创建一个全新空白本体：

from owlready2 import \*

‘’‘get\_ontology()方法，传入一个iri作为本体的标识，iri可以是不真实存在的，如果此iri是一个已存在的本体的标识，则会返回该本体；有些本体使用#分割本体名和本体内的实体名，有些使用/，owlready2默认使用#。如果想使用/则iri应该以/结尾

‘’’

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

从已有的OWL文件中加载本体

from owlready2 import \*

‘’‘仍然使用get\_ontology()方法，但传入的iri之前要加上file://，这样此方法就认为是从已有文件中加载本体

‘’’

path = 'file://C:/Users/朗语/Desktop/语义检索/数据/CFNOnto1.owl'

CFNOnto = get\_ontology(path).load()

‘’’

.call()方法从本地或者互联网中加载一个本体的副本，如果调用多次call()方法则只会加载一次

‘’’

访问本体内容：

你可以用.运算符来访问本体内容。这种方式支持访问本体中定义的Class、Instances、Properties、Annotation Properties。[]的访问方式也支持。

1. 本体类属性：
2. .base\_iri：本体的标识iri
3. imported\_ontologies：已导入的本体列表
4. 本体类方法：
5. .classes()：返回一个本体内定义的Class的生成器
6. .individuals()：返回一个本体内定义的个体（或者实例）的生成器
7. .object\_properties() : returns a generator for ObjectProperties defined in the ontology
8. .data\_properties() : returns a generator for DataProperties defined in the ontology
9. .annotation\_properties() : returns a generator for AnnotationProperties defined in the ontology
10. .properties() : returns a generator for all Properties (object-, data- and annotation-) defined in the ontology
11. .disjoint\_classes() : returns a generator for AllDisjoint constructs for Classes defined in the ontology
12. .disjoint\_properties() : returns a generator for AllDisjoint constructs for Properties defined in the ontology
13. .disjoints() : returns a generator for AllDisjoint constructs (for Classes and Properties) defined in the ontology
14. .different\_individuals() : returns a generator for AllDifferent constructs for individuals defined in the ontology
15. .get\_namepace(base\_iri) : returns a namespace for the ontology and the given base IRI

使用IRIS伪字典可以根据full IRI访问本体的实体（必须先加载本体）

from owlready2 import \*

path = 'file://C:/Users/朗语/Desktop/语义检索/数据/CFNOnto1.owl'

CFNOnto = get\_ontology(path).load() ‘’’

print(IRIS['http://www.semanticweb.org/apple/ontologies/2015/10/untitled-ontology-38#框架'])

----------------------------------------------------------------------------------------------

CFNOnto1.框架

----------------------------------------------------------------------------------------------

简单的查询

可以用.search()方法进行简单的查询，传入一些关键值参数指定查询要求

* **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

导入其他本体：

如同一个python文件可以导入其他python模型一样，也可以向一个本体中导入其他本体，例如：CFNOnto.imported\_ontologies.append(other\_onto)

将本体保存到文件：

可以用.save()方法保存本体，默认的无参数方法将会把本体保存到onto\_path中的第一条路径中；.save也可接受两个参数来指定保存到的文件和保存格式，默认格式为RDF/XML

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

类和类实例

1. 创建类

通过继承owlready2.Thing类来在本体中创建类；本体类属性(attribute)可以被用于将你的类和指定的本体相连接，如果创建类时没有指定属性，则属性将从父类中继承。

owlready2中定义的Thing类中的namespace属性可以用来创建本体类的full IRI

from owlready2 import \*

onto = get\_ontology("http://test.org/onto.owl")

class Drug(Thing):

namespace = onto

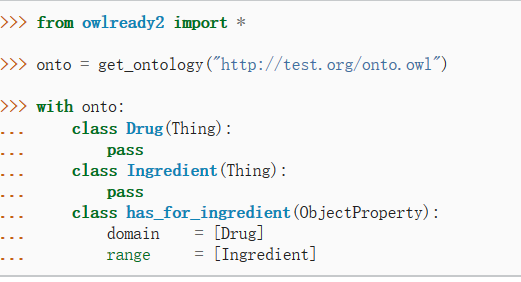
#或者可以这样写，用with关键字加namespace的值

with onto:

class Drug(Thing):

pass

#两者效果相同



可以用with在同一namespace下创建多个类

Thing类中定义的iri属性保存有类的full IRI

print(Drug.iri)

------------------------------------------------------------------------------

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

------------------------------------------------------------------------------

可以通过直接修改类的iri属性来更改类的iri

1. 创建和管理子类

通过继承一个已有的本体类来创建子类，在上例中Drug就是Thing的子类，而Thing是所有本体类的父类。

.is\_a属性保存有当前类的一个父类列表，也可以通过修改列表的内容来增加或移除一些父类

.descendants()方法返回当前类的所有子类的集合；.ancestors()方法返回当前类的所有父类集合；两种方法都包括当前类自身，但是不包括非实体类（python里的集合是无序的，所以集合里的元素顺序和类的等级无关）

1. 动态地创建类

利用python的types模块可以动态地创建类

import types

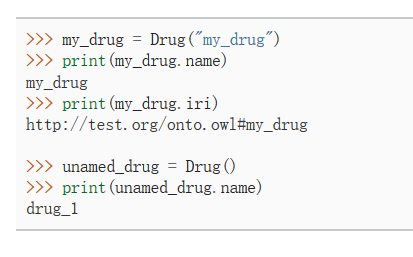
NewClass = types.new\_class("NewClassName", (SuperClass,), kwds = { "namespace" : my\_ontology })

1. 创建相等的类

.equivalent\_to属性保存当前类的所有相等类的列表，可以通过修改此属性的值来增加或删除相等类

1. 创建类实例

创建类实例的方法和在Python中创建类实例相同



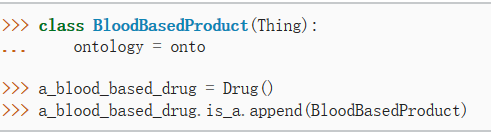
也可以在创建类实例时指定多个额外的关键词参数，如namespace值和各种本体中的类属性值。



创建完成的类实例可以立即访问；类名.instances()方法会返回一个生成器，可以用于遍历或者创建一个包含该类所有实例的列表；实例也包含有.equivalent\_to()方法

1. 属于多个类的实例

Owlready2中也支持实例属于多个类的情况，类实例有.is\_a属性，此属性和Class.is\_a相似。创建属于多个类的实例时，要先创建单类实例，然后通过向.is\_a中添加类名来增加它所从属的类。



Owlready2会自动创建一个隐藏的类，这个类继承自Drug以及BloodBasedProdut。这个隐藏类在a\_blood\_based\_drug.\_\_class\_\_中可见，但在a\_blood\_based\_drug.is\_a中不可见

1. 销毁实体

destroy\_entity()这个全局函数可以被用来销毁一个实体。它将实体从本体以及quad store中移除。Owlreaddy2和Protege4在销毁实体时的处理相似：将实体所有的相关的关系也一并移除，所有相关的类、constructs以及blank nodes也一并移除

属性

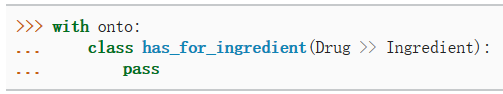
1. 创建属性

一个新的属性可以用继承一个ObjectProperty类或者DataProperty类来创建（两个都是Owlready2中的类）。这两个类中的domain和range属性可以用来指定本体中属性的domain和range，这两个类属性为列表类型，因为OWL允许为本体属性指定多个域，但是最终的domain或者range为列表中各元素的交集。

下例在本体中建立了两个类，Drug和Ingredient以及一个关联两者的ObjectProperty属性



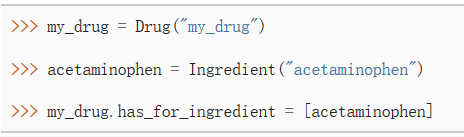
也可以用以下的方法创建ObjectProperty，作用同上，其中‘domain >> range’将取代ObjectProperty这个父类



1. 创建一个关系

在本体中，关系是一个三元组（主语、谓语、宾语），其中本体的属性用作谓语，类实例或者文字类型作为主语和宾语，具体谁做主语谁做宾语会在属性的domain和range中定义。

