type_uri: str = SBOL_VARIABLE_FEATURE*)

Bases: `Identified`

As described above, the VariableComponent VariableFeature class specifies a variable and set of values that will replace one of the SubComponent Feature objects in the template of a CombinatorialDerivation. The variable is specified by the variable property, and the set of values is defined by the union of Component objects referred to by the variant, variantCollection, and variantDerivation properties.

`accept`(*visitor: Any*) → Any

Invokes *visit_variable_feature* on *visitor* with *self* as the only argument.

Parameters
`visitor` (*Any*) – The visitor instance

Raises
`AttributeError` – If visitor lacks a visit_variable_feature method

Returns
Whatever *visitor.visit_variable_feature* returns

Return type
Any

`validate`(*report:* ValidationReport *= None*) → *ValidationReport*

`build_variable_feature`(*identity: str*, *type_uri: str = SBOL_VARIABLE_FEATURE*)

Used by Document to construct a VariableFeature when reading an SBOL file.

8.1.2 Package Contents

`__version__ = '1.1'`

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX