Owlready2 Documentation

Release 0.44

Jean-Baptiste LAMY

Aug 02, 2023

Contents

1 Table of content **3**

1.1 Introduction . . . 3
1.2 Owlready2 Installation . . . 5
1.3 Managing ontologies . . . 6
1.4 Classes and Individuals (Instances) . . . 11
1.5 Properties . . . 15
1.6 Datatypes . . . 20
1.7 Class constructs, restrictions and logical operators . . . 21
1.8 Disjointness, open and local closed world reasoning . . . 25
1.9 Mixing Python and OWL . . . 27
1.10 Reasoning . . . 29
1.11 Annotations . . . 32
1.12 Namespaces . . . 36
1.13 SPARQL queries . . . 38
1.14 Worlds . . . 45
1.15 Parallelism, multiprocessing and synchronization . . . 47
1.16 SWRL rules . . . 51
1.17 General class axioms . . . 52
1.18 PyMedTermino2 . . . 53
1.19 Observation framework . . . 61
1.20 Development . . . 62
1.21 Differences between Owlready version 1 and 2 . . . 63
1.22 Contact and links . . . 64

Owlready2 is a package for ontology-oriented programming in Python. It can load OWL 2.0 ontologies as Python objects, modify them, save them, and perform reasoning via HermiT (included). Owlready2 allows a transparent access to OWL ontologies (contrary to usual Java-based API).

Owlready version 2 includes an optimized triplestore / quadstore, based on SQLite3. This quadstore is optimized both for performance and memory consumption. Contrary to version 1, Owlready2 can deal with big ontologies. Owlready2 can also access to UMLS and medical terminology (using the integrated PyMedTermino2 submodule).

Owlready2 has been created at the LIMICS reseach lab, University Paris 13, Sorbonne Paris Cité, INSERM UMRS 1142, Paris 6 University, by Jean-Baptiste Lamy. It was initially developed during the VIIIP research project funded by ANSM, the French Drug Agency; this is why some examples in this documentation relate to drug ;).

Owlready2 is available under the GNU LGPL licence v3. If you use Owlready2 in scientific works, **please cite the following article**:

> **Lamy JB**. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. **Artificial Intelligence In Medicine 2017**;80:11-28

In case of troubles, questions or comments, please use this Forum/Mailing list: http://owlready.306.s1.nabble.com

CHAPTER 1

Table of content

1.1 Introduction

Owlready2 is a package for manipulating OWL 2.0 ontologies in Python. It can load, modify, save ontologies, and it supports reasoning via HermiT (included). Owlready allows a transparent access to OWL ontologies.

Owlready2 can:

- Import ontologies in RDF/XML, OWL/XML or NTriples format.
- Manipulates ontology classes, instances and annotations as if they were Python objects.
- Add Python methods to ontology classes.
- Re-classify instances automatically, using the HermiT reasoner.
- Import medical terminologies from UMLS (see *PyMedTermino2*).

If you need to "convert" formulas between Protégé, Owlready2 and/or Description Logics, the following cheat sheet may be of interest:

The great table of Description Logics and formal ontology notations

1.1.1 Short example: What can I do with Owlready?

Load an ontology from a local repository, or from Internet:

```
>>> from owlready2 import *
>>> onto_path.append("/path/to/your/local/ontology/repository")
>>> onto = get_ontology("http://www.lesfleursdunormal.fr/static/_downloads/pizza_onto.
↪owl")
>>> onto.load()
```

Create new classes in the ontology, possibly mixing OWL constructs and Python methods:

```
>>> class NonVegetarianPizza(onto.Pizza):
...   equivalent_to = [
...     onto.Pizza
...   & ( onto.has_topping.some(onto.MeatTopping)
...     | onto.has_topping.some(onto.FishTopping)
...     ) ]

...   def eat(self): print("Beurk! I'm vegetarian!")
```

Access the classes of the ontology, and create new instances / individuals:

```
>>> onto.Pizza
pizza_onto.Pizza

>>> test_pizza = onto.Pizza("test_pizza_owl_identifier")
>>> test_pizza.has_topping = [ onto.CheeseTopping(),
...                             onto.TomatoTopping() ]
```

In Owlready2, almost any lists can be modified *in place*, for example by appending/removing items from lists. Owlready2 automatically updates the RDF quadstore.

```
>>> test_pizza.has_topping.append(onto.MeatTopping())
```

Perform reasoning, and classify instances and classes:

```
>>> test_pizza.__class__
onto.Pizza

>>> # Execute HermiT and reparent instances and classes
>>> sync_reasoner()

>>> test_pizza.__class__
onto.NonVegetarianPizza
>>> test_pizza.eat()
Beurk! I'm vegetarian !
```

Export to OWL file:

```
>>> onto.save()
```

Load Gene Ontology (GO), a large ontology (~ 170 Mb, can take a moment!):

```
>>> go = get_ontology("http://purl.obolibrary.org/obo/go.owl").load()
```

Access entities with an IRI that does not start with the ontology's IRI, by creating a Namespace:

```
>>> obo = get_namespace("http://purl.obolibrary.org/obo/")

>>> print(obo.GO_0000001.label)
['mitochondrion inheritance']
```

1.1.2 Architecture

Owlready2 maintains a RDF quadstore in an optimized database (SQLite3), either in memory or on the disk (see *Worlds*). It provides a high-level access to the Classes and the objects in the ontology (aka. ontology-oriented programming). Classes and Invididuals are loaded dynamically from the quadstore as needed, cached in memory and destroyed when no longer needed.

1.2 Owlready2 Installation

Owlready2 can be installed with 'pip', the Python Package Installer.

Owlready2 include an optimized Cython module. This module speeds up by about 20% the loading of large ontologies, but its use is entirely optional. To build this module, you need a C compiler, and to install the 'cython' Python package.

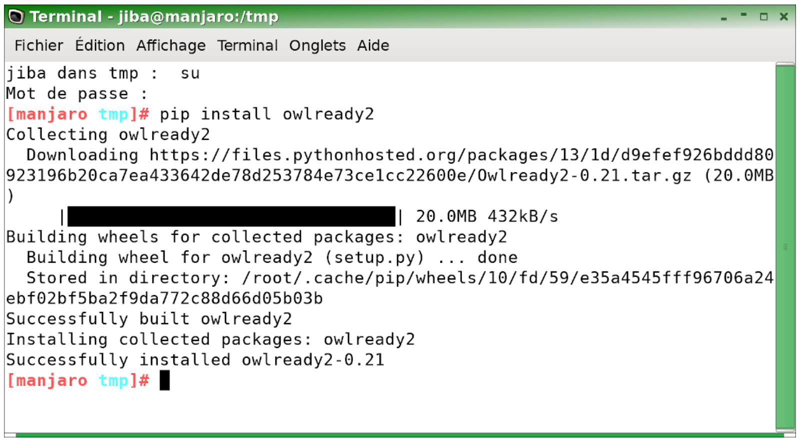
On the contrary, if you don't have a C compiler, to **not build** the optimized module you need to uninstall Cython if it is already installed (or to use the manual installation described below).

Owlready2 can be installed from terminal, from Python, or manually.

1.2.1 Installation from terminal (Bash under Linux or DOS under Windows)

You can use the following Bash / DOS commands to install Owlready2 in a terminal:

```
pip install owlready2
```


Terminal - jiba@manjaro:/tmp
Fichier Édition Affichage Terminal Onglets Aide

```
jiba dans tmp :  su
Mot de passe :
[manjaro tmp]# pip install owlready2
Collecting owlready2
  Downloading https://files.pythonhosted.org/packages/13/1d/d9efef926bddd80
923196b20ca7ea433642de78d253784e73ce1cc22600e/Owlready2-0.21.tar.gz (20.0MB
)
     |████████████████████████████████| 20.0MB 432kB/s
Building wheels for collected packages: owlready2
  Building wheel for owlready2 (setup.py) ... done
  Stored in directory: /root/.cache/pip/wheels/10/fd/59/e35a4545fff96706a24
ebf02bf5ba2f9da772c88d66d05b03b
Successfully built owlready2
Installing collected packages: owlready2
Successfully installed owlready2-0.21
[manjaro tmp]#
```

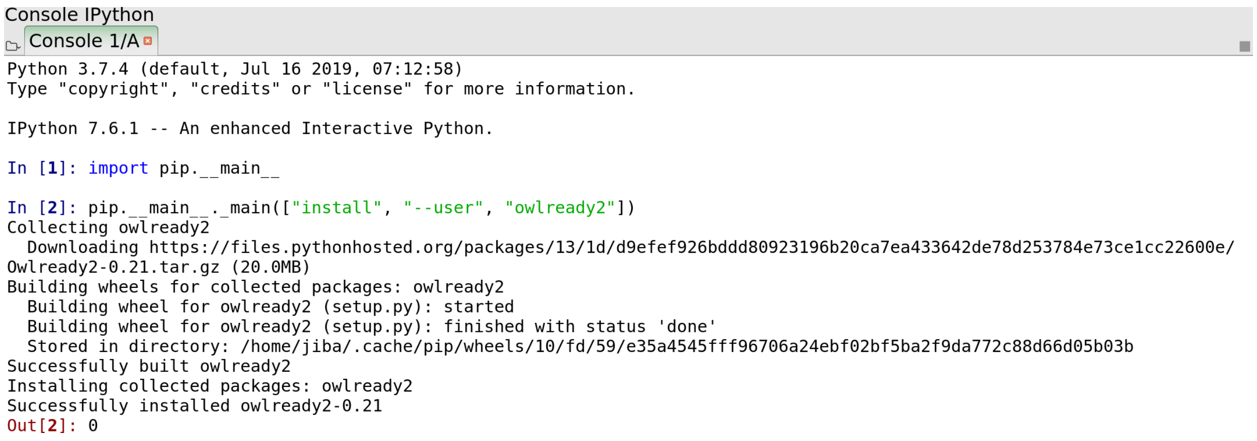
If you don't have the permissions for writing in system files, you can install Owlready2 in your user directory with this command:

```
pip install --user owlready2
```

1.2.2 Installation in Spyder / IDLE (or any other Python console)

You can use the following Python commands to install Owlready2 from a Python 3.7.x console (including those found in Spyder3 or IDLE):

```
>>> import pip.__main__
>>> pip.__main__._main(["install", "--user", "owlready2"])
```



```
Python 3.7.4 (default, Jul 16 2019, 07:12:58)
Type "copyright", "credits" or "license" for more information.

IPython 7.6.1 -- An enhanced Interactive Python.

In [1]: import pip.__main__

In [2]: pip.__main__._main(["install", "--user", "owlready2"])
Collecting owlready2
  Downloading https://files.pythonhosted.org/packages/13/1d/d9efef926bddd80923196b20ca7ea433642de78d253784e73ce1cc22600e/
Owlready2-0.21.tar.gz (20.0MB)
Building wheels for collected packages: owlready2
  Building wheel for owlready2 (setup.py): started
  Building wheel for owlready2 (setup.py): finished with status 'done'
  Stored in directory: /home/jiba/.cache/pip/wheels/10/fd/59/e35a4545fff96706a24ebf02bf5ba2f9da772c88d66d05b03b
Successfully built owlready2
Installing collected packages: owlready2
Successfully installed owlready2-0.21
Out[2]: 0
```

1.2.3 Manual installation

Owlready2 can also be installed manually in 3 steps:

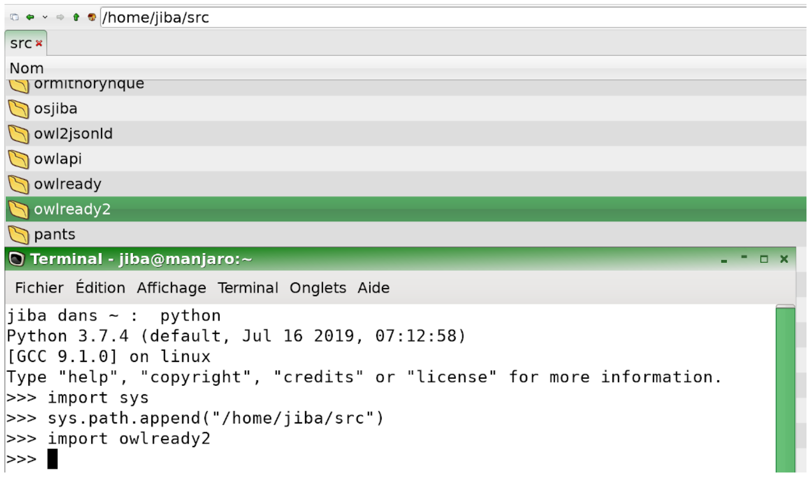
# Uncompress the Owlready2-0.21.tar.gz source release file (or any other version), for example in C:\ under Windows

# Rename the directory C:\Owlready2-0.21 as C:\owlready2

# Add the C:\ directory in your PYTHONPATH; this can be done in Python as follows:

```
import sys
sys.path.append("C:\")
import owlready2
```

In the following screenshot, I used /home/jiba/src instead of C:\, under Linux:



```
jiba dans ~ :  python
Python 3.7.4 (default, Jul 16 2019, 07:12:58)
[GCC 9.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path.append("/home/jiba/src")
>>> import owlready2
>>> 
```

1.3 Managing ontologies

1.3.1 Creating an ontology

A new empty ontology can be obtained with the get_ontology() function; it takes a single parameter, the IRI of the ontology. The IRI is a sort of URL; IRIs are used as identifier for ontologies.

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")
```

Note: If an ontology has already been created for the same IRI, it will be returned.

Note: Some ontologies use a # character in IRI to separate the name of the ontology from the name of the entities, while some others uses a /. By default, Owlready2 uses a #, if you want to use a /, the IRI should ends with /.

Examples:

```
>>> onto = get_ontology("http://test.org/onto.owl") # => http://test.org/onto.owl
↪#entity

>>> onto = get_ontology("http://test.org/onto") # => http://test.org/onto#entity

>>> onto = get_ontology("http://test.org/onto/") # => http://test.org/onto/entity
```

1.3.2 Loading an ontology from OWL files

Use the .load() method of an ontology for loading it.

The easiest way to load the ontology is to load a local copy. In this case, the IRI is the local filename prefixed with "file://", for example:

```
>>> onto = get_ontology("file:///home/jiba/onto/pizza_onto.owl").load()
```

If an URL is given, Owlready2 first searches for a local copy of the OWL file and, if not found, tries to download it from the Internet. For example:

```
>>> onto_path.append("/path/to/owlready/onto/")

>>> onto = get_ontology("http://www.lesfleursdunormal.fr/static/_downloads/pizza_onto.
↪owl").load()
```

The onto_path global variable contains a list of directories for searching local copies of ontologies. It behaves similarly to sys.path (for Python modules / packages).

The get_ontology() function returns an ontology from its IRI, and creates a new empty ontology if needed.

The .load() method loads the ontology from a local copy or from Internet. It is **safe** to call .load() several times on the same ontology. It will be loaded only once.

Note: Owlready2 currently reads the following file format: RDF/XML, OWL/XML, NTriples. The file format is automatically detected.

NTriples is a very simple format and is natively supported by Owlready2.

RDF/XML is the most common format; it is also natively supported by Owlready2 (since version 0.2).

OWL/XML is supported using a specific parser integrated to Owlready2. This parser supports a large subset of OWL, but is not complete. It has been tested mostly with OWL files created with the Protégé editor or with Owlready itself.

Consequently, preferred formats are RDF/XML and NTriples.

1.3.3 Accessing the content of an ontology

You can access to the content of an ontology using the 'dot' notation, as usual in Python or more generally in Object-Oriented Programming. In this way, you can access the Classes, Instances, Properties, Annotation Properties,... defined in the ontology. The [] syntax is also accepted.

```
>>> print(onto.Pizza)
onto.Pizza

>>> print(onto["Pizza"])
onto.Pizza
```

An ontology has the following attributes:

- .base_iri : base IRI for the ontology
- .imported_ontologies : the list of imported ontologies (see below)

and the following methods:

- .classes() : returns a generator for the Classes defined in the ontology (see *Classes and Individuals (Instances)*)
- .individuals() : returns a generator for the individuals (or instances) defined in the ontology (see *Classes and Individuals (Instances)*)
- .object_properties() : returns a generator for ObjectProperties defined in the ontology (see *Properties*)
- .data_properties() : returns a generator for DataProperties defined in the ontology (see *Properties*)
- .annotation_properties() : returns a generator for AnnotationProperties defined in the ontology (see *Annotations*)
- .properties() : returns a generator for all Properties (object-, data- and annotation-) defined in the ontology
- .disjoint_classes() : returns a generator for AllDisjoint constructs for Classes defined in the ontology (see *Disjointness, open and local closed world reasoning*)
- .disjoint_properties() : returns a generator for AllDisjoint constructs for Properties defined in the ontology (see *Disjointness, open and local closed world reasoning*)
- .disjoints() : returns a generator for AllDisjoint constructs (for Classes and Properties) defined in the ontology
- .different_individuals() : returns a generator for AllDifferent constructs for individuals defined in the ontology (see *Disjointness, open and local closed world reasoning*)
- .get_namepace(base_iri) : returns a namespace for the ontology and the given base IRI (see namespaces below, in the next section)

Note: Many methods returns a generator. Generators allows iterating over the values without creating a list, which can improve performande. However, they are often not very convenient when exploring the ontology:

```
>>> onto.classes()
<generator object _GraphManager.classes at 0x7f854a677728>
```

A generator can be trandformed into a list with the list() Python function:

```
>>> list(onto.classes())
[pizza_onto.CheeseTopping, pizza_onto.FishTopping, pizza_onto.MeatTopping,
pizza_onto.Pizza, pizza_onto.TomatoTopping, pizza_onto.Topping,
pizza_onto.NonVegetarianPizza]
```

The IRIS pseudo-dictionary can be used for accessing an entity from its full IRI:

```
>>> IRIS["http://www.lesfleursdunormal.fr/static/_downloads/pizza_onto.owl#Pizza"]
pizza_onto.Pizza
```

Ontologies can also define entities located in other namespaces, for example Gene Ontology (GO) has the following IRI: 'http://purl.obolibrary.org/obo/go.owl', but the IRI of GO entities are of the form 'http://purl.obolibrary.org/obo/GO_entity' (note the missing 'go.owl#'). See *Namespaces* to learn how to access such entities.

1.3.4 Simple queries

Simple queries can be performed with the .search() method of the ontology. It expects one or several keyword arguments. The supported keywords are:

- **iri**, for searching entities by its full IRI
- **type**, for searching Individuals of a given Class
- **subclass_of**, for searching subclasses of a given Class
- **is_a**, for searching both Individuals and subclasses of a given Class
- **subproperty_of**, for searching subproperty of a given Property
- any object, data or annotation property name

Special arguments are:

- **_use_str_as_loc_str**: whether to treats plain Python strings as strings in any language (default is True)
- **_case_sensitive**: whether to take lower/upper case into consideration (default is True)
- **_bm25**: if True, returns a list of (entity, relevance) pairs instead of just the entities (default is False)

The value associated to each keyword can be a single value or a list of several values. A star * can be used as a jocker in string values.

Warning: .search() does not perform any kind of reasoning, it just searches in asserted facts. In addition, it cannot find Classes through SOME or ONLY restrictions.

For example, for searching for all entities with an IRI ending with 'Topping':

```
>>> onto.search(iri = "*Topping")
[pizza_onto.CheeseTopping, pizza_onto.FishTopping, pizza_onto.MeatTopping,
pizza_onto.TomatoTopping, pizza_onto.Topping]
```

In addition, the special value "*" can be used as a wildcard for any object. For example, the following line searches for all individuals that are related to another one with the 'has_topping' relation (NB there is none in the default pizza_onto.owl file):

```
>>> onto.search(has_topping = "*")
```

When a single return value is expected, the .search_one() method can be used. It works similarly:

```
>>> onto.search_one(label = "my label")
```

Owlready classes and individuals can be used as values within search(), as follows:

```
>>> onto.search_one(is_a = onto.Pizza)
```

Finally, search() can be nested, as in the following example:

```
>>> onto.search(is_a = onto.Pizza, has_topping = onto.search(is_a = onto.
↪TomatoTopping))
```

Owlready automatically combines nested searches in a single, optimized, search.

For more complex queries, SQPARQL can be used with RDFlib (see *Worlds*).

1.3.5 Ontology metadata

The metadata of the ontology can be accessed with .metadata, in read and write access:

```
>>> print(onto.metadata.comment)
[...]
>>> onto.metadata.comment.append("my first comment")
```

Any annotation can be used with .metadata.

In addition, you can list the available annotions by iterating through .metadata, for example:

```
>>> for annot_prop in onto.metadata:
...         print(annot_prop, ":", annot_prop[onto.metadata])
```

1.3.6 Importing other ontologies

An ontology can import other ontologies, like a Python module can import other modules.

The imported_ontologies attribute of an ontology is a list of the ontology it imports. You can add or remove items in that list:

```
>>> onto.imported_ontologies.append(owlready_ontology)
```

1.3.7 Saving an ontology to an OWL file

The .save() method of an ontology can be used to save it. It will be saved in the first directory in onto_path.

```
>>> onto.save()
>>> onto.save(file = "filename or fileobj", format = "rdfxml")
```

.save() accepts two optional parameters: 'file', a file object or a filename for saving the ontology, and 'format', the file format (default is RDF/XML).

Note: Owlready2 currently writes the following file format: "rdf/xml", "ntriples".

NTriples is a very simple format and is natively supported by Owlready2.

RDF/XML is the most common format; it is also natively supported by Owlready2 (since version 0.2).

OWL/XML is not yet supported for writing.

1.3.8 Changing the ontology base IRI

You can change the ontology base IRI as follows:

```
>>> onto.base_iri = "http://test.org/new_base_iri.owl#"
```

This will also change the IRI or all entities in the ontology. If you don't want to change IRI entities, you can use the set_base_iri() method:

```
>>> onto.set_base_iri("http://test.org/new_base_iri.owl#", rename_entities = False)
```

1.3.9 Destroying an ontology

You can destroy an ontology as follows:

```
>>> onto.destroy()
```

By default, Owlready does not update the Python object in memory. This may cause problem in some situations, e.g. if you continue using a class from another ontology that inherits from a class in the destroyed ontology.

You can destroy the ontology and update the Python objet as follows:

```
>>> onto.destroy(update_relation = True, update_is_a = True)
```

The update_is_a optional argument updates is-a relation (subClassOf, subPropertyOf and RDF type), while update_relation updates other relations.

1.4 Classes and Individuals (Instances)

1.4.1 Creating a Class

A new Class can be created in an ontology by inheriting the owlready2.Thing class.

The ontology class attribute can be used to associate your class to the given ontology. If not specified, this attribute is inherited from the parent class (in the example below, the parent class is Thing, which is defined in the 'owl' ontology).

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")

>>> class Drug(Thing):
...     namespace = onto
```

The namespace Class attribute is used to build the full IRI of the Class, and can be an ontology or a namespace (see *Namespaces*). The 'with' statement can also be used to provide the ontology (or namespace):

```
>>> onto = get_ontology("http://test.org/onto.owl")

>>> with onto:
...     class Drug(Thing):
...         pass
```

The .iri attribute of the Class can be used to obtain the full IRI of the class.

```
>>> print(Drug.iri)
http://test.org/onto.owl#Drug
```

.name and .iri attributes are writable and can be modified (this allows to change the IRI of an entity, which is sometimes called "refactoring").

1.4.2 Creating and managing subclasses

Subclasses can be created by inheriting an ontology class. Multiple inheritance is supported.

```
>>> class DrugAssociation(Drug): # A drug associating several active principles
...     pass
```

Owlready2 provides the .is_a attribute for getting the list of superclasses (__bases__ can be used, but with some limits described in *Class constructs, restrictions and logical operators*). It can also be modified for adding or removing superclasses.

```
>>> print(DrugAssociation.is_a)
[onto.Drug]
```

The .subclasses() method returns the list of direct subclasses of a class.

```
>>> print(Drug.subclasses())
[onto.DrugAssociation]
```

The .descendants() and .ancestors() Class methods return a set of the descendant and ancestor Classes (including self, but excluding non-entity classes such as restrictions).

```
>>> DrugAssociation.ancestors()
{onto.DrugAssociation, owl.Thing, onto.Drug}
```

1.4.3 Creating classes dynamically

The 'types' Python module can be used to create classes and subclasses dynamically:

```
>>> import types

>>> with my_ontology:
...     NewClass = types.new_class("NewClassName", (SuperClass,))
```

1.4.4 Creating equivalent classes

The .equivalent_to Class attribute is a list of equivalent classes. It behaves like .is_a (programmatically).

To obtain all equivalent classes, including indirect ones (due to transitivity), use .INDIRECT_equivalent_to.

1.4.5 Creating Individuals

Individuals are instances in ontologies. They are created as any other Python instances. The first parameter is the name (or identifier) of the Individual; it corresponds to the .name attribute in Owlready2. If not given, the name if automatically generated from the Class name and a number.

```
>>> my_drug = Drug("my_drug")
>>> print(my_drug.name)
my_drug
>>> print(my_drug.iri)
http://test.org/onto.owl#my_drug

>>> unamed_drug = Drug()
>>> print(unamed_drug.name)
drug_1
```

Additional keyword parameters can be given when creating an Individual, and they will be associated to the corresponding Properties (for more information on Properties, see *Properties*).

```
my_drug = Drug("my_drug2", namespace = onto, has_for_active_principle = [],...)
```

The Instances are immediately available in the ontology:

```
>>> print(onto.drug_1)
onto.drug_1
```

The .instances() class method can be used to iterate through all Instances of a Class (including its subclasses). It returns a generator.

```
>>> for i in Drug.instances(): print(i)
```

Multiple calls with the individual name and namespace will returns the same individual (without creating a dupplicate), and update the individual if property values are given.

```
>>> assert Drug("my_drug3") is Drug("my_drug3")
```

Finally, Individuals also have the .equivalent_to attribute (which correspond to the "same as" relation).

1.4.6 Querying Individual relations

For a given Individual, the values of a property can be obtained with the usual "object.property" dot notation. See *Properties* for more details.

```
>>> print(onto.my_drug.has_for_active_principle)
```

Property name can be prefixed with "**INDIRECT_**" to obtain all indirect relations (i.e. those asserted at the class level with restriction, implied by transistive properties, subproperties, equivalences, etc):

```
>>> print(onto.my_drug.INDIRECT_has_for_active_principle)
```

1.4.7 Introspecting Individuals

The list of properties that exist for a given individual can be obtained by the .get_properties() method. It returns a generator that yields the properties (without dupplicates).

```
>>> onto.drug_1.get_properties()
```

The following example shows how to list the properties of a given individual, and the associated values:

```
>>> for prop in onto.drug_1.get_properties():
>>>     for value in prop[onto.drug_1]:
>>>         print(".%s == %s" % (prop.python_name, value))
```

Notice the "Property[individual]" syntax. It allows to get the values as a list, even for functional properties (contrary to getattr(individual, Property.python_name).

Inverse properties can be obtained by the .get_inverse_properties() method. It returns a generator that yields (subject, property) tuples.

```
>>> onto.drug_1.get_inverse_properties()
```

1.4.8 Mutli-Class Individuals

In ontologies, an Individual can belong to more than one Class. This is supported in Owlready2.

Individuals have a .is_a atribute that behaves similarly to Class .is_a, but with the Classes of the Individual. In order to create a mutli-Class Individual, you need to create the Individual as a single-Class Instance first, and then to add the other Class(ses) in its .is_a attribute:

```
>>> class BloodBasedProduct(Thing):
...     ontology = onto

>>> a_blood_based_drug = Drug()
>>> a_blood_based_drug.is_a.append(BloodBasedProduct)
```

Owlready2 will automatically create a hidden Class that inherits from both Drug and BloodBasedProduct. This hidden class is visible in a_blood_based_drug.__class__, but not in a_blood_based_drug.is_a.

1.4.9 Equivalent (identical, SameAs) individuals

The .equivalent_to Individual attribute is a list of equivalent individuals (corresponding to OWL SameAs relation). This list can be modified.

To obtain all equivalent individuals, including indirect ones (due to transitivity), use .INDIRECT_equivalent_to.

1.4.10 Destroying entities

The destroy_entity() global function can be used to destroy an entity, i.e. to remove it from the ontology and the quad store. Owlready2 behaves similarly to Protege4 when destroying entities: all relations involving the destroyed entity are destroyed too, as well as all class constructs and blank nodes that refer it.

```
>>> destroy_entity(individual)
>>> destroy_entity(Klass)
>>> destroy_entity(Property)
```

1.5 Properties

1.5.1 Creating a new class of property

A new property can be created by sublcassing the ObjectProperty or DataProperty class. The 'domain' and 'range' properties can be used to specify the domain and the range of the property. Domain and range must be given in list, since OWL allows to specify several domains or ranges for a given property (if multiple domains or ranges are specified, the domain or range is the intersection of them, *i.e.* the items in the list are combined with an AND logical operator).

The following example creates two Classes, Drug and Ingredient, and then an ObjectProperty that relates them.

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")

>>> with onto:
...     class Drug(Thing):
...         pass
...     class Ingredient(Thing):
...         pass
...     class has_for_ingredient(ObjectProperty):
...         domain    = [Drug]
...         range     = [Ingredient]
```

In addition, the 'domain >> range' syntax can be used when creating property. It replaces the ObjectProperty or DataProperty parent Class, as follows:

```
>>> with onto:
...     class has_for_ingredient(Drug >> Ingredient):
...         pass
```

In addition, the following subclasses of Property are available: FunctionalProperty, InverseFunctionalProperty, TransitiveProperty, SymmetricProperty, AsymmetricProperty, ReflexiveProperty, IrreflexiveProperty. They should be used in addition to ObjectProperty or DataProperty (or the 'domain >> range' syntax).

1.5.2 Getting domain and range

The .domain and .range attributes of a Property can be used to query its domain and range. They returns a list.

```
>>> has_for_ingredient.domain
[Drug]

>>> has_for_ingredient.range
[Ingredient]
```

1.5.3 Creating a relation

A relation is a triple (subject, property, object) where property is a Property class, and subject and object are instances (or literal, such as string or numbers) which are subclasses of the domain and range defined for the property class. A relation can be get or set using Python attribute of the subject, the attribute name being the same as the Property class name:

```
>>> my_drug = Drug("my_drug")

>>> acetaminophen = Ingredient("acetaminophen")

>>> my_drug.has_for_ingredient = [acetaminophen]
```

The attribute contains a list of the subjects:

```
>>> print(my_drug.has_for_ingredient)
[onto.acetaminophen]
```

This list can be modifed *in place* or set to a new value; Owlready2 will automatically add or delete RDF triples in the quadstore as needed:

```
>>> codeine = Ingredient("codeine")

>>> my_drug.has_for_ingredient.append(codeine)

>>> print(my_drug.has_for_ingredient)
[onto.acetaminophen, onto.codeine]
```

1.5.4 Data Property

Data Properties are Properties with a data type in their range. The following data types are currently supported by Owlready2:

- int
- float
- bool
- str (string)
- owlready2.normstr (normalized string, a single-line string)
- owlready2.locstr (localized string, a string with a language associated)
- datetime.date
- datetime.time
- datetime.datetime

Here is an example of a string Data Property:

```
>>> with onto:
...     class has_for_synonym(DataProperty):
...         range = [str]

>>> acetaminophen.has_for_synonym = ["acetaminophen", "paracétamol"]
```

The 'domain >> range' syntax can also be used:

```
>>> with onto:
...     class has_for_synonym(Thing >> str):
...         pass
```

1.5.5 Inverse Properties

Two properties are inverse if they express the same meaning, but in a reversed way. For example the 'is_ingredient_of' Property is the inverse of the 'has_for_ingredient' Property created above: saying "a drug A has for ingredient B" is equivalent to "B is ingredient of drug A".

In Owlready2, inverse Properties are defined using the 'inverse_property' attribute.

```
>>> with onto:
...     class is_ingredient_of(ObjectProperty):
...         domain           = [Ingredient]
...         range            = [Drug]
...         inverse_property = has_for_ingredient
```

Owlready automatically handles Inverse Properties. It will automatically set has_for_ingredient.inverse_property, and automatically update relations when the inverse relation is modified.

```
>>> my_drug2 = Drug("my_drug2")

>>> aspirin = Ingredient("aspirin")

>>> my_drug2.has_for_ingredient.append(aspirin)

>>> print(my_drug2.has_for_ingredient)
[onto.aspirin]

>>> print(aspirin.is_ingredient_of)
[onto.my_drug2]


>>> aspirin.is_ingredient_of = []

>>> print(my_drug2.has_for_ingredient)
[]
```

Note: This won't work for the drug created previously, because we created the inverse property **after** we created the relation between my_drug and acetaminophen.

1.5.6 Functional and Inverse Functional properties

A functional property is a property that has a single value for a given instance. Functional properties are created by inheriting the FunctionalProperty class.

```
>>> with onto:
...     class has_for_cost(DataProperty, FunctionalProperty): # Each drug has a␣
↪single cost
...         domain    = [Drug]
...         range     = [float]

>>> my_drug.has_for_cost = 4.2

>>> print(my_drug.has_for_cost)
4.2
```

Contrary to other properties, a functional property returns a single value instead of a list of values. If no value is defined, they returns None.

```
>>> print(my_drug2.has_for_cost)
None
```

Owlready2 is also able to guess when a Property is functional with respect to a given class. In the following example, the 'prop' Property is not functional, but Owlready2 guesses that, for Individuals of Class B, there can be only a single value. Consequently, Owlready2 considers prop as functional for Class B.

```
>>> with onto:
...     class prop(ObjectProperty): pass
...     class A(Thing): pass
...     class B(Thing):
...         is_a = [ prop.max(1) ]

>>> A().prop
[]
>>> B().prop
None
```

An Inverse Functional Property is a property whose inverse property is functional. They are created by inheriting the InverseFunctionalProperty class.

1.5.7 Creating a subproperty

A subproperty can be created by subclassing a Property class.

```
>>> with onto:
...     class ActivePrinciple(Ingredient):
...         pass
...     class has_for_active_principle(has_for_ingredient):
...         domain    = [Drug]
...         range     = [ActivePrinciple]
```

Note: Owlready2 currently does not automatically update parent properties when a child property is defined. This might be added in a future version, though.

1.5.8 Obtaining indirect relations (considering subproperty, transitivity, etc)

Property name can be prefixed by "**INDIRECT_**" to obtain all indirectly related entities. It takes into account:

- transitive, symmetric and reflexive properties,
- property inheritance (i.e. subproperties),
- classes of an individual (i.e. values asserted at the class level),
- class inheritance (i.e. parent classes).
- equivalences (i.e. equivalent classes, identical "same-as" individuals,. . .)

```
>>> with onto:
...     class BodyPart(Thing): pass
```

(continues on next page)

(continued from previous page)

```
...     class part_of(BodyPart >> BodyPart, TransitiveProperty): pass
...     abdomen          = BodyPart("abdomen")
...     heart            = BodyPart("heart"            , part_of = [abdomen])
...     left_ventricular = BodyPart("left_ventricular", part_of = [heart])
...     kidney           = BodyPart("kidney"           , part_of = [abdomen])

... print(left_ventricular.part_of)
[heart]

... print(left_ventricular.INDIRECT_part_of)
[heart, abdomen]
```

1.5.9 Associating Python alias name to Properties

In ontologies, properties are usually given long names, *e.g.* "has_for_ingredient", while in programming languages like Python, shorter attribute names are more common, *e.g.* "ingredients" (notice also the use of a plural form, since it is actually a list of several ingredients).

Owlready2 allows to rename Properties with more Pythonic name through the 'python_name' annotation (defined in the Owlready ontology, file 'owlready2/owlready_ontology.owl' in Owlready2 sources, URI http://www.lesfleursdunormal.fr/static/_downloads/owlready_ontology.owl):

```
>>> has_for_ingredient.python_name = "ingredients"

>>> my_drug3 = Drug()

>>> cetirizin = Ingredient("cetirizin")

>>> my_drug3.ingredients = [cetirizin]
```

Note: The Property class is still considered to be called 'has_for_ingredient', for example it is still available as 'onto.has_for_ingredient', but not as 'onto.ingredients'.

For more information about the use of annotations, see *Annotations*.

The 'python_name' annotations can also be defined in ontology editors like Protégé, by importing the Owlready ontology (file 'owlready2/owlready_ontology.owl' in Owlready2 sources, URI http://www.lesfleursdunormal.fr/static/_downloads/owlready_ontology.owl).

1.5.10 Getting relation instances

The list of relations that exist for a given property can be obtained by the .get_relations() method. It returns a generator that yields (subject, object) tuples.

```
>>> onto.has_for_active_principle.get_relations()
```

Warning: The quadstore is not indexed for the .get_relations() method. Thus, it can be slow on huge ontologies.

1.6 Datatypes

Owlready automatically recognizes and translates basic datatypes to Python, such as string, int, float, etc.

1.6.1 Creating custom datatypes

The declare_datatype() global function allows to declare a new datatype. It takes 4 arguments:

- datatype: the Python datatype (for example, a Python type or class)
- iri: the IRI used to represent the datatype in ontologies
- parser: a function that takes a serialized string and returns the corresponding datatype
- unparser: a function that takes a datatype and returns its serialization in a string

The function returns the storid associated to the datatype.

Warning: The datatype must be declared **BEFORE** loading any ontology that uses it.

Here is an example for adding support for the XSD "hexBinary" datatype:

```
>>> class Hex(object):
...   def __init__(self, value):
...     self.value = value

>>> def parser(s):
...   return Hex(int(s, 16))

>>> def unparser(x):
...   h = hex(x.value)[2:]
...   if len(h) % 2 != 0: return "0%s" % h
...   return h

>>> declare_datatype(Hex, "http://www.w3.org/2001/XMLSchema#hexBinary", parser,␣
↪unparser)
```

The new datatype can then be used as any others:

```
>>> onto = world.get_ontology("http://www.test.org/t.owl")

>>> with onto:
...   class p(Thing >> Hex): pass

...   class C(Thing): pass

...   c1 = C()
...   c1.p.append(Hex(14))
```

In addition, the define_datatype_in_ontology() function allows to define the datatype in a given ontology. This was not needed for hexBinary above, because it is already defined in XMLSchema. However, for user-defined datatype, it is recommended to define them in an ontology (Owlready does not strictly require that, but other tools like Protégé do).

The following example (re)define the hexBinary datatype in our ontology:

```
>>> define_datatype_in_ontology(Hex, "http://www.w3.org/2001/XMLSchema#hexBinary",␣
↪onto)
```

This add the (xsd:hexBinary, rdf:type, rdfs:datatype) RDF triple in the quadstore.

As said above, declare_datatype() must be called * before * using the datatype. On the contrary, define_datatype_in_ontology() may be called after loading an ontology that use the datatype.

1.7 Class constructs, restrictions and logical operators

Restrictions are special types of Classes in ontology.

1.7.1 Restrictions on a Property

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")

>>> with onto:
...     class Drug(Thing):
...         pass
...     class ActivePrinciple(Thing):
...         pass
...     class has_for_active_principle(Drug >> ActivePrinciple):
...         pass
```

For example, a non-Placebo Drug is a Drug with an Active Principle:

```
>>> class NonPlaceboDrug(Drug):
...     equivalent_to = [Drug & has_for_active_principle.some(ActivePrinciple)]
```

And a Placebo is a Drug with no Active Principle:

```
>>> class Placebo(Drug):
...     equivalent_to = [Drug & Not(has_for_active_principle.some(ActivePrinciple))]
```

In the example above, 'has_for_active_principle.some(ActivePrinciple)' is the Class of all objects that have at least one Active Principle. The Not() function returns the negation (or complement) of a Class. The & operator returns the intersection of two Classes.

Another example, an Association Drug is a Drug that associates two or more Active Principle:

```
>>> with onto:
...     class DrugAssociation(Drug):
...         equivalent_to = [Drug & has_for_active_principle.min(2, ActivePrinciple)]
```

Owlready provides the following types of restrictions (they have the same names than in Protégé):

- some : Property.some(Range_Class)
- only : Property.only(Range_Class)
- min : Property.min(cardinality, Range_Class)
- max : Property.max(cardinality, Range_Class)
- exactly : Property.exactly(cardinality, Range_Class)
- value : Property.value(Range_Individual / Literal value)
- has_self : Property.has_self(Boolean value)

When defining classes, restrictions can be used in class definition (i.e. 'equivalent_to ='), but also as superclasses, using 'is_a =', as in the following example:

```
>>> with onto:
...     class MyClass(Thing):
...         is_a = [my_property.some(Value)]
```

In addition, restrictions can be added to existing classes by adding them to .is_a or .equivalent_to, as in the two following examples:

```
>>> MyClass.is_a.append(my_property.some(Value))

>>> MyClass.equivalent_to.append(my_property.some(Value))
```

Restrictions can be modified *in place* (Owlready2 updates the quadstore automatically), using the following attributes: .property, .type (SOME, ONLY, MIN, MAX, EXACTLY or VALUE), .cardinality and .value (a Class, an Individual, a class contruct or another restriction).

Finally, the Inverse(Property) construct can be used as the inverse of a given Property.

1.7.2 Restrictions as class properties

Owlready allows to access restriction as class properties.

By default, existential restrictions (i.e. SOME restrictions and VALUES restrictions) can be accessed as if they were class properties in Owlready. For example:

```
>>> NonPlaceboDrug.has_for_active_principle
[onto.ActivePrinciple]
```

These class attributes can also be modified (e.g. NonPlaceboDrug.has_for_active_principle.append(. . .)).

The .class_property_type attribute of Properties allows to indicate how to handle class properties. It is a list made of the following values:

- "some": handle class properties as existential restrictions (i.e. SOME restrictions and VALUES restrictions).
- "only": handle class properties as universal restrictions (i.e. ONLY restrictions).
- "relation": handle class properties as relations (i.e. simple RDF triple, as in Linked Data).

When more than one value is specified, all the specified method are used when defining the value of the property for a class.

The .class_property_type attribute corresponds to the "http://www.lesfleursdunormal.fr/static/_downloads/owlready_ontology.owl#class_property_type" annotation.

The set_default_class_property_type(types) global function allows to set the default type of class property used, when no type is specified for a given property. The default value is ["some"].

1.7.3 Restrictions as class properties in defined classes

Defined classes are classes that are defined by an "equivalent to" relation, such as Placebo and NonPlaceboDrug above.

The .defined_class Boolean attribute can be used to mark a class as "defined". It corresponds to the "http://www.lesfleursdunormal.fr/static/_downloads/owlready_ontology.owl#defined_class" annotation.

When a class is marked as "defined", Owlready automatically generates an equivalent_to formula, taking into account the class parents and the class properties.

The following program shows an example. It creates a drug ontology, with a Drug class and several HealthConditions. In addition, two properties are created, for indiciations and contraindications. Here, we choose to manage indications with SOME restrictions and contraindication with ONLY restrictions.

Then, the program creates two subclasses of Drug: Antalgic and Aspirin. Thoses subclasses are marked as defined (with defined_class = True), and their properties are defined also.

```
>>> onto2 = get_ontology("http://test.org/onto2.owl")

>>> with onto2:
...     class Drug(Thing): pass
...     class ActivePrinciple(Thing): pass
...     class has_for_active_principle(Drug >> ActivePrinciple): pass

...     class HeathCondition(Thing): pass
...     class Pain(HeathCondition): pass
...     class ModeratePain(Pain): pass
...     class CardiacDisorder(HeathCondition): pass
...     class Hypertension(CardiacDisorder): pass

...     class Pregnancy(HeathCondition): pass
...     class Child(HeathCondition): pass
...     class Bleeding(HeathCondition): pass

...     class has_for_indications      (Drug >> HeathCondition): class_property_type␣
↪= ["some"]
...     class has_for_contraindications(Drug >> HeathCondition): class_property_type␣
↪= ["only"]

...     class Antalgic(Drug):
...         defined_class = True
...         has_for_indications = [Pain]
...         has_for_contraindications = [Pregnancy, Child, Bleeding]

...     class Aspirin(Antalgic):
...         defined_class = True
...         has_for_indications = [ModeratePain]
...         has_for_contraindications = [Pregnancy, Bleeding]
```

Owlready automatically produces the appropriate equivalent_to formula, as we can verify:

```
>>> print(Antalgic.equivalent_to)
[onto.Drug
& onto.has_for_indications.some(onto.Pain)
& onto.has_for_contraindications.only(onto.Child | onto.Pregnancy | onto.Bleeding)]

>>> print(Aspirin.equivalent_to)
[onto.Antalgic
& onto.has_for_indications.some(onto.ModeratePain)
& onto.has_for_contraindications.only(onto.Pregnancy | onto.Bleeding)]
```

Notice that this mapping between class properties and definition is bidirectional: one can also use it to access an existing definition as class properties. The following example illustrates that:

```
>>> with onto2:
...     class Antihypertensive(Drug):
...         equivalent_to = [Drug
...                          & has_for_indications.some(Hypertension)
```

(continues on next page)

(continued from previous page)

```
...                            &has_for_contraindications.only(Pregnancy)]

>>> print(Antihypertensive.has_for_indications)
[onto.Hypertension]

>>> print(Antihypertensive.has_for_contraindications)
[onto.Pregnancy]
```

1.7.4 Logical operators (intersection, union and complement)

Owlready provides the following operators between Classes (normal Classes but also class constructs and restrictions):

- '&' : And operator (intersection). For example: Class1 & Class2. It can also be written: And([Class1, Class2])
- '|' : Or operator (union). For example: Class1 | Class2. It can also be written: Or([Class1, Class2])
- Not() : Not operator (negation or complement). For example: Not(Class1)

The Classes used with logical operators can be normal Classes (inheriting from Thing), restrictions or other logical operators.

Intersections, unions and complements can be modified *in place* using the .Classes (intersections and unions) or .Class (complement) attributes.

1.7.5 One-Of constructs

In ontologies, a 'One Of' statement is used for defining a Class by extension, *i.e.* by listing its Instances rather than by defining its properties.

```
>>> with onto:
...     class DrugForm(Thing):
...         pass

>>> tablet     = DrugForm()
>>> capsule    = DrugForm()
>>> injectable = DrugForm()
>>> pomade     = DrugForm()

# Assert that there is only four possible drug forms
>>> DrugForm.is_a.append(OneOf([tablet, capsule, injectable, pomade]))
```

The construct be modified *in place* using the .instances attribute.

1.7.6 Inverse-of constructs

Inverse-of constructs produces the inverse of a property, without creating a new property.

```
Inverse(has_for_active_principle)
```

The construct be modified *in place* using the .property attribute.

1.7.7 ConstrainedDatatype

A constrained datatype is a data whose value is restricted, for example an integer between 0 and 20.

The global function ConstrainedDatatype() create a constrained datatype from a base datatype, and one or more facets:

- length
- min_length
- max_length
- pattern
- white_space
- max_inclusive
- max_exclusive
- min_inclusive
- min_exclusive
- total_digits
- fraction_digits

For example:

```
ConstrainedDatatype(int, min_inclusive = 0, max_inclusive = 20)
ConstrainedDatatype(str, max_length = 100)
```

1.7.8 Property chain

Property chain allows to chain two properties (this is sometimes noted prop1 o prop2). The PropertyChain() function allows to create a new property chain from a list of properties:

```
PropertyChain([prop1, prop2])
```

The construct be modified *in place* using the .properties attribute.

1.8 Disjointness, open and local closed world reasoning

By default, OWL considers the world as 'open', *i.e.* everything that is not stated in the ontology is not 'false' but 'possible' (this is known as *open world assumption*). Therfore, things and facts that are 'false' or 'impossible' must be clearly stated as so in the ontology.