使用主语的Python属性（attribute）可以获取或者创建一个关系，属性（attribute）名称与本体属性类（Property）名称相同

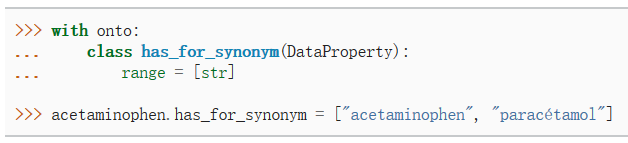


可以使用instance\_1\_name.property\_name.append(instance\_2\_name)或者instance\_1\_name.property\_name.remove(instance\_2\_name)方法来增加或者删除关系的宾语。

1. 数据属性

数据属性是本体中属性的range为特定数据类型的属性，Owlready2目前支持以下数据类型：

* 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



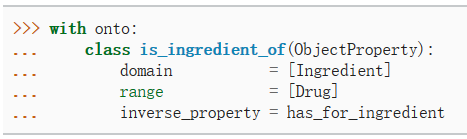
上例创建了一个has\_for\_synonym的数据属性，它的range为str类型，并且为acetaminophen这一实例创建了has\_for\_synonym关系；效果相同的数据属性也可以如下图所示创建：



1. 反转属性（Inverse Properties）

Inverse Properties的定义：如果两个属性以相反的方式（domain和range相反）表达同样的意思，则这两个属性互为相反属性。如is\_ingredient\_of和has\_for\_ingredient互为相反属性。就像：“a drug A has for ingredient B” 与“B is ingredient of drug A”含义相同。

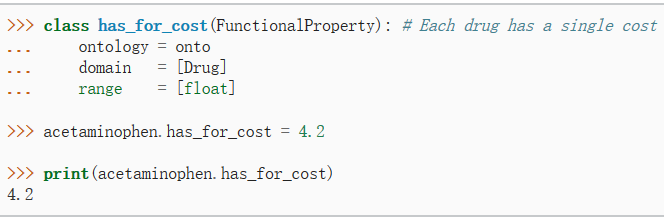
在Owlready2中定义相反属性需要先定义属性的domain和range然后在属性类的inverse\_property的属性（attribute）中指定与它相反的属性名。



Owlready2会自动处理反转属性，如果设置A is\_ingredient\_of B则会自动设置B has\_for\_ingredient A。但是这只会在声明反转属性之后生效，例如在先创建A类、B类以及属性has\_for\_ingredient，再设置A.has\_for\_ingredient.append(B)，最后才创建反转属性is\_ingredient\_of，则B.is\_ingredient\_of中不会有A

1. 函数型属性和反转的函数型属性

函数型属性是一种属性值为特定数据类型的属性，继承FunctionalProperty类来创建函数型属性。



反转的函数型属性的反转属性是函数型属性，可以通过继承InverseFunctionalProperty创建。

1. 创建子属性

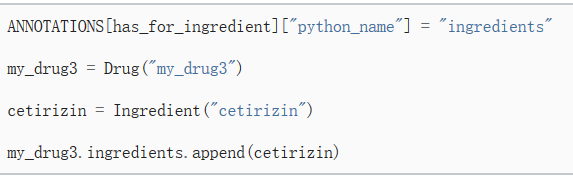
通过继承已经创建的属性（或类）来创建子属性（或子类）。

Owlready2暂时不支持在子属性被定义后自动更新父属性，如果需要这个功能，使用get\_relations()全局函数。



1. 将Python别称（alias）关联到属性

在本体中属性名一般都很长，例如has\_for\_ingredient，但是在Python中短的属性名更常见一些，如ingredients。Owlready2允许使用ANNOTATIONS二维列表来为属性名重命名。



注意，此时在属性类里，被声明别名的属性仍然只能使用原名。例如如果要创建has\_for\_ingredient的子属性，应该是 class sub\_prop(has\_for\_ingredient)，写成 class sub\_prop(ingredients)无效。



