Owlready2 Documentation

Release 0.5

Jean-Baptiste LAMY

Apr 21, 2018

Contents

Owlready2 is a module 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. Controray to version 1, Owlready2 can deal with big ontologies.

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 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.8326.n8.nabble.com

CHAPTER 1

Table of content

1.1 Introduction

Owlready2 is a module 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.

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()
```

1.1.2 Architecture

Owlready2 maintains a RDF quadstore in an optimized SQLite3 database, 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 Managing ontologies

1.2.1 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 owlready 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
```

1.2.2 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).

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.

1.2.3 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.

1.2.4 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
- any object, data or annotation property name

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.

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 entities that are related to another one with the 'has_topping' relation:

```
>>> 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)
```

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

1.2.5 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)
```

1.2.6 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 Classes and Individuals (Instances)

1.3.1 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").

1.3.2 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}
```

1.3.3 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,))
```

1.3.4 Creating equivalent classes

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

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

1.3.5 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)
```

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

1.3.6 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)
[]
```

1.3.7 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()
```

1.3.8 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.

1.3.9 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)
```

1.4 Properties

1.4.1 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).

1.4.2 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]
```

1.4.3 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
```

1.4.4 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.

1.4.5 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.

1.4.6 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.

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

The .indirect() method can be used to obtain a generator over all indirectly related entities. It takes into account:

- subproperties,
- transitive properties,
- symmetric properties,
- reflexive properties.

```
>>> with onto:
...     class BodyPart(Thing): pass
...     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(list(left_ventricular.part_of.indirect()))
[heart, abdomen]
```

1.4.8 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).

1.4.9 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.

1.5 Class constructs, restrictions and logical operators

Restrictions are special types of Classes in ontology.

1.5.1 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 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)

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

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).

1.5.2 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])
- 'I' : 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.

1.5.3 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.

1.5.4 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.

1.5.5 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)
```

1.5.6 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.

1.6 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.

1.6.1 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])
```

1.6.2 Disjoint Properties

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

1.6.3 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).

1.6.4 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.

1.6.5 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)
```

1.6.6 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.

1.6.7 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**.

1.7 Mixing Python and OWL

1.7.1 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")

>>> 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
```

1.7.2 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).

1.7.3 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':

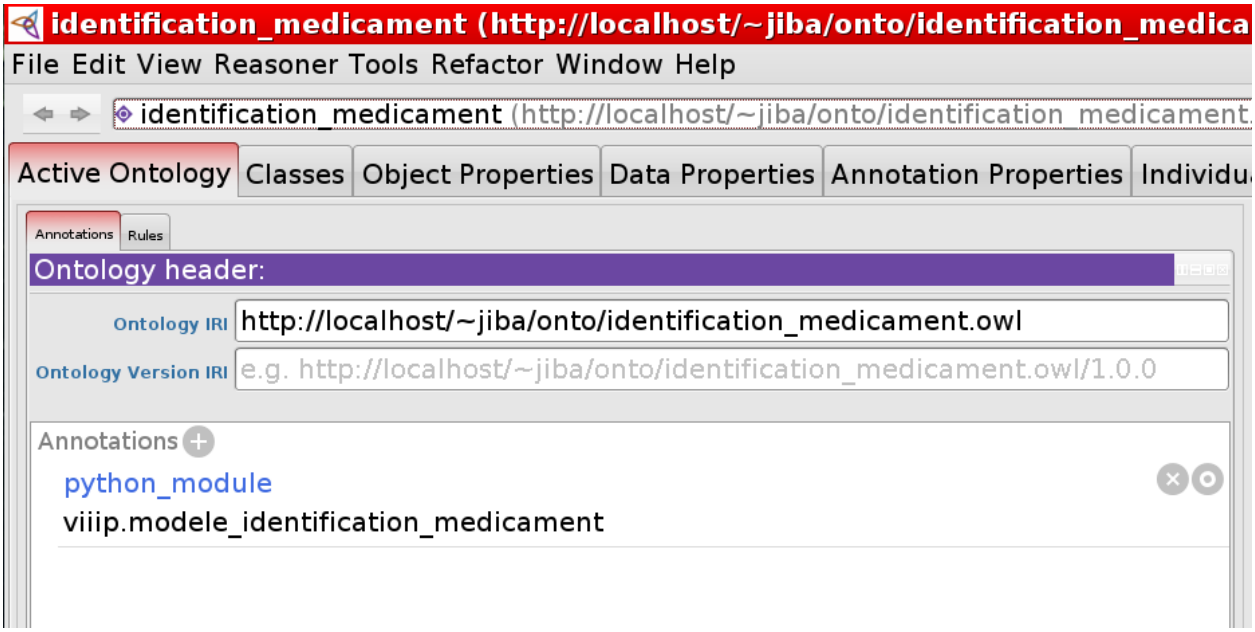
```
>>> from owlready2 import *

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

>>> 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!

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

1.8.1 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"
```

1.8.2 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)
```

1.8.3 Running the reasoner

The reasoner 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).

1.8.4 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.

1.9 Annotations

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

1.9.1 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 AnnotationProperty (here comment) as a pseudo-dictionary:

```
>>> 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)

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

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

```
>>> comment[Drug, owl_subclassof, Thing] = "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"
```

1.9.2 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(comment[acetaminophen, is_prescribed_for, pain])
['A comment on the acetaminophen-pain relation']

>>> print(comment[Drug, owl_subclassof, Thing])
['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'
```

1.9.3 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 = []
```

1.9.4 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

>>> 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
```

1.9.5 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")
```

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.

1.9.6 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"
```

1.10 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#').

1.10.1 Accessing entities defined in another namespace

These entities can be accessed in Owlready2 using a namespace. The .get_namepace(base_iri) method of an ontology 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 = go.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']
```

1.10.2 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
```

1.10.3 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)
```

1.11 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.

1.11.1 Persistent world: storing the quadstore in an SQLite3 file

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 ontogolies: 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).

1.11.2 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)
```

1.11.3 Working with RDFlib for performing SPARQL queries

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 .
}"""))
```

1.12 Differences between Owlready version 1 and 2

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

1.12.1 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
```

1.12.2 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.

1.12.3 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.

1.12.4 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)
```

1.12.5 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,...])
```

1.12.6 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()
```

1.12.7 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"
```

1.13 Contact and links

A forum/mailing list is available for Owlready on Nabble: http://owlready.8326.n8.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 (development repository): https://bitbucket.org/jibalamy/owlready