1.8.1 Disjoint Classes

Two (or more) Classes are disjoint if there is no Individual belonging to all these Classes (remember that, contrary to Python instances, an Individual can have several Classes, see *Classes and Individuals (Instances)*).

A Classes disjointness is created with the AllDisjoint() function, which takes a list of Classes as parameter. In the example below, we have two Classes, Drug and ActivePrinciple, and we assert that they are disjoint (yes, we need to specify that explicitely – sometimes ontologies seem a little dumb!).

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")

>>> with onto:
...     class Drug(Thing):
...         pass
...     class ActivePrinciple(Thing):
...         pass
...     AllDisjoint([Drug, ActivePrinciple])
```

1.8.2 Disjoint Properties

OWL also introduces Disjoint Properties. Disjoint Properties can also be created using AllDisjoint().

1.8.3 Different Individuals

Two Individuals are different if they are distinct. In OWL, two Individuals might be considered as being actually the same, single, Individual, unless they are stated different. Difference is to Individuals what disjointness is to Classes.

The following example creates two active principles and asserts that they are different (yes, we also need to state explicitely that acetaminophen and aspirin are not the same!)

```
>>> acetaminophen = ActivePrinciple("acetaminophen")
>>> aspirin       = ActivePrinciple("aspirin")

>>> AllDifferent([acetaminophen, aspirin])
```

Note: In Owlready2, AllDifferent is actually the same function as AllDisjoint – the exact meaning depends on the parameters (AllDisjoint if you provide Classes, AllDifferent if you provide Instances, and disjoint Properties if you provide Properties).

1.8.4 Querying and modifying disjoints

The .disjoints() method returns a generator for iterating over AllDisjoint constructs involving the given Class or Property. For Individuals, .differents() behaves similarly.

```
>>> for d in Drug.disjoints():
...     print(d.entities)
[onto.Drug, onto.ActivePrinciple]
```

The 'entities' attribute of an AllDisjoint is writable, so you can modify the AllDisjoint construct by adding or removing entities.

OWL also provides the 'disjointWith' and 'propertyDisjointWith' relations for pairwise disjoints (involving only two elements). Owlready2 exposes **all** disjoints as AllDisjoints, *including* those declared with the 'disjointWith' or 'propertyDisjointWith' relations. In the quad store (or when saving OWL files), disjoints involving 2 entities are defined using the 'disjointWith' or 'propertyDisjointWith' relations, while others are defined using AllDisjoint or AllDifferent.

1.8.5 Closing Individuals

The open world assumption also implies that the properties of a given Individual are not limited to the ones that are explicitely stated. For example, if you create a Drug Individual with a single Active Principle, it does not mean that it has *only* a single Active Principle.

```
>>> with onto:
...     class has_for_active_principle(Drug >> ActivePrinciple): pass

>>> my_acetaminophen_drug = Drug(has_for_active_principle = [acetaminophen])
```

In the example above, 'my_acetaminophen_drug' has an acetaminophen Active Principle (this fact is true) and it may have other Active Principle(s) (this fact is possible).

If you want 'my_acetaminophen_drug' to be a Drug with acetaminophen and no other Active Principle, you have to state it explicitely using a restriction (see *Class constructs, restrictions and logical operators*):

```
>>> my_acetaminophen_drug.is_a.append(has_for_active_principle.
↪only(OneOf([acetaminophen])))
```

Notice that we used OneOf() to 'turn' the acetaminophen Individual into a Class that we can use in the restriction.

You'll quickly find that the open world assumption often leads to tedious and long lists of AllDifference and Restrictions. Hopefully, Owlready2 provides the close_world() function for automatically 'closing' an Individual. close_world() will automatically add ONLY restrictions as needed; it accepts an optional parameter: a list of the Properties to 'close' (defaults to all Properties whose domain is compatible with the Individual).

```
>>> close_world(my_acetaminophen_drug)
```

1.8.6 Closing Classes

close_world() also accepts a Class. In this case, it closes the Class, its subclasses, and all their Individuals.

By default, when close_world() is not called, the ontology performs **open world reasoning**. By selecting the Classes and the Individuals you want to 'close', the close_world() function enables **local closed world reasoning** with OWL.

1.8.7 Closing an ontology

Finally, close_world() also accepts an ontology. In this case, it closes all the Classes defined in the ontology. This corresponds to fully **closed world reasoning**.

1.9 Mixing Python and OWL

1.9.1 Adding Python methods to an OWL Class

Python methods can be defined in ontology Classes as usual in Python. In the example below, the Drug Class has a Python method for computing the per-tablet cost of a Drug, using two OWL Properties (which have been renamed in Python, see *Associating Python alias name to Properties*):

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")
```

(continues on next page)

(continued from previous page)

```
>>> with onto:
...     class Drug(Thing):
...         def get_per_tablet_cost(self):
...             return self.cost / self.number_of_tablets

...     class has_for_cost(Drug >> float, FunctionalProperty):
...         python_name = "cost"

...     class has_for_number_of_tablets(Drug >> int, FunctionalProperty):
...         python_name = "number_of_tablets"

>>> my_drug = Drug(cost = 10.0, number_of_tablets = 5)
>>> print(my_drug.get_per_tablet_cost())
2.0
```

1.9.2 Forward declarations

Sometimes, you may need to forward-declare a Class or a Property. If the same Class or Property (same name, same namespace) is redefined, the new definition **extends** the previous one (and do not replace it!).

For example:

```
>>> class has_for_active_principle(Property): pass

>>> with onto:
...     class Drug(Thing): pass

...     class has_for_active_principle(Drug >> ActivePrinciple): pass

...     class Drug(Thing): # Extends the previous definition of Drug
...         is_a = [has_for_active_principle.some(ActivePrinciple)]
```

(Notice that this definition of drug exclude Placebo).

1.9.3 Associating a Python module to an OWL ontology

It is possible to associate a Python module with an OWL ontology. When Owlready2 loads the ontology, it will automatically import the Python module. This is done with the 'python_module' annotation, which should be set on the ontology itself. The value should be the name of your Python module, *e.g.* 'my_package.my_module'. This annotation can be set with editor like Protégé, after importing the 'owlready_ontology.owl' ontology (file 'owlready2/owlready_ontology.owl' in Owlready2 sources, URI http://www.lesfleursdunormal.fr/static/_downloads/owlready_ontology.owl):

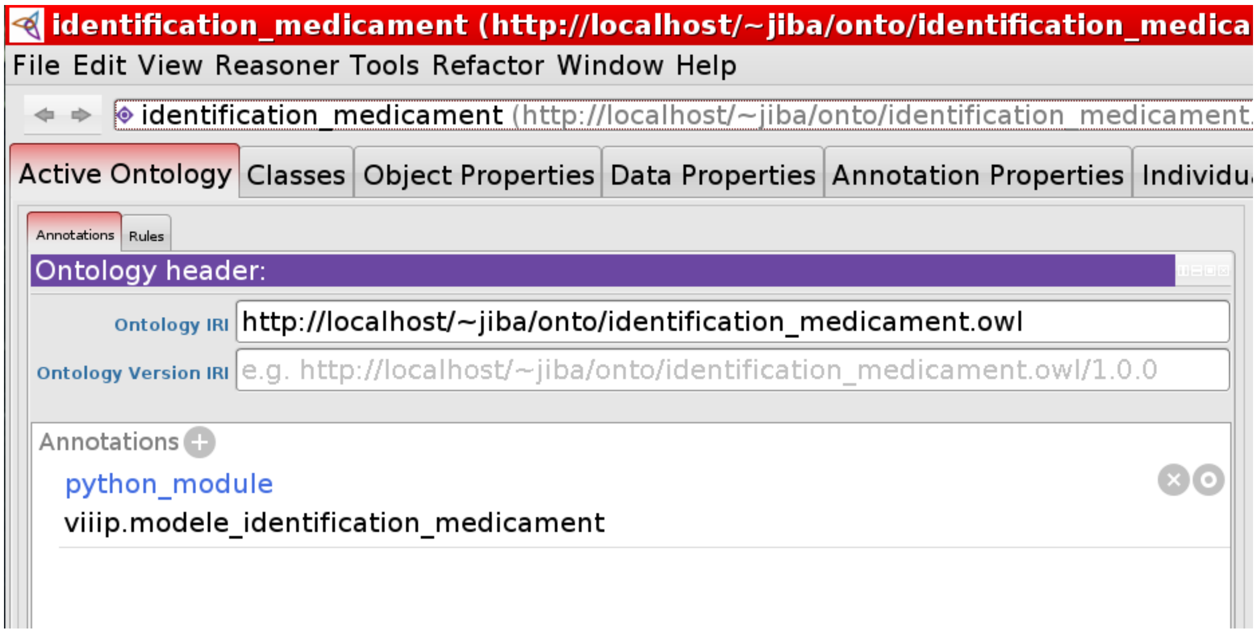
The Python module can countain Class and Properties definitions, and methods. However, it does not need to include all the is-a relations, domain, range,...: any such relation defined in OWL does not need to be specified again in Python. Therefore, if your Class hierarchy is defined in OWL, you can create all Classes as child of Thing.

For example, in file 'my_python_module.py':

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl") # Do not load the ontology here!
```

(continues on next page)


identification_medicament (http://localhost/~jiba/onto/identification_medica
File Edit View Reasoner Tools Refactor Window Help
identification_medicament (http://localhost/~jiba/onto/identification_medicament.
Active Ontology
Classes
Object Properties
Data Properties
Annotation Properties
Individu
Annotations
Rules
Ontology header:

Ontology IRI: http://localhost/~jiba/onto/identification_medicament.owl

Ontology Version IRI: e.g. http://localhost/~jiba/onto/identification_medicament.owl/1.0.0

Annotations

python_module
viiip.modele_identification_medicament

(continued from previous page)

```
>>> with onto:
...     class Drug(Thing):
...         def get_per_tablet_cost(self):
...             return self.cost / self.number_of_tablets
```

And then, in OWL file 'onto.owl', you can define:

- The 'python_module' annotation (value: 'my_python_module')
- The 'Drug' Class with superclasses if needed
- The 'has_for_cost' Property (ommitted in Python – not needed because it has no method)
- The 'has_for_number_of_tablets' Property (also ommitted)

In this way, Owlready2 allows you to take the best of Python and OWL!

1.10 Reasoning

OWL reasoners can be used to check the *consistency* of an ontology, and to deduce new fact in the ontology, typically be *reclassing* Individuals to new Classes, and Classes to new superclasses, depending on their relations.

Several OWL reasoners exist; Owlready2 includes:

- a modified version of the HermiT reasoner, developed by the department of Computer Science of the University of Oxford, and released under the LGPL licence.
- a modified version of the Pellet reasoner, released under the AGPL licence.

HermiT and Pellet are written in Java, and thus you need a Java Vitual Machine to perform reasoning in Owlready2.

HermiT is used by default.

1.10.1 Configuration

Under Linux, Owlready should automatically find Java.

Under windows, you may need to configure the location of the Java interpreter, as follows:

```
>>> from owlready2 import *
>>> import owlready2
>>> owlready2.JAVA_EXE = "C:\\path\\to\\java.exe"
```

1.10.2 Setting up everything

Before performing reasoning, you need to create all Classes, Properties and Instances, and to ensure that restrictions and disjointnesses / differences have been defined too.

Here is an example creating a 'reasoning-ready' ontology:

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")

>>> with onto:
...     class Drug(Thing):
...         def take(self): print("I took a drug")

...     class ActivePrinciple(Thing):
...         pass

...     class has_for_active_principle(Drug >> ActivePrinciple):
...         python_name = "active_principles"

...     class Placebo(Drug):
...         equivalent_to = [Drug & Not(has_for_active_principle.
↪some(ActivePrinciple))]
...         def take(self): print("I took a placebo")

...     class SingleActivePrincipleDrug(Drug):
...         equivalent_to = [Drug & has_for_active_principle.exactly(1,␣
↪ActivePrinciple)]
...         def take(self): print("I took a drug with a single active principle")

...     class DrugAssociation(Drug):
...         equivalent_to = [Drug & has_for_active_principle.min(2, ActivePrinciple)]
...         def take(self): print("I took a drug with %s active principles" %␣
↪len(self.active_principles))

>>> acetaminophen   = ActivePrinciple("acetaminophen")
>>> amoxicillin     = ActivePrinciple("amoxicillin")
>>> clavulanic_acid = ActivePrinciple("clavulanic_acid")

>>> AllDifferent([acetaminophen, amoxicillin, clavulanic_acid])

>>> drug1 = Drug(active_principles = [acetaminophen])
>>> drug2 = Drug(active_principles = [amoxicillin, clavulanic_acid])
>>> drug3 = Drug(active_principles = [])

>>> close_world(Drug)
```

1.10.3 Running the reasoner

The reasoner (HermiT) is simply run by calling the sync_reasoner() global function:

```
>>> sync_reasoner()
```

By default, sync_reasoner() places all inferred facts in a special ontology, 'http://inferrences/'. You can control in which ontology the inferred facts are placed using the 'with ontology' statement (remember, all triples asserted inside a 'with ontology' statement go inside this ontology). For example, for placing all inferred facts in the 'onto' ontology:

```
>>> with onto:
...     sync_reasoner()
```

This allows saving the ontology with the inferred facts (using onto.save() as usual).

The reasoner can also be limited to some ontologies:

```
>>> sync_reasoner([onto1, onto2,...])
```

If you also want to infer object property values, use the "infer_property_values" parameter:

```
>>> sync_reasoner(infer_property_values = True)
```

To use Pellet instead of HermiT, just use the sync_reasoner_pellet() function instead.

In addition, Pellet also supports the inference of data property values, using the "infer_data_property_values" parameter:

```
>>> sync_reasoner(infer_property_values = True, infer_data_property_values = True)
```

1.10.4 Results of the automatic classification

Owlready automatically gets the results of the reasoning from HermiT and reclassifies Individuals and Classes, *i.e* Owlready changes the Classes of Individuals and the superclasses of Classes.

```
>>> print("drug2 new Classes:", drug2.__class__)
drug2 new Classes: onto.DrugAssociation

>>> drug1.take()
I took a drug with a single active principle

>>> drug2.take()
I took a drug with 2 active principles

>>> drug3.take()
I took a placebo
```

In this example, drug1, drug2 and drug3 Classes have changed! The reasoner *deduced* that drug2 is an Association Drug, and that drug3 is a Placebo.

Also notice how the example combines automatic classification of OWL Classes with polymorphism on Python Classes.

1.10.5 Inconsistent classes and ontologies

In case of inconsistent ontology, an OwlReadyInconsistentOntologyError is raised.

Inconcistent classes may occur without making the entire ontology inconsistent, as long as these classes have no individuals. Inconsistent classes are inferred as equivalent to Nothing. They can be obtained as follows:

```
>>> list(default_world.inconsistent_classes())
```

In addition, the consistency of a given class can be tested by checking for Nothing in its equivalent classes, as follows:

```
>>> if Nothing in Drug.equivalent_to:
...       print("Drug is inconsistent!")
```

Note: To debug an inconsistent ontology the `explain` command of the Pellet reasoner can provide some useful information. The output of this command is shown if for `sync_reasoner_pellet(...)` the keyword argument `debug` has a value >=2 (default is 1). However, note that the additional call to `pellet explain` might take more time than the reasoning itself.

1.10.6 Querying inferred classification

The .get_parents_of(), .get_instances_of() and .get_children_of() methods of an ontology can be used to query the hierarchical relations, limited to those defined in the given ontology. This is commonly used after reasoning, to obtain the inferred hierarchical relations.

- .get_parents_of(entity) accepts any entity (Class, property or individual), and returns the superclasses (for a class), the superproperties (for a property), or the classes (for an individual). (NB for obtaining all parents, independently of the ontology they are asserted in, use entity.is_a).
- .get_instances_of(Class) returns the individuals that are asserted as belonging to the given Class in the ontology. (NB for obtaining all instances, independently of the ontology they are asserted in, use Class.instances()).
- .get_children_of(entity) returns the subclasses (or subproperties) that are asserted for the given Class or property in the ontology. (NB for obtaining all children, independently of the ontology they are asserted in, use entity.subclasses()).