约束（是一种特殊的本体类）

1. 属性约束

例如，Placebo是一个没有活性成分（Active Principle）的药物





在上例中，has\_for\_active\_principle.some(ActivePrinciple)返回一个至少有一种ActivePrinciple的类。NOT()函数返回一个类的补集，&运算符求两个类的交集。所以这行代码定义了一个Placebo类，并且声明它与Drug类和没有任何ActivePrinciple的类的交集类等价。

Owlready2现在支持以下的约束形式：

* 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)



上例说明DrugAssociation是一种有至少两种ActivePrinciple的Drug。

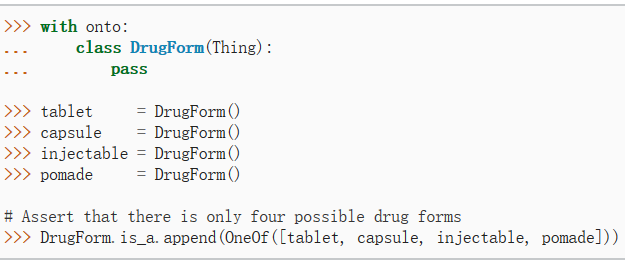
1. 逻辑运算符

Owlready2提供了以下的逻辑运算符，用于类之间的计算：

* ‘&’ : and operator (intersection). For example: Class1 & Class2
* ‘|’ : or operator (union). For example: Class1 | Class2
* Not() : not operator (negation or complement). For example: Not(Class1)

1. One Of构造器

在本体中，One Of语句被用来通过扩展来定义类，例如通过列举出类的实例而不是定义它的属性来定义类。



1. Inverse-of构造器

Inverse-of构造器返回一个属性的反转，不必新建一个属性

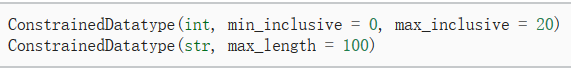


1. 有约束的数据类型

一个被约束的数据类型是一种值被限制的数据，例如整数被限制在0到20。

全局函数ConstrainedDataType()从一个基础数据类型中创建一个约束数据类型，它有如下可选参数：

* length
* min\_length
* max\_length
* pattern
* white\_space
* max\_inclusive
* max\_exclusive
* min\_inclusive
* min\_exclusive
* total\_digits
* fraction\_digits



1. 属性链

属性链支持链接两个属性，使用PropertyChain()函数传入一个包含两个要被链接的属性的列表：PropertyChain([prop2, prop2])

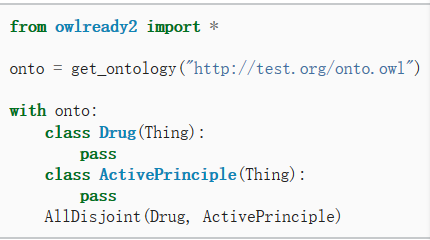
不相交、开放以及本地限制的推理（Disjointness, open and local closed world reasoning）

OWL默认世界是开放的，例如所有在本体中生命的东西都被认为不是错的，而是可能的（这种说法一般叫做开放世界假设）。因此如果想要让事物或事实是错的，就一定要在本体中明显地声明出来。

1. 不相交的(disjoint)类

如果没有一个个体同时属于这几个类，那就认为这几个类不相交。

类的不相交性（disjointness）是用AllDisjoint()来创建，它接收一个类的列表作为参数。在下例中，有两个类Drug和ActivePrinciple，我们可以声明他们是不相交的（根据开放世界假设，我们必须显性地将他们不相交的这一事实声明出来）。

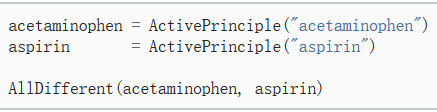


1. 不相交的属性

使用AllDisjoint()也可声明不相交的属性

1. 不同的个体

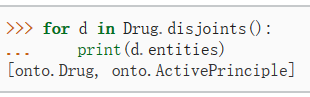
在OWL中，两个个体有可能会被认为是相同的个体，除非他们被显性地声明他俩不同。不同性之于个体就像不相交之于类。下例创建了两个ActivePrinciple实例并且声明他俩不同。



在Owlready2中AllDifferent()函数和AllDisjoint()函数事实上是一样的，不一样的仅仅是参数。如果传来了类，那它就是AllDisjoint()，如果传来实例那就是AllDifferent()。

1. 查询和修改不相交

.disjoint()方法返回一个生成器（Python 里的generator）以遍历某个类、实例、属性的所有不相交的对象。

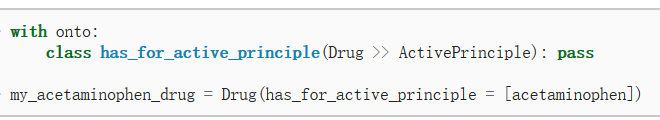


上例中的.entities属性可以修改，通过修改这个属性就可以增加或删除不相交的对象。

OWL也提供了‘disjointWith’ 和 ‘propertyDisjointWith’关系来设定只涉及两个元素的不相交。Owlready2

1. 关闭个体

开放世界假设同样意味着一个给定个体的属性并不会限制于被声明出来的那些。例如，你创建了一个有单个ActivePrinciple的Drug实例，但并不意味着它只有单个ActivePrinciple。



在上例中，‘my\_acetaminophen\_drug’是一个有一个ActivePrinciple的药物，但是它可能还有别的ActivePrinciple。

如果你想让‘my\_acetaminophen\_drug’是一个只有acetaminophen作为ActivePrinciple的药物，你必须显性地声明它。



在约束这一章中，我们知道Property.only(Range\_Class)这个函数只接受类域作为参数，所以我们使用OneOf()这一函数来将acetaminophen这个个体转换为类。

你也许已经注意到，开放世界假设这一原则经常会导致乏味的一长列的AllDifference或者其他约束。幸运的是，Owlready2提供了close\_world()这一函数以自动地‘关闭’一个实例。close\_world()会自动地添加ONLY约束，它接收一个可选参数：一个供‘关闭’的属性列表（传入一个实例则会关闭以该实例为domain的所有属性）。



1. 关闭类

close\_world()也接受一个类，在这种情况下，它会关闭这个类以及它的子类和个体。

默认地，当close\_world()没有被调用时，本体实施开放世界推理。通过选择你想关闭的类和个体，close\_world()函数允许OWL实施本地关闭世界推理（loal closed world reasoning）。

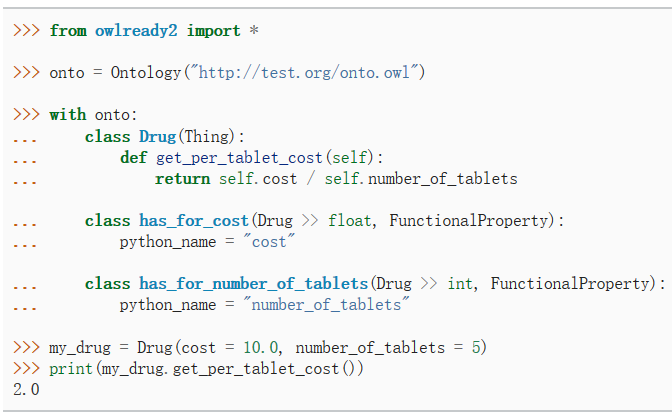
1. 关闭本体

最后，close\_world()也可接受一个本体。在这种情况下，它关闭本体中定义的所有的类。这相当于全关闭世界推理。

混合Python与OWL

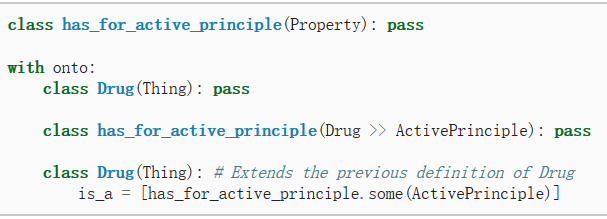
1. 向OWL类中添加Python函数

Python函数可以被定义在本体类中，在下面的例子里Drug类有一个Python函数以计算per-tablet的成本，使用两种OWL属性（这两种属性已经在Python中被重命名，见 将Python别称关联到属性 这一节）。