Here is an example:

```
>>> inferences = get_ontology("http://test.org/onto_inferences.owl")
>>> with inferences:
...     sync_reasoner()

>>> inferences.get_parents_of(drug1)
[onto.SingleActivePrincipleDrug]

>>> drug1.is_a
[onto.has_for_active_principle.only(OneOf([onto.acetaminophen])), onto.
↪SingleActivePrincipleDrug]
```

1.11 Annotations

In Owlready2, annotations are accessed as attributes. For Classes, notice that annotations are **not** inherited.

1.11.1 Adding an annotation

For a given entity (a Class, a Property or an Individual), the following syntax can be used to add annotations:

```
>>> from owlready2 import *

>>> onto = get_ontology("http://test.org/onto.owl")

>>> with onto:
...     class Drug(Thing):
...         pass

>>> Drug.comment = ["A first comment on the Drug class", "A second comment"]

>>> Drug.comment.append("A third comment")
```

The following annotations are available by default: comment, isDefinedBy, label, seeAlso, backwardCompatibleWith, deprecated, incompatibleWith, priorVersion, versionInfo.

Owlready2 also supports annotations on relation triples, using the AnnotatedRelation class as folows:

```
>>> with onto:
...     class HealthProblem(Thing):
...         pass

...     class is_prescribed_for(Drug >> HealthProblem):
...         pass

>>> acetaminophen = Drug("acetaminophen")
>>> pain = HealthProblem("pain")
>>> acetaminophen.is_prescribed_for.append(pain)

>>> AnnotatedRelation(acetaminophen, is_prescribed_for, pain).comment = ["A comment␣
↪on the acetaminophen-pain relation"]
```

The AnnotatedRelation class constructor takes three parameters, corresponding to a subject-predicate-object triple. Then, you can use the dotted notation on the AnnotatedRelation object to access the various annotations (e.g., .comment, .label, etc).

Note: The following, old, syntax remains supported:

```
>>> comment[acetaminophen, is_prescribed_for, pain] = ["A comment on the␣
↪acetaminophen-pain relation"]
```

Special pseudo-properties are provided for annotating is-a relations (rdfs_subclassof and rdf_type), domains (rdf_domain) and ranges (rdf_range).

```
>>> AnnotatedRelation(Drug, rdfs_subclassof, Thing).comment = ["A comment on an is-a␣
↪relation"]
```

Annotation values are usually lists of values. However, in many cases, a single value is used. Owlready2 accepts to set an annotation property to a single value, for example:

```
>>> acetaminophen.comment = "This comment replaces all existing comments on␣
↪acetaminophen"
```

1.11.2 Querying annotations

Annotation values can be obtained using the dot notation, as if they were attributes of the entity:

```
>>> print(Drug.comment)
['A first comment on the Drug class', 'A second comment', 'A third comment']

>>> print(AnnotatedRelation(acetaminophen, is_prescribed_for, pain).comment)
['A comment on the acetaminophen-pain relation']

>>> print(AnnotatedRelation(Drug, rdfs_subclassof, Thing).comment)
['A comment on an is-a relation']
```

If you expect a single value, the .first() method of the list can be used. It returns the first value of the list, or None if the list is empty.

```
>>> acetaminophen.comment.first()
'This comment replaces all existing comments on acetaminophen'
```

Note: The following, old, syntax remains supported:

```
>>> comment[acetaminophen, is_prescribed_for, pain]
```

1.11.3 Deleting annotations

To delete an annotation, simply remove it from the list.

```
>>> Drug.comment.remove("A second comment")
```

For removing **all** annotations of a given type:

```
>>> Drug.comment = []
```

1.11.4 Nested annotated relations

AnnotatedRelation can be nested if desired, as follows:

```
>>> annotr = AnnotatedRelation(acetaminophen, is_prescribed_for, pain)
>>> nested = AnnotatedRelation(annotr, comment, "A comment on the acetaminophen-pain␣
↪relation")
>>> nested.comment = ["A comment on the previous comment"]
```

1.11.5 Custom rendering of entities

The set_render_func() global function can be used to specify how Owlready2 renders entities, i.e. how they are converted to text when printing them. set_render_func() accepts a single param, a function which takes one entity and return a string.

The 'label' annotation is commonly used for rendering entities. The following example renders entities using their 'label' annotation, defaulting to their name:

```
>>> def render_using_label(entity):
...     return entity.label.first() or entity.name

```

(continues on next page)

(continued from previous page)

```
>>> set_render_func(render_using_label)

>>> Drug    # No label defined yet => use entity.name
Drug

>>> Drug.label = "The drug class"

>>> Drug
The drug class
```

The following example renders entities using their IRI:

```
>>> def render_using_iri(entity):
...     return entity.iri

>>> set_render_func(render_using_iri)

>>> Drug
http://test.org/onto.owl#Drug
```

1.11.6 Language-specific annotations

To specify the language of textual annotations, the 'locstr' (localized string) type can be used:

```
>>> Drug.comment = [ locstr("Un commentaire en Français", lang = "fr"),
...                  locstr("A comment in English", lang = "en") ]
>>> Drug.comment[0]
'Un commentaire en Français'
>>> Drug.comment[0].lang
'fr'
```

In addition, the list of values support language-specific sublists, available as '.<language code>' (e.g. .fr, .en, .es, .de,. . .). These sublists contain normal string (not locstr), and they can be modified.

```
>>> Drug.comment.fr
['Un commentaire en Français']

>>> Drug.comment.en
['A comment in English']

>>> Drug.comment.en.first()
'A comment in English'

>>> Drug.comment.en.append("A second English comment")
```

The get_lang() method does the same (but is easier to call if the lang is in a variable):

```
>>> lang = "fr"
>>> Drug.comment.get_lang(lang)
['Un commentaire en Français']
```

The get_lang_first() method return only the first language-specific string found (it is equivalent to get_lang().first()):

```
>>> lang = "fr"
>>> Drug.comment.get_lang_first(lang)
'Un commentaire en Français'
```

Warning: Modifying the language-specific sublist will automatically update the list of values (and the quad store). However, the contrary is not true: modifying the list of values does **not** update language-specific sublists.

1.11.7 Creating new classes of annotation

The AnnotationProperty class can be subclasses to create a new class of annotation:

```
>>> with onto:
...     class my_annotation(AnnotationProperty):
...         pass
```

You can also create a subclass of an existing annotation class:

```
>>> with onto:
...     class pharmaceutical_comment(comment):
...         pass

>>> acetaminophen.pharmaceutical_comment = "A comment related to pharmacology of␣
↪acetaminophen"
```

1.11.8 Full-text search (FTS)

Full-text search (FTS) can optimize search in textual properties and annotations. FTS uses Sqlite3 FTS5 implementation.

First, FTS needs to be enabled on the desired properties, by adding them to default_world.full_text_search_properties, for example for label:

```
>>> default_world.full_text_search_properties.append(label)
```

Then, FTS can be used in search as follows:

```
>>> default_world.search(label = FTS("keyword1 keyword2*"))
```

Stars can be used as joker, but only at the END of the keyword.

When using full-text search, the _bm25 argument can be used to obtain the BM25 relevance score for each entity found:

```
>>> default_world.search(label = FTS("keyword1 keyword2*"), _bm25 = True)
```

1.12 Namespaces

Ontologies can define entities located in other namespaces. An example is Gene Ontology (GO): the ontology IRI is 'http://purl.obolibrary.org/obo/go.owl', but the IRI of GO entities are not of the form 'http://purl.obolibrary.org/obo/go.owl#GO_entity' but 'http://purl.obolibrary.org/obo/GO_entity' (note the missing 'go.owl#').

1.12.1 Accessing entities defined in another namespace

These entities can be accessed in Owlready2 using a namespace. The get_namepace(base_iri) global function returns a namespace for the given base IRI.

The namespace can then be used with the dot notation, similarly to the ontology.

```
>>> # Loads Gene Ontology (~ 170 Mb), can take a moment!
>>> go = get_ontology("http://purl.obolibrary.org/obo/go.owl").load()

>>> print(go.GO_0000001) # Not in the right namespace
None

>>> obo = get_namespace("http://purl.obolibrary.org/obo/")

>>> print(obo.GO_0000001)
obo.GO_0000001

>>> print(obo.GO_0000001.iri)
http://purl.obolibrary.org/obo/obo.GO_0000001

>>> print(obo.GO_0000001.label)
['mitochondrion inheritance']
```

.get_namepace(base_iri) can also be called on an Ontology, for example:

```
>>> obo = go.get_namespace("http://purl.obolibrary.org/obo/")
```

Namespaces created on an Ontology can also be used for asserting facts and creating classes, instances,. . . :

```
>>> with obo:
>>>     class MyNewClass(Thing): pass # Create http://purl.obolibrary.org/obo/
↪MyNewClass
```

1.12.2 Creating classes in a specific namespace

When creating a Class or a Property, the namespace attribute is used to build the full IRI of the Class, and to define in which ontology the Class is defined (RDF triples are added to this ontology). The Class IRI is equals to the namespace's base IRI (base_iri) + the Class name.

An ontology can always be used as a namespace, as seen in *Classes and Individuals (Instances)*. A namespace object can be used if you want to locate the Class at a different IRI. For example:

```
>>> onto      = get_ontology("http://test.org/onto/")
>>> namespace = onto.get_namespace("http://test.org/onto/pharmaco")

>>> class Drug(Thing):
...     namespace = namespace
```

In the example above, the Drug Class IRI is "http://test.org/pharmaco/Drug", but the Drug Class belongs to the 'http://test.org/onto' ontology.

Owlready2 proposes 3 methods for indicating the namespace:

- the 'namespace' Class attribute
- the 'with namespace' statement

- if not provided, the namespace is inherited from the first parent Class

The following examples illustrate the 3 methods:

```
>>> class Drug(Thing):
...     namespace = namespace

>>> with namespace:
...     class Drug(Thing):
...         pass

>>> class Drug2(Drug):
...     # namespace is implicitely Drug.namespace
...     pass
```

1.12.3 Modifying a class defined in another ontology

In OWL, an ontology can also *modify* a Class already defined in another ontology.

In Owlready2, this can be done using the 'with namespace' statement. Every RDF triples added (or deleted) inside a 'with namespace' statement goes in the ontology corresponding to the namespace of the 'with namespace' statement.

The following example creates the Drug Class in a first ontology, and then asserts that Drug is a subclass of Substance in a second ontology.

```
>>> onto1 = get_ontology("http://test.org/my_first_ontology.owl")
>>> onto2 = get_ontology("http://test.org/my_second_ontology.owl")

>>> with onto1:
...     class Drug(Thing):
...         pass

>>> with onto2:
...     class Substance(Thing):
...         pass

...     Drug.is_a.append(Substance)
```

1.12.4 Renaming entities defined in a namespace

Owlready has no direct support for renaming entities defined in a namespace that do not correspond to an ontology. However, a simple workaround is to create an ontology with the same base IRI as the namespace, change this ontology base iri (with Ontoloy.base_iri = ...), and then destroy the ontology.

1.13 SPARQL queries

Since version 0.30, Owlready proposes 2 methods for performing SPARQL queries: the native SPARQL engine and RDFlib.

1.13.1 Native SPARQL engine

The native SPARQL engine automatically translates SPARQL queries into SQL queries, and then run the SQL queries with SQLite3.

The native SPARQL engine has better performances than RDFlib (about 60 times faster when tested on Gene Ontology, but it highly depends on queries and data). It also has no dependencies and it has a much shorter start-up time.

However, it currently supports only a subset of SPARQL.

SPARQL elements supported

- SELECT, INSERT and DELETE queries
- UNION
- OPTIONAL
- FILTER, BIND, FILTER EXISTS, FILTER NOT EXISTS
- GRAPH clauses
- SELECT sub queries
- VALUES in SELECT queries
- All SPARQL functions and aggregation functions
- Blank nodes notations with square bracket, e.g. '[a XXX]'
- Parameters in queries (i.e. '??' or '??1')
- Property path expressions, e.g. 'a/rdfs:subClassOf*', excepted those listed below

SPARQL elements not supported

- ASK, DESCRIBE, LOAD, ADD, MOVE, COPY, CLEAR, DROP, CONSTRUCT queries
- INSERT DATA, DELETE DATA, DELETE WHERE queries (you may use INSERT or DELETE instead)
- SERVICE (Federated queries)
- FROM, FROM NAMED keywords
- MINUS
- Property path expressions with parentheses of the following forms:
  - nested repeats, e.g. (a/p*)*
  - sequence nested inside a repeat, e.g. (p1/p2)*
  - negative property set nested inside a repeat, e.g. (!(p1 | p2))*

  i.e. repeats cannot contain other repeats, sequences and negative property sets.

Performing SPARQL queries

The .sparql() methods of the World object can be used to perform a SPARQL query and obtain the results. Notice that .sparql() returns a generator, so we used here the list() function to show the results. The list contains one row for each result found, with one or more columns (depending on the query).

```
>>> # Loads Gene Ontology (~ 170 Mb), can take a moment!
>>> go = get_ontology("http://purl.obolibrary.org/obo/go.owl").load()

>>> # Get the number of OWL Class in GO
>>> list(default_world.sparql("""
```

(continues on next page)

(continued from previous page)

```
            SELECT (COUNT(?x) AS ?nb)
            { ?x a owl:Class . }
    """))
[[60448]]
```

Notice that the following prefixes are automatically pre-defined:

- rdf: -> http://www.w3.org/1999/02/22-rdf-syntax-ns#
- rdfs: -> http://www.w3.org/2000/01/rdf-schema#
- owl: -> http://www.w3.org/2002/07/owl#
- xsd: -> http://www.w3.org/2001/XMLSchema#
- obo: -> http://purl.obolibrary.org/obo/
- owlready: -> http://www.lesfleursdunormal.fr/static/_downloads/owlready_ontology.owl#

In addition, Owlready automatically create prefixes from the last part of ontology IRI (without .owl extension), e.g. the ontology "http://test.org/onto.owl" with be automatically associated with the "onto:" prefix. Consequently, in most case you don't need to define prefixes (but you can still define them if you want).

The classes counted above include OWL named classes, but also some OWL constructs. One may count only named classes using a FILTER condition and the ISIRI function, as follows:

```
>>> # Get the number of OWL Class in GO
>>> list(default_world.sparql("""
            SELECT (COUNT(?x) AS ?nb)
            { ?x a owl:Class . FILTER(ISIRI(?x)) }
    """))
[[48535]]
```

We may also search for a given concept. When a query returns an entity, it returns it as an Owlready object.

```
>>> # Get the "mitochondrion inheritance" concept from GO
>>> r = list(default_world.sparql("""
            SELECT ?x
            { ?x rdfs:label "mitochondrion inheritance" . }
    """))
>>> r
[[obo.GO_0000001]]
>>> mito_inher = r[0][0]
```

Here, the resulting object 'mito_inher' is an Owlready object (here, a Class) that can be used as any other classes in Owlready.

Owlready support simple property path expressions, such as 'rdfs:subClassOf*' or 'a/rdfs:subClassOf*'. For example, we can get the superclasses of "mitochondrion inheritance" as follows:

```
>>> list(default_world.sparql("""
            SELECT ?y
            { ?x rdfs:label "mitochondrion inheritance" .
              ?x rdfs:subClassOf* ?y }
    """))
[[obo.GO_0000001], [obo.GO_0048308], [obo.GO_0048311], [obo.GO_0006996], [obo.GO_
↪0007005], [obo.GO_0051646], [obo.GO_0016043], [obo.GO_0051640], [obo.GO_0009987],␣
↪[obo.GO_0071840], [obo.GO_0051641], [obo.GO_0008150], [obo.GO_0051179]]
```

Or we can search for individuals belonging to the class “mitochondrion inheritance” or one of its descendants, as follows:

```
>>> list(default_world.sparql("""
           SELECT ?y
           { ?x rdfs:label "mitochondrion inheritance" .
             ?y a/rdfs:subClassOf* ?x }
    """))
[]
```

(Here, we have no results because Gene Ontology does not include individuals).