1. 向前声明

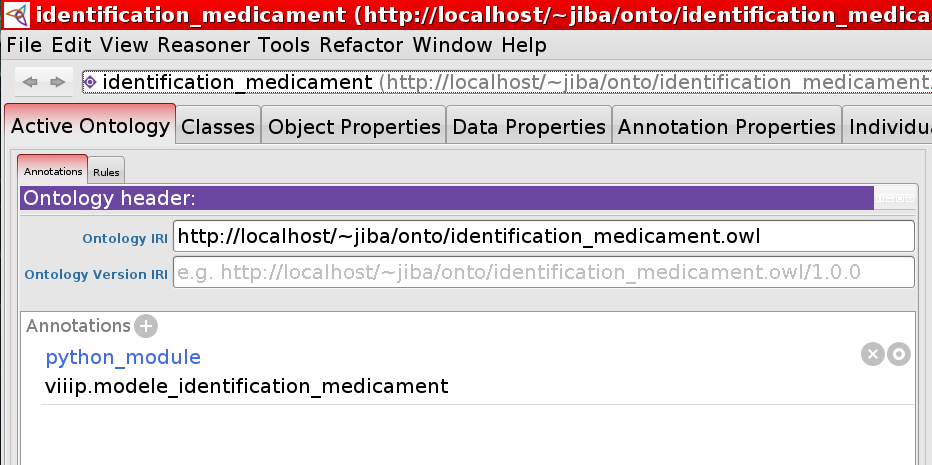
有时你也许需要向前声明一个类或者属性。如果一个类或者属性（名称相同命名空间也相同）被定义，新的定义将会继承旧的定义，但不会替代它。



前一个定义创建了Drug类，后一个定义在前者基础上增加了一个is\_a声明。

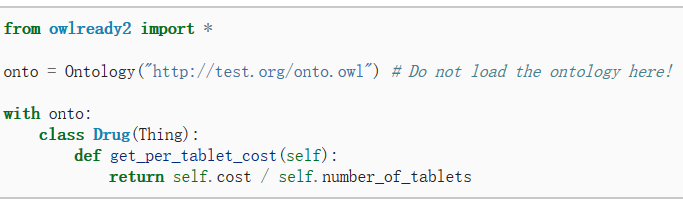
1. 关联Python模块与OWL本体

关联一个Python模块和一个本体是可行的。当Owlready2加载本体时，它会自动地引入此Python模块。这可以通过‘python\_module’注释来完成，这需要在本体上设置。它的值必须是你要关联的Python模块的名字，具体的设置可以在protégé上完成。在导入本体之后：



Python模块可以包括类和属性的定义以及方法。然而它并不需要包含所有的is\_a关系、定义域（domain）、值域（range）等，因为这些关系已经在owl定义过，不需要再在Python中重新指定。因此，如果你的类等级树在owl中被定义，你可以创建所有的类为Thing 的子类。

例如，在文件‘my\_python\_module.py’中：



然后在‘onto.owl’中，你可以定义：

* ‘python\_module’ 注释(值: ‘my\_python\_module’)
* 需要时，定义‘Drug’的超类
* ‘has\_for\_cost’ 属性(在Python中省略了，因为它没有方法)
* The ‘has\_for\_number\_of\_tablets’ 属性(也被省略了)

推理

OWL推理器可以用来检查本体的一致性，然后依据本体内容推理出新的事实，一般是推断个体从属的类，类从属的父类。

OWL有多种推理器，Owlready2包含一个修改过的HermiT reasoner，由斯坦福大学计算机科学部开发，在LGPL证书下发布。HermiT是用Java开发的，因此你的电脑需要装有Java虚拟机以在Owlready2中执行推理。

1. 建立所需的一切

开始推理之前，你要先创建所有的类、属性以及实例，并且确保约束、不相交性（disjointness）、不同一性（differences）都已被定义。下面是一个创建满足推理条件的本体的例子。

from owlready2 import \*

onto = 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])

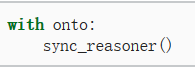
drug3 = Drug(active\_principles = [])

close\_world(Drug)