INSERT queries

The ontology in which the new RDF triples are inserted can be given using a “with ontology:” block or using the “WITH <ontology IRI> INSERT . . . ” syntax in SPARQL. If both are present, the “with ontology:” block takes priority.

```
>>> insertion = get_ontology("http://test.org/insertion.owl")
>>> with insertion:
...     default_world.sparql("""
           INSERT { ?x rdfs:label "héritage mitochondrial"@fr }
           WHERE  { ?x rdfs:label "mitochondrion inheritance" . }
           """)
1
```

INSERT / DELETE queries returns the number of matches found by the WHERE part.

When running INSERT / DELETE queries, Owlready tries to update the Python objects corresponding to the modified entities, if they were loaded from the quadstore.

The following example shows how to create new individuals with an INSERT query. It creates an individual for each subclass of “membrane”.

```
>>> insertion = get_ontology("http://test.org/insertion.owl")
>>> with insertion:
...     default_world.sparql("""
           INSERT { ?n rdfs:label "New individual!" . }
           WHERE  { ?x rdfs:label "membrane" .
                    ?y rdfs:subClassOf ?x .
                    BIND(NEWINSTANCEIRI(?y) AS ?n) }
           """)
```

We use here a BIND statement in order to create a new IRI, using the NEWINSTANCEIRI() function that create a new IRI for an individual, similar to those created automatically by Owlready. You may also use the more standard UUID() SPARQL function, which create a random arbitrary IRI.

The following example shows how to create OWL construct like restrictions with an INSERT query.

```
>>> insertion = get_ontology("http://test.org/insertion.owl")
>>> with insertion:
...     default_world.sparql("""
           INSERT { ?x rdfs:subClassOf [ a owl:Restriction ;
                                         owl:onProperty obo:BFO_0000050 ;
                                         owl:someValuesFrom obo:GO_0005623 ] . }
           WHERE  { ?x rdfs:label "membrane" . }
           """)
1
```

(continues on next page)

(continued from previous page)

```
>>> obo.GO_0016020.label
['membrane']
>>> obo.GO_0016020.is_a
[obo.GO_0044464, obo.BFO_0000050.some(obo.GO_0005623)]
```

DELETE queries

DELETE queries are supported; they do not need to specify the ontology from which RDF triples are deleted.

```
>>> default_world.sparql("""
        DELETE { ?r ?p ?o . }
        WHERE  {
            ?x rdfs:label "membrane" .
            ?x rdfs:subClassOf ?r .
            ?r a owl:Restriction .
            ?r ?p ?o .
        }
        """)
```

The native SPARQL engine supports queries with both a DELETE and an INSERT statement.

Parameters in SPARQL queries

Parameters allow to run the same query multiple times, with different parameter values. They have two interests. First, they increase performances since the same query can be reused, thus avoiding to parse new queries. Second, they prevent security problems by avoiding SPARQL code injection, e.g. if a string value includes quotation marks.

Parameters can be included in the query by using double question marks, e.g. “??”. Parameter values can be Owlready entities or datatype values (int, float, string, etc.). Parameter values are passed in a list after the query:

```
>>> list(default_world.sparql("""
           SELECT ?y
           { ?? rdfs:subClassOf* ?y }
    """, [mito_inher]))
[[obo.GO_0000001], [obo.GO_0048308], [obo.GO_0048311],
 [obo.GO_0006996], [obo.GO_0007005], [obo.GO_0051646],
 [obo.GO_0016043], [obo.GO_0051640], [obo.GO_0009987],
 [obo.GO_0071840], [obo.GO_0051641], [obo.GO_0008150],
 [obo.GO_0051179]]
```

Parameters can also be numbered, e.g. “??1”, “??2”, etc. This is particularly usefull if the same parameter is used multiple times in the query.

```
>>> list(default_world.sparql("""
           SELECT ?y
           { ??1 rdfs:subClassOf* ?y }
    """, [mito_inher]))
[[obo.GO_0000001], [obo.GO_0048308], [obo.GO_0048311],
 [obo.GO_0006996], [obo.GO_0007005], [obo.GO_0051646],
 [obo.GO_0016043], [obo.GO_0051640], [obo.GO_0009987],
 [obo.GO_0071840], [obo.GO_0051641], [obo.GO_0008150],
 [obo.GO_0051179]]
```

Non-standard additions to SPARQL

The following functions are supported by Owlready, but not standard:

- The SIMPLEREPLACE(a, b) function is a version of REPLACE() that does not support Regex. It works like Python or SQLite3 replace, and has better performances.
- THE LIKE(a, b) function performs similarly to the SQL Like operator. It is more limited, but faster than the Regex SPARQL functions.
- The NEWINSTANCEIRI() function create a new IRI for an instance of the class given as argument. This IRI is similar to those created by default by Owlready. Note that the function creates 2 RDF triples, asserting that the new individual is an OWL NamedIndividual and an instance of the desired class passed as argument.
- The LOADED(iri) function returns True if the entity with the given IRI is currently loaded in Python, and False otherwise.
- The STORID(iri) function returns the integer Store-ID used by Owlready in the quadstore for representing the entity.
- The DATE(), TIME() and DATETIME() functions can be used to handle date and time. They behave as in SQLite3 (see https://www.sqlite.org/lang_datefunc.html).
- The DATE_SUB(), DATE_ADD(), DATETIME_SUB and DATETIME_ADD() functions can be used to substract or add a time duration to a date or a datetime, for example : DATETIME_ADD(NOW(), “P1Y”^^xsd:duration)

In Owlready, INSERT and DELETE queries can have a GROUP BY, HAVING and/or ORDER BY clauses. This is normally not allowed by the SPARQL specification.

Prepare SPARQL queries

The .prepare_sparql() method of the World object can be used to prepare a SPARQL query. It returns a PreparedQuery object.

The .execute() method of the PreparedQuery can be used to execute the query. It takes as argument the list of parameters, if any.

Note: The .sparql() method calls .prepare_sparql(). Thus, there is no interest, in terms of performances, to use .prepare_sparql() instead of .sparql().

The PreparedQuery can be used to determine the type of query:

```
>>> query = default_world.prepare_sparql("""SELECT (COUNT(?x) AS ?nb) { ?x a␣
↪owl:Class . }""")
>>> isinstance(query, owlready2.sparql.main.PreparedSelectQuery)
True
>>> isinstance(query, owlready2.sparql.main.PreparedModifyQuery) # INSERT and/or␣
↪DELETE
False
```

The following attributes are availble on the PreparedQuery object:

- .nb_parameter: the number of parameters
- .column_names: a list with the names of the columns in the query results, e.g. [“?nb”] in the example above.
- .world: the world object for which the query has been prepared

- .sql: the SQL translation of the SPARQL query

```
>>> query.sql
'SELECT  COUNT(q1.s), 43 FROM objs q1 WHERE q1.p=6 AND q1.o=11'
```

Note: For INSERT and DELETE query, the .sql translation only involves the WHERE part. Insertions and deletions are performed in Python, not in SQL, in order to update the modified Owlready Python objects, if needed.

Open a SPARQL endpoint

The owlready2.sparql.endpoint module can be used to open a SPARQL endpoint. It requires Flask or WSGI. It contains the EndPoint class, that takes a World and can be used as a Flask page function.

The following script creates a SPARQL endpoint with Flask:

```
import flask

from owlready2 import *
from owlready2.sparql.endpoint import *

# Load one or more ontologies
go = get_ontology("http://purl.obolibrary.org/obo/go.owl").load() # (~ 170 Mb), can␣
↪take a moment!

app = flask.Flask("Owlready_sparql_endpoint")
endpoint = EndPoint(default_world)
app.route("/sparql", methods = ["GET"])(endpoint)

# Run the server with Werkzeug; you may use any other WSGI-compatible server
import werkzeug.serving
werkzeug.serving.run_simple("localhost", 5000, app)
```

And the following script does the same, but with WSGI:

```
from owlready2 import *
from owlready2.sparql.endpoint import *

# Load one or more ontologies
go = get_ontology("http://purl.obolibrary.org/obo/go.owl").load() # (~ 170 Mb), can␣
↪take a moment!

endpoint = EndPoint(default_world)
app = endpoint.wsgi_app

# Run the server with Werkzeug; you may use any other WSGI-compatible server
import werkzeug.serving
werkzeug.serving.run_simple("localhost", 5000, app)
```

You can then query the endpoint, e.g. by opening the following URL in your browser:

> http://localhost:5000/sparql?query=SELECT(COUNT(?x)AS%20?nb)\protect\T1\textbraceleft?x%20a%20owl:Class.\protect\T1\textbraceright

1.13.2 Using RDFlib for executing SPARQL queries

The Owlready quadstore can be accessed as an RDFlib graph, which can be used to perform SPARQL queries:

```
>>> graph = default_world.as_rdflib_graph()
>>> r = list(graph.query("""SELECT ?p WHERE {
  <http://www.semanticweb.org/jiba/ontologies/2017/0/test#ma_pizza> <http://www.
↪semanticweb.org/jiba/ontologies/2017/0/test#price> ?p .
}"""))
```

The results can be automatically converted to Python and Owlready using the .query_owlready() method instead of .query():

```
>>> r = list(graph.query_owlready("""SELECT ?p WHERE {
  <http://www.semanticweb.org/jiba/ontologies/2017/0/test#ma_pizza> <http://www.
↪semanticweb.org/jiba/ontologies/2017/0/test#price> ?p .
}"""))
```

1.14 Worlds

Owlready2 stores every triples in a 'World' object, and it can handles several Worlds in parallel. 'default_world' is the World used by default.

1.14.1 Persistent world: storing the quadstore in an SQLite3 file database

Owlready2 uses an optimized quadstore. By default, the quadstore is stored in memory, but it can also be stored in an SQLite3 file. This allows persistance: all ontologies loaded and created are stored in the file, and can be reused later. This is interesting for big ontologies: loading huge ontologies can take time, while opening the SQLite3 file takes only a fraction of second, even for big files. It also avoid to load huge ontologies in memory, if you only need to access a few entities from these ontologies.

The .set_backend() method of World sets the SQLite3 filename associated to the quadstore, for example:

```
>>> default_world.set_backend(filename = "/path/to/your/file.sqlite3")
```

Note: If the quad store is not empty when calling .set_backend(), RDF triples are automatically copied. However, this operation can have a high performance cost (especially if there are many triples).

When using persistence, the .save() method of World must be called for saving the actual state of the quadstore in the SQLite3 file:

```
>>> default_world.save()
```

Storing the quadstore in a file does not reduce the performance of Owlready2 (actually, it seems that Owlready2 performs a little *faster* when storing the quadstore on the disk).

To reload an ontology stored in the quadstore (when the corresponding OWL file has been updated), the reload and reload_if_newer optional parameters of .load() can be used (the former reload the ontology, and the latter reload it only if the OWL file is more recent).

By default, Owlready2 opens the SQLite3 database in exclusive mode. This mode is faster, but it does not allow several programs to use the same database simultaneously. If you need to have several Python programs that access simultaneously the same Owlready2 quadstore, you can disable the exclusive mode as follows:

```
>>> default_world.set_backend(filename = "/path/to/your/file.sqlite3", exclusive =␣
↪False)
```

1.14.2 Using several isolated Worlds

Owlready2 can support several, isolated, Worlds. This is interesting if you want to load several version of the same ontology, for example before and after reasoning.

A new World can be created using the World class:

```
>>> my_world = World()
>>> my_second_world = World(filename = "/path/to/quadstore.sqlite3")
```

Ontologies are then created and loaded using the .get_ontology() methods of the World (when working with several Worlds, this method replaces the get_ontology() global function):

```
>>> onto = my_world.get_ontology("http://test.org/onto/").load()
```

The World object can be used as a pseudo-dictionary for accessing entities using their IRI. (when working with several Worlds, this method replaces the IRIS global pseudo-dictionary):

```
>>> my_world["http://test.org/onto/my_iri"]
```

Finally, the reasoner can be executed on a specific World:

```
>>> sync_reasoner(my_world)
```

1.14.3 Working with RDFlib

Owlready2 uses an optimized RDF quadstore. This quadstore can also be accessed as an RDFlib graph as follows:

```
>>> graph = default_world.as_rdflib_graph()
```

In particular, the RDFlib graph can be used for performing SPARQL queries:

```
>>> r = list(graph.query("""SELECT ?p WHERE {
  <http://www.semanticweb.org/jiba/ontologies/2017/0/test#ma_pizza> <http://www.
↪semanticweb.org/jiba/ontologies/2017/0/test#price> ?p .
}"""))
```

The results can be automatically converted to Python and Owlready using the .query_owlready() method instead of .query():

```
>>> r = list(graph.query_owlready("""SELECT ?p WHERE {
  <http://www.semanticweb.org/jiba/ontologies/2017/0/test#ma_pizza> <http://www.
↪semanticweb.org/jiba/ontologies/2017/0/test#price> ?p .
}"""))
```

Note: Owlready now include its own SPARQL engine, documented here: *SPARQL queries*.

Owlready blank nodes can be created with the graph.BNode() method:

```
>>> bn = graph.BNode()
>>> with onto:
...     graph.add((bn, rdflib.URIRef("http://www.w3.org/1999/02/22-rdf-syntax-ns#type
↪"), rdflib.URIRef("http://www.w3.org/2002/07/owl#Class")))
```

1.15 Parallelism, multiprocessing and synchronization

Parallelism consist in executing several part of your program in parallel. Three options are possible:

- cooperative microthread (e.g. greenlets with GEvent): it allows running several "greenlet" in parallel, switching from one to others, but it does not actually run several commands in parallel and increase performances. Nevertheless, it is very interesting in a server setting:
- multi-thread parallelism: it allows sharing data and objects between threads, however, Python has poor multithreading supports (due to the global interpreter lock (GIL), only one thread at a time may execute Python commands).
- multi-process parallelism: it allows executing Python commands in parallel, however, data sharing is more difficult and objects cannot be shared between processes. In addition, keep in mind that Owlready does not update the local Python objects from the quadstore if they are modified by other processes.

Owlready (>= 0.41) supports all options:

- cooperative microthread can be used in a server setting, in order to let the server answer a simple/small request while a long request is running.
- multi-thread parallelism can be used to parallelize long SPARQL queries (only the SQL query is parallelized, allowing to run Python commands meanwhile). There is no other interesting in multi-threading, due to Python's GIL.
- multi-process parallelism can be used to run several process in parallel.

Two difficulties arise when using parallelism:

- Sharing data between processes is complex. When using Owlready, the easier solution is to put the quadstore with the ontology data on disk.
- Sensible parts of the code must be synchronized, e.g. one should avoid that severa processes write in the quadstore at the same time.

Several web application servers use multiple processes, and thus you will also encounter these difficulties when using them.

1.15.1 Parallelized file parsing

For huge OWL file (> 8 Mb), Owlready (>= 0.41) automatically uses a separate process for parsing the file (the main process being in charge of inserting triples in the quadstore). This provide a 25% performance boost on huge ontologies.

1.15.2 Thread-based parallel execution of SPARQLqueries

This is the simplest options, and probably the best if you have long SPARQL queries. Since version 0.41, Owlready supports some level of thread-based parallelization, for increasing performances by executing several SPARQL queries in parallel. It does not require to care about synchronization or data sharing.

In order to use this feature, you first need to use a World stored in a local file, to deactive exclusive mode and to activate thread parallelism support, as follows:

```
>>> default_world.set_backend(filename = "my_quadstore.sqlite3", exclusive = False,␣
↪enable_thread_parallelism = True)
```

When thread parallelism is activated, Owlready opens 3 additional connexions to the SQLite3 database storing the quadstore, allowing 3 parallel threads.

Then, the quadstore must be saved on disk before running parallel queries, as follows:

```
>>> default_world.save()
```

Executing many SPARQL queries in parallel

The owlready2.sparql.execute_many() function can be used to execute several prepared SPARQL queries in parallel. Both SELECT and INSERT/DELETE queries are supported.

execute_many() will start 3 threads for executing the queries in parallel, and returns a list of query results.

You may expect up to 100% performance boost, especially when the queries are long and complex and the number of results is small (currently, Owlready only parallelize the SQL execution, but not the loading of the resulting objects from the quadstore).

Here is a typical usage:

```
>>> my_onto = get_ontology("XXX ontology IRI here")

>>> queries = [
...     default_world.prepare_sparql("""XXX First SPARQL query here"""),
...     ...,
... ]

>>> queries_params = [
...     [], # First SPARQL query parameters
...     ...,
... ]

>>> import owlready2.sparql
>>> results = [list(gen) for gen in owlready2.sparql.execute_many(my_onto, queries,␣
↪queries_params)]
```

If you are also using Gevent (or another similar library), you may use the Gevent thread pool. This can be done by providing a “spawn” function to execute_many(). The spawn function must accept a callable with no argument, start a thread executing that callable, and return the thread object (which is expected to have a .join() method). Here is an example for Gevent:

```
>>> import gevent.hub
>>> gevent_spawn = gevent.hub.get_hub().threadpool.apply_async
>>> results = [list(gen) for gen in owlready2.sparql.execute_many(my_onto, queries,␣
↪queries_params, gevent_spawn)]
```

Executing a single SPARQL query in parallel

A single SPARQL query can be executed in parallel, in a separate thread. The query will not run faster (it will rather takes a little more time), but the main thread will be let available for other tasks. This can be interesting e.g. on a server, where a long query can be parallelized; meanwhile, the main thread may answer to other clients.

```
>>> query = default_world.prepare_sparql("""XXX SPARQL query here""")
>>> query.execute(spawn = True)
```

Similarly, you may want to use the Gevent thread pool, as follows:

```
>>> import gevent.hub
>>> gevent_spawn = gevent.hub.get_hub().threadpool.apply_async
>>> query.execute(spawn = gevent_spawn)
```

1.15.3 Cooperative microthreads (e.g. GEvent)

Microthreads will not improve the performances of Owlready, however, they will allow running several tasks in parallel, which is interesting if you need to perform small tasks during long tasks (e.g. in a server), or if some part of your program is waiting on an external, non-Python, task (e.g. a network call, including the use of a server database like Postgresql).

Synchronization

For using Owlready with cooperative microthreads, you need to:

- Use a custom lock for the quadstore. By default, Owlready use the internal SQLite3 database as a lock; this does not work with microthreads because all microthreads share the same SQLite3 connexion. The solution is to use a custom lock, for example with GEvent :

  ```
  >>> gevent.lock
  >>> default_world.set_backend(filename = "your_quadstore.sqlite3",
  ...                           lock     = gevent.lock.RLock())
  ```

- Perform each modification to an ontology inside a “with ontology:” block. This prevents multiple writes at the same time. For improving performances, you should also avoid long computation inside “with ontology:” blocks.
- Switch to other microthreads when desired (e.g. by calling gevent.sleep(0)). To let other microthreads write in the quadstore, you should do that outside “with ontology:” blocks.

Other synchronization tasks (listed below, for multiprocessing) are not needed for microthreads.

1.15.4 Multiprocessing

Multiprocessing requires synchronization, which can be very complex (and may have a significant performance cost).

Multiprocessing is recommended mostly when using a read-only quadstore, because Owlready does not update the local Python objects from the quadstore if they are modified by another process.

Synchronization

For using Owlready with multiple processes, and sharing the quadstore between processes, you need to:

- Store the quadstore on disk.
- Open the quadstore in non-exclusive mode (exclusive = False in set_backend()).
- Perform each modification to an ontology inside a “with ontology:” block. Owlready maintain a lock for each quadstore, which prevents multiple writes at the same time. Thus, for improving performances, you should also avoid long computation inside “with ontology:” blocks.

- Call World.save() at the end of each "with ontology:" block, in order to commit the changes to the quadstore database.

Server example

This section gives a small example of a multi-process server using a shared Owlready quadstore.

The example uses Flask and Gunicorn. It provides 2 URL: the first one (/gen) creates 5 new instances of the C class. The second (/test) returns the ID of the current process and the number of instances in the quadstore.

```
import sys, os, flask, time
from owlready2 import *

default_world.set_backend(filename = "/tmp/t.sqlite3", exclusive = False)

onto = get_ontology("http://test.org/onto.owl")

with onto:
  class C(Thing): pass
  default_world.save()


app = flask.Flask("OwlreadyBench")

@app.route("/gen")
def gen():
  with onto:
    for i in range(5):
      c = C()
      c.label = [os.getpid()]
      print(c, c.storid)
    default_world.save()
  return ""

@app.route("/test")
def test():
  time.sleep(0.02)
  nb = len(list(C.instances()))
  return "%s %s" % (os.getpid(), nb)
```

You can run this server in multiprocessor mode with Gunicorn as follows:

```
gunicorn -b 127.0.0.1:5000 --preload -w 5 --worker-class=gevent test:app
```

where "test" is the previous file's name (without ".py"), and 5 in "-w 5" is recommended to be the number of CPU plus 1 (here, my computer has 4 CPU, thus -w 5).

Then, after running the server, you can use the following script to make 100 concurrent calls to /gen, and then 10 concurrent calls to /test:

```
from urllib.request import *

import eventlet, eventlet.green.urllib.request
def fetch(url): return eventlet.green.urllib.request.urlopen(url).read()

urls = ["http://localhost:5000/gen"] * 100
pool = eventlet.GreenPool()
```

(continues on next page)

(continued from previous page)

```
for body in pool.imap(fetch, urls): pass

urls = ["http://localhost:5000/test"] * 10
pool = eventlet.GreenPool()
for body in pool.imap(fetch, urls): print(body)
```

As the 10 calls to /test are executed by different processes, this allows to verify that the various processes have access to all the created instances (normally, 500 instances).

The previous server example can also be run with uWSGI as follows:

```
uwsgi --http 127.0.0.1:5000 --plugin python -p 5 --module test:app
```

1.16 SWRL rules

SWRL rules can be used to integrate 'if… then…' rules in ontologies.

Note: loading SWRL rules is **only** supported from RDF/XML and NTriples files, but not from OWL/XML files.

1.16.1 Creating SWRL rules

The Imp class ("Implies") represent a rule. The easiest way to create a rule is to define it using a Protégé-like syntax, with the .set_as_rule() method.

The following example use a rule to compute the per-tablet cost of a drug:

```
>>> onto = get_ontology("http://test.org/drug.owl")

>>> with onto:
...     class Drug(Thing): pass
...     class number_of_tablets(Drug >> int, FunctionalProperty): pass
...     class price(Drug >> float, FunctionalProperty): pass
...     class price_per_tablet(Drug >> float, FunctionalProperty): pass
...
...     rule = Imp()
...     rule.set_as_rule("""Drug(?d), price(?d, ?p), number_of_tablets(?d, ?n),␣
↪divide(?r, ?p, ?n) -> price_per_tablet(?d, ?r)""")
```

We can now create a drug, run the reasoner (only Pellet support inferrence on data property value) and print the result:

```
>>> drug = Drug(number_of_tablets = 10, price = 25.0)
>>> sync_reasoner_pellet(infer_property_values = True, infer_data_property_values =␣
↪True)
>>> drug.price_per_tablet
2.5
```

1.16.2 Displaying rules

The str() Python function can be used to format rules, for example:

```
>>> str(rule)
'Drug(?d), price(?d, ?p), number_of_tablets(?d, ?n), divide(?r, ?p, ?n) -> price_per_
↪tablet(?d, ?r)'
```

(continues on next page)

(continued from previous page)

1.16.3 Modifying rules manually

Owlready also allows to access to the inner content of rules. Each rules have a body (= conditions) and head (= consequences) :

```
>>> rule.body
[Drug(?d), price(?d, ?p), number_of_tablets(?d, ?n), divide(?r, ?p, ?n)]
>>> rule.head
[price_per_tablet(?d, ?r)]
```

Body and head are list of SWRL atoms. The attributes of each atom can be read and modified:

```
>>> rule.body[0]
Drug(?d)
>>> rule.body[0].class_predicate
drug.Drug
>>> rule.body[0].arguments
[?d]
```

Please refer to SWRL documentation for the list of atoms and their description. One notable difference is that Owlready always use the "arguments" attributes for accessing arguments, while SWRL uses sometimes "arguments" and sometimes "argument1" and "argument2".

1.17 General class axioms

General class axioms are axioms of the form "A is a B" where "A" is not a named class, but a class construct (e.g. an intersection, a union or a restriction).

1.17.1 Creating a general class axiom

One can create a general class axiom as follows:

```
>>> with onto:
...     gca = GeneralClassAxiom(onto.Disorder & onto.has_location.some(onto.Heart)) #␣
↪Left side
...     gca.is_a.append(onto.CardiacDisorder) # Right side
```

The GeneralClassAxiom class take as parameter the left side class construct.

The right side is available as the .is_a attribute. Notice that one may add several right sides, by calling is_a.append multiple times.

The left side is available as the .left_side attribute.

1.17.2 Accessing general class axioms

One can list general class axioms with Ontology.general_class_axioms:

```
>>> gcas = list(onto.general_class_axioms())
```

One can then test the left side by comparison, for example:

```
>>> searched_left_side = onto.Disorder & onto.has_location.some(onto.Heart)
>>> for gca in gcas:
...     if gca.left_side == searched_left_side: print("Found!")
```

1.18 PyMedTermino2

1.18.1 Introduction

PyMedTermino (Medical Terminologies for Python) is a Python module for easy access to the main medical terminologies in Python. The following terminologies are supported:

- All terminologies in UMLS, including: - SNOMED CT - ICD10 - MedDRA
- ICD10 in French (CIM10)

The main features of PyMedTermino are:

- A single API for accessing all terminologies
- Optimized full-text search
- Access to terms, synonyms and translations
- Manage concepts and relations between concepts
- Mappings between terminologies (e.g. via UMLS or manual mapping)

PyMedTermino has been designed for "batch" access to terminologies; it is *not* a terminology browser (althought it can be used to write a terminology browser in Python).

The first version of PyMedTermino was an independent Python package. The second version (PyMedTermino2) is integrated with Owlready2, and store medical terminologies as OWL ontlogies. This allows relating medical terms from terminologies with user created concepts.

UMLS data is not included, but can be downloaded for free (see Intallation below). Contrary to PyMedTermino1, PyMedTermino2 do not require a connection to an external UMLS database: it imports UMLS data in its own local database, automatically.

If you use PyMedTermino in scientific works, **please cite the following article**:

> **Lamy JB**, Venot A, Duclos C. PyMedTermino: an open-source generic API for advanced terminology services. **Studies in health technology and informatics 2015**;210:924-928

1.18.2 Installation

1. Install Python 3.7 and Owlready2 (if not already done). **PyMedTermino2 requires Python >= 3.7 for importing UMLS** (However, after importing the data in the quadstore, it can be used with Python 3.6 if you really need to).
2. After registration with NLM, download UMLS data (Warning: some restriction may apply depending on country; see UMLS licence and its SNOMED CT appendix):

- https://www.nlm.nih.gov/research/umls/licensedcontent/umlsknowledgesources.html

  PyMedTermino2 suports both the "Full UMLS Release Files" and the "UMLS Metathesaurus Files", but the latter is recommended since it is faster to uncompress. E.g. download "umls-2019AA-metathesaurus.zip". Do not unzip it!

3. Import UMLS data in Python as follows:

```
>>> from owlready2 import *
>>> from owlready2.pymedtermino2 import *
>>> from owlready2.pymedtermino2.umls import *
>>> default_world.set_backend(filename = "pym.sqlite3")
>>> import_umls("umls-2019AA-metathesaurus.zip", terminologies = ["ICD10", "SNOMEDCT_
↪US", "CUI"])
>>> default_world.save()
```

were:

- "pym.sqlite3" is the quadstore file in which the data are stored.
- ["ICD10", "SNOMEDCT_US", "CUI"] are the terminologies imported (valid codes are UMLS code, plus "CUI" for CUI). If the 'terminologies' parameter is missing, all terminologies are imported.

To import also suppressed/deprecated concept, add the following parameter: remove_suppressed = "".

The importation can take several minutes or hours, depending on the number of terminologies imported.

4. Import French ICD10 (optional):

```
>>> from owlready2.pymedtermino2.icd10_french import *
>>> import_icd10_french()
>>> default_world.save()
```

1.18.3 SNOMED CT

Loading

To load SNOMED CT in Python:

```
>>> from owlready2 import *
>>> default_world.set_backend(filename = "pym.sqlite3")
>>> PYM = get_ontology("http://PYM/").load()
>>> SNOMEDCT_US = PYM["SNOMEDCT_US"]
```

Here, 'PYM' is the abbreviation for PyMedTermino. PYM can be indiced with a terminology code, to obtain the corresponding terminology object (here, SNOMEDCT_US).

Concepts

The SNOMEDCT_US object represents the SNOMED CT terminology. A SNOMED CT concept can be obtained from its code (in the following example, 302509004, which is the code for the heart concept) by indexing this object with curly brackets:

```
>>> concept = SNOMEDCT_US[302509004]
>>> concept
SNOMEDCT_US["302509004"] # Entire heart
```

The has_concept() method can be used to verify if a code corresponds to a concept or not:

```
>>> SNOMEDCT_US.has_concept("invalid_code")
False
```

Each concept has a code, available as the name of the entity, and a preferred term, available as the label RDF annotation:

```
>>> concept.name
'302509004'
>>> concept.label
['Entire heart']
>>> concept.label.first()
'Entire heart'
```

SNOMED CT also proposes synonym terms, available via the 'synonyms' annotation :

```
>>> concept.synonyms
['Entire heart (body structure)']
```

The 'terminology' attribute contains the terminology of the concept:

```
>>> concept.terminology
PYM["SNOMEDCT_US"] # US Edition of SNOMED CT
```

Full-text search

The search() method allows full-text search in SNOMED CT terms (including synonyms):

```
>>> SNOMEDCT_US.search("Cardiac structure")
[SNOMEDCT_US["24964005"] # Cardiac conducting system structure
, SNOMEDCT_US["10746000"] # Cardiac septum structure
...]
```

Full-text search uses the FTS engine of SQLite, it is thus possible to use its functionalities. For example, for searching for all words beginning by a given prefix:

```
>>> SNOMEDCT_US.search("osteo*")
[SNOMEDCT_US["66467005"] # Osteochondromatosis
, SNOMEDCT_US["40970001"] # Chronic osteomyelitis
...]
```

Is-a relations: parent and child concepts

The "parents" and "children" attributes return the list of parent and child concepts (i.e. the concepts with is-a relations):

```
>>> concept.parents
[SNOMEDCT_US["116004006"] # Entire hollow viscus
, SNOMEDCT_US["187639008"] # Entire thoracic viscus
, SNOMEDCT_US["80891009"] # Heart structure
]
>>> concept.children
[SNOMEDCT_US["195591003"] # Entire transplanted heart
]
```

The ancestor_concepts() and descendant_concepts() methods return all the ancestor concepts (parents, parents of parents, and so on) and the descendant concepts (children, children of children, and so on) :

```
>>> concept.ancestor_concepts()
[SNOMEDCT_US["302509004"] # Entire heart
, SNOMEDCT_US["116004006"] # Entire hollow viscus
, SNOMEDCT_US["118760003"] # Entire viscus
...]
>>> concept.descendant_concepts()
[SNOMEDCT_US["302509004"] # Entire heart
, SNOMEDCT_US["195591003"] # Entire transplanted heart
]
```

Both methods remove dupplicates automatically. They also include the starting concept in the results. If you do not want it, use the 'include_self' parameter:

```
>>> concept.descendant_concepts(include_self = False)
[SNOMEDCT_US["195591003"] # Entire transplanted heart
]
```

PyMedTermino2 concepts are OWL and Python classes. As a consequence, you can use the Python issubclass() function to test whether a concept is a descendant of another:

```
>>> issubclass(concept, SNOMEDCT_US["272625005"])
True
```

Part-of relations

"part_of" and "has_part" attributes provide access to subparts or superpart of the concept:

```
>>> concept.part_of
[SNOMEDCT_US["362010009"] # Entire heart AND pericardium
]
>>> concept.has_part
[SNOMEDCT_US["244258000"] # Entire marginal branch of right coronary artery
, SNOMEDCT_US["261405004"] # Entire atrium
, SNOMEDCT_US["244378006"] # Lateral atrioventricular leaflet
...]
```

Other relations

The "get_class_properties" method returns the set of relations available for a given concept. Is-a relations are never included in this list, and are handled with the "parents" and "children" attributes previously seen, however part-of relations are included.

```
>>> concept = SNOMEDCT_US["3424008"]
>>> concept
SNOMEDCT_US["3424008"] # Tachycardia
>>> concept.get_class_properties()
{PYM.mapped_to, PYM.case_significance_id, PYM.unifieds, PYM.terminology, rdf-schema.
↪label, PYM.subset_member, PYM.definition_status_id, PYM.synonyms, PYM.has_
↪interpretation, PYM.active, PYM.interprets, PYM.effective_time, PYM.ctv3id, PYM.
↪groups, PYM.has_finding_site, PYM.type_id}
```

Each relation corresponds to an attribute in the concept. The name of the attribute is the part after the '.', e.g. for 'PYM.interprets' the name is 'interprets'. The attribute's value is a list with the corresponding values:

```
>>> concept.has_finding_site
[SNOMEDCT_US["24964005"] # Cardiac conducting system structure
]
>>> concept.interprets
[SNOMEDCT_US["364075005"] # Heart rate
]
```

Relation groups

In SNOMED CT, relations can be grouped together. The "groups" attribute returns the list of groups. It is then possible to access to the group's relation.

```
>>> concept = SNOMEDCT_US["186675001"]
>>> concept
SNOMEDCT_US["186675001"] # Viral pharyngoconjunctivitis
>>> concept.groups
[<Group 453170_0> # mapped_to=Viral conjunctivitis, unspecified
, <Group 453170_3> # has_causative_agent=Virus ; has_associated_
↪morphology=Inflammation ; has_finding_site=Pharyngeal structure ; has_pathological_
↪process=Infectious process
, <Group 453170_4> # has_causative_agent=Virus ; has_associated_
↪morphology=Inflammation ; has_finding_site=Conjunctival structure ; has_
↪pathological_process=Infectious process
>>> concept.groups[2].get_class_properties()
{PYM.has_causative_agent, PYM.has_associated_morphology, PYM.has_finding_site, PYM.
↪has_pathological_process}
>>> concept.groups[2].has_finding_site
[SNOMEDCT_US["29445007"] # Conjunctival structure
]
>>> concept.groups[2].has_associated_morphology
[SNOMEDCT_US["23583003"] # Inflammation
]
```

Iterating over SNOMED CT

To obtain the terminology's first level concepts (i.e. the root concepts), use the children attribute of the terminology:

```
>>> SNOMEDCT_US.children
[SNOMEDCT_US["138875005"] # SNOMED CT Concept
]
```

The descendant_concepts() method returns all concepts in SNOMED CT.

```
>>> for concept in SNOMEDCT_US.descendant_concepts(): [...]
```

1.18.4 ICD10

Loading modules

To load SNOMED CT in Python:

```
>>> from owlready2 import *
>>> default_world.set_backend(filename = "pym.sqlite3")
>>> PYM = get_ontology("http://PYM/").load()
>>> ICD10 = PYM["ICD10"]
```

Or, for the French version (if you imported it during installation):

```
>>> CIM10 = PYM["CIM10"]
```

CIM10 can be used as ICD10.

Concepts

The ICD10 object allows to access to ICD10 concepts. This object behaves similarly to the SNOMED CT terminology previously described (see *SNOMED CT*).

```
>>> ICD10["E10"]
ICD10["E10"] # Insulin-dependent diabetes mellitus
>>> ICD10["E10"].parents
[ICD10["E10-E14.9"] # Diabetes mellitus
]
>>> ICD10["E10"].ancestor_concepts()
[ICD10["E10"] # Insulin-dependent diabetes mellitus
, ICD10["E10-E14.9"] # Diabetes mellitus
, ICD10["E00-E90.9"] # Endocrine, nutritional and metabolic diseases
]
```

ICD10 being monoaxial, the parents list always includes at most one parent.

1.18.5 UMLS

Loading modules

```
>>> from owlready2 import *
>>> default_world.set_backend(filename = "pym.sqlite3")
>>> PYM = get_ontology("http://PYM/").load()
>>> CUI = PYM["CUI"]
```

UMLS concepts (CUI)

In UMLS, CUI correspond to concepts: a given concept gathers equivalent terms or codes from various terminologies.

CUI can be accessed with the UMLS_CUI terminology:

```
>>> concept = CUI["C0085580"]
>>> concept
CUI["C0085580"] # Essential hypertension
>>> concept.name
'C0085580'
>>> concept.label
['Essential hypertension']
>>> concept.synonyms
['Essential (primary) hypertension', 'Idiopathic hypertension', 'Primary hypertension
↪', 'Systemic primary arterial hypertension', 'Essential hypertension (disorder)']
```

(continues on next page)

(continued from previous page)

```
```

Relations of CUI are handled in the same way than for SNOMED CT (see above), for example:

```
>>> concept.get_class_properties()
{PYM.originals, PYM.terminology, rdf-schema.label, PYM.synonyms}
```

Relation with source terminologies

The originals attribute of a CUI concept contains the corresponding concepts in UMLS sources terminologies:

```
>>> concept.originals
[SNOMEDCT_US["59621000"] # Essential hypertension
, CIM10["I10"] # Hypertension essentielle (primitive)
, ICD10["I10"] # Essential (primary) hypertension
]
```

The inverse attribute is unifieds. For concepts in the source terminologies, it contains the corresponding CUI (some concepts may be associated with several CUI):

```
>>> ICD10["I10"].unifieds
[CUI["C0085580"] # Essential hypertension
]
```

Mapping between terminologies

PyMedTermino uses the '>>' operator for mapping from a terminology to another. For example, you can map a SNOMED CT concept to UMLS as follows:

```
>>> SNOMEDCT_US[186675001]
SNOMEDCT_US["186675001"] # Viral pharyngoconjunctivitis
>>> SNOMEDCT_US[186675001] >> CUI
Concepts([
  CUI["C0542430"] # Viral pharyngoconjunctivitis
])
```

Or you can map a UMLS concept to ICD10:

```
>>> CUI["C0542430"] >> ICD10
Concepts([
  ICD10["B30.2"] # Viral pharyngoconjunctivitis
])
```

Finally, you can map directly from a terminology in UMLS to another terminology in UMLS, for example from SNOMED CT to ICD10:

```
>>> SNOMEDCT_US[186675001] >> ICD10
Concepts([
  ICD10["B30.9"] # Viral conjunctivitis, unspecified
])
```

The direct mapping considers 'mapped_to' relations available first, and default to mapping using CUI.

1.18.6 Set of concepts

The Concepts class implements a set of concepts.

```
>>> concepts = PYM.Concepts([ ICD10["E10"], ICD10["E11"], ICD10["E12"] ])
>>> concepts
Concepts([
  ICD10["E10"] # Insulin-dependent diabetes mellitus
, ICD10["E12"] # Malnutrition-related diabetes mellitus
, ICD10["E11"] # Non-insulin-dependent diabetes mellitus
])
```

Concepts class inherits from Python's set and supports all its methods (such as add(), remove(), etc).

Concepts can be used to map several concepts simultaneously, using the '>>' operator, for example:

```
>>> PYM.Concepts([ ICD10["E10"], ICD10["E11"], ICD10["E12"] ]) >> SNOMEDCT_US
Concepts([
  SNOMEDCT_US["44054006"] # Type 2 diabetes mellitus
, SNOMEDCT_US["46635009"] # Type 1 diabetes mellitus
, SNOMEDCT_US["75524006"] # Malnutrition related diabetes mellitus
])
```

In addition, the Concepts class also provides advanced terminology-oriented methods:

- keep_most_generic() keeps only the most generic concepts in the set (i.e. it removes all concepts that are a descendant of another concept in the set)
- keep_most_specific() keeps only the most specific concepts in the set (i.e. it removes all concepts that are an ancestor of another concept in the set)
- lowest_common_ancestors() computes the lower common ancestors
- find(c) search the set for a concept that is a descendant of c (including c itself)
- extract(c) search the set for all concepts that are descendant of c (including c itself)
- subtract(c) return a new set with all concepts in the set, except those that are descendant of c (including c itself)
- subtract_update(c) remove from the set for all concepts that are descendant of c (including c itself)
- all_subsets() computes all subsets included in the set.
- imply(other) returns True if all concepts in the 'other' set are descendants of (at least) one of the concepts in the set
- is_semantic_subset(other) returns True if all concepts in this set are descendants of (at least) one of the concept in the 'other' set
- is_semantic_superset(other) returns True if all concepts in this set are ancestors of (at least) one of the concept in the 'other' set
- is_semantic_disjoint(other) returns True if all concepts in this set are semantically disjoint from all concepts in the 'other' set
- semantic_intersection(other) returns the intersection of the set with 'other', considering is-a relations between the concepts in the sets
- remove_entire_families(only_family_with_more_than_one_child = True) replaces concepts in the set by their parents, whenever all the children of the parent are present

1.19 Observation framework

Owlready2 provides an observation framework in the owlready2.observe module. It allows adding listeners to any entity of an ontology, in order to be notified when the entity is modified.

1.19.1 Adding and removing listeners

Let us create a (very) small ontology:

```
>>> onto = get_ontology("http://test.org/test.owl")
>>> with onto:
...   class Pizza(Thing): pass
...   class price(Thing >> float): pass
...   pizza = Pizza()
```

And then import the observe module and add a listener to pizza:

```
>>> from owlready2.observe import *
>>> def listener(entity, props):
...     print("Listener:", entity, props)
>>> observe(pizza, listener)
```

The observe() function is used to add a listener to an entity.

Whenever relation are added or removed to the entity, listener is called:

```
>>> pizza.price = [11.0]
Listener: 305 [304]
```

The listener receives two arguments: the entity and a list of properties (NB unless you coalesce event as explained below, the list includes a single value). For performance purpose, Observe uses "store-IDs" as argument for the entity and the properties, and not Python objects (hence you see integer values above). Here, 305 is the "store-ID" of the pizza entity and 304 the "store-ID" of the price property (NB the number may differ).

You may convert store-IDs to Python objects with World._get_by_storid(storid). Here is a modified listener that shows entity and property objects instead of store-IDs:

```
>>> def listener(entity, props):
...     entity =   default_world._get_by_storid(entity)
...     props  = [ default_world._get_by_storid(prop) for prop in props ]
...     print("Listener:", entity, props)

>>> unobserve(pizza) # Remove previous listener
>>> observe(pizza, listener)

>>> pizza.price = [11.0]
Listener: onto.pizza [onto.price]
```

The unobserve() function is used to remove a listener from an entity (if no listener is given, all listeners are removed).

1.19.2 Coalescing events

The coalesced_observations environment can be used to coalesce events and listener calls.

For instance, the following code generates 3 calls to the listener:

```
>>> pizza.price.append(12.0)
Listener: onto.pizza [onto.price]
>>> pizza.label = ["Pizz", "Test pizza"]
Listener: onto.pizza [rdf-schema.label]
Listener: onto.pizza [rdf-schema.label]
```

Since two labels are added, there are 2 calls for the set label operation. These multiple calls can be problematic if the listener has a performance cost (e.g. updating the user interface).

Multiple calls can be coalesced and merged using the coalesced_observations environment, as follows:

```
>>> with coalesced_observations:
...     pizza.price.append(13.0)
...     pizza.label = ["Pizz2", "Test pizza2"]
Listener: onto.pizza [onto.price, rdf-schema.label]
```

No call to listeners are emitted inside the "with coalesced_observations" block, and a single call is emitted at the end, possibly with more than one property.

1.19.3 Stopping observation

Using the observation framework may have a performance cost. After using it, if you no longer need it, you should stop it by calling stop_observing(), as follows:

```
>>> stop_observing(default_world)
```

1.20 Development

1.20.1 Development installation

Due due legacy compatibility, the development installation needs to be done

- either manually, by following these steps:
  1. Create a directory (e.g. `src/`).
  2. Add this directory to the $PYTHONPATH shell variable (= traditional way).
  3. Put Owlready sources in that directory (in a subdirectory named `src/owlready2/`).
- or with pip by following these steps:
  1. Create a virtual environment for development and activate it.
  2. Create an directory with an arbitrary name, e.g. `mkdir owlready_dev`.
  3. Move or cloning the Owlready2 repository into this directory and change into it.
  4. Run `pip install -e .[test]` inside of this Owlready directory.
  5. In case *Python.h* is missing, install python3-dev (e.g. `sudo apt-get install python3-dev`).
  6. Run the *setup_develop_mode.py* script : `python setup_develop_mode.py` inside of this Owlready directory (there are explainations in the script, why this is necessary).

Finally, To test everything, cd into the **'test'** directory and run `python regtest.py`.

1.21 Differences between Owlready version 1 and 2

This section summarizes the major differences between Owlready version 1 and 2.

1.21.1 Creation of Classes, Properties and Individuals

The 'ontology' parameters is now called 'namespace' in Owlready2. It accepts a namespace or an ontology.

Owlready 1:

```
>>> class Drug(Thing):
...     ontology = onto
```

Owlready 2:

```
>>> class Drug(Thing):
...     namespace = onto
```

1.21.2 Generated Individual names

Owlready 1 permitted to generate dynamically Individual names, depending on their relations. This is no longer possible in Owlready 2, due to the different architecture.

1.21.3 Functional properties

In Owlready 1, functional properties had default values depending on their range. For example, if the range was float, the default value was 0.0.

In Owlready 2, functional properties always returns None if not relation has been asserted.

1.21.4 Creation of restrictions

In Owlready 1, restrictions were created by calling the property:

```
>> Property(SOME, Range_Class)
>> Property(ONLY, Range_Class)
>> Property(MIN, cardinality, Range_Class)
>> Property(MAX, cardinality, Range_Class)
>> Property(EXACTLY, cardinality, Range_Class)
>> Property(VALUE, Range_Instance)
```

In Owlready 2, they are created by calling the .some(), .only(), .min(), .max(), .exactly() and .value() methods of the Property:

```
>> Property.some(Range_Class)
>> Property.only(Range_Class)
>> Property.min(cardinality, Range_Class)
>> Property.max(cardinality, Range_Class)
>> Property.exactly(cardinality, Range_Class)
>> Property.value(Range_Individual)
```

1.21.5 Logical operators and 'One of' constructs

In Owlready 1, the negation was called 'NOT()'. In Owlready 2, the negation is now called 'Not()'.

In Owlready 1, the logical operators (Or and And) and the one_of construct expect several values as parameters.

```
>>> Or(ClassA, ClassB,...)
```

In Owlready 2, the logical operators (Or and And) and the OneOf construct expect a list of values.

```
>>> Or([ClassA, ClassB,...])
```

1.21.6 Reasoning

In Owlready 1, the reasoner was executed by calling ontology.sync_reasoner().

```
>>> onto.sync_reasoner()
```

In Owlready 2, the reasoner is executed by calling sync_reasoner(). The reasoning can involve several ontologies (all those that have been loaded). sync_reasoner() actually acts on a World (see *Worlds*).

```
>>> sync_reasoner()
```

1.21.7 Annotations

In Owlready 1, annotations were available through the ANNOTATIONS pseudo-dictionary.

```
>>> ANNOTATIONS[Drug]["label"] = "Label for Class Drug"
```

In Owlready 2, annotations are available as normal attributes.

```
>>> Drug.label = "Label for Class Drug"
```

1.22 Contact and links

A forum/mailing list is available for Owlready on Nabble: http://owlready.306.s1.nabble.com

In case of trouble, please write to the forum or contact Jean-Baptiste Lamy <jean-baptiste.lamy @ univ-paris13 . fr>

```
LIMICS
University Paris 13, Sorbonne Paris Cité
Bureau 149
74 rue Marcel Cachin
93017 BOBIGNY
FRANCE
```

Owlready on BitBucket (Git development repository): https://bitbucket.org/jibalamy/owlready2