1. 启动推理器

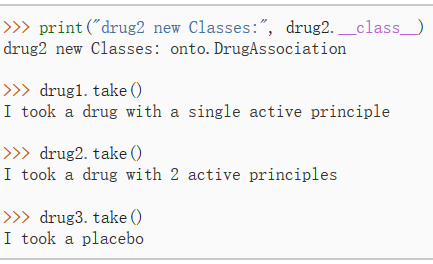
启动推理器仅需要调用这个全局函数：sync\_reasoner()。

sync\_reasoner()默认将所有推断出来的事实放入一个特殊的本体中，但你可以通过 with ontology\_to\_save: 这个语句方式来控制事实存入哪个本体（要记得，所有的在with ontology\_to\_save:之内声明的三元组都存在于ontology\_to\_save这个本体）。例如，将所有推理结果存入onto本体：



1. 自动分类的结果

Owlready自动地从HermiT获取推理的结果并且对个体和类进行重分类，例如Owlready更改个体的类以及更改类的超类。



上图是上一节中给出的本体在推理过后的重新分类结果，可以看到，drug1、2、3刚开始定都被定义为Drug类，在进行推理之后，依据它们has\_for\_active\_principle的属性的不同推理器给它们重新确定了所属的类别。

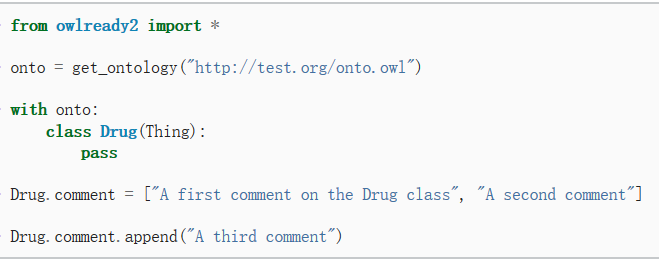
请读者注意Owlready2如何将本体类的自动分类与Python类的多态性结合的。

注释（Annotation）

在Owlready2中注释可以像Python类属性（attribute）一样被访问。需要注意的是注释不可被继承。

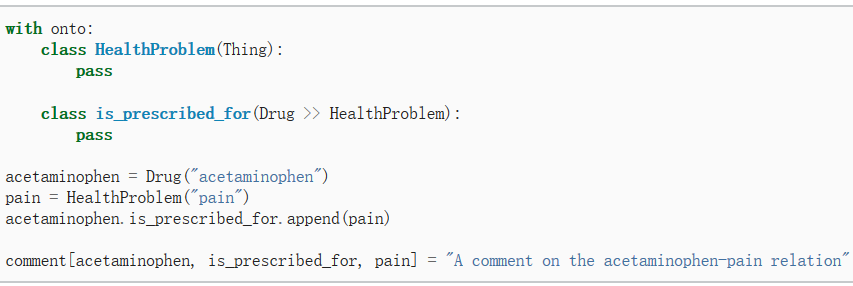
1. 添加注释

对于一个给定的实体（类、属性或者实例），可以用以下方法添加注释：



可以看到，注释是以列表的形式出现的，默认有以下几种方式添加注释：comment、isDefinedBy、seeAlso、backwardCompatibleWith、deprecated、incompatibleWith、priorVersion、versionInfo。

Owlready2同时支持给关系三元组添加注释，使用AnnotationProperty（在这里用的是comment）以伪字典的方式添加注释（字典的key分别是定义域类实例、关系名、值域类实例，字典的value是评论文本）。



Owlready2提供特殊的伪属性以注释is-a关系（包括owl\_subclassof和rdf\_type）、定义域（domains，如rdf\_domain）以及定义域（ranges，如rdf\_range）。

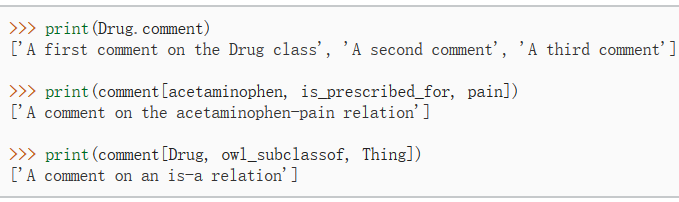


注释值通常是值列表。然而在很多情况下都会使用单个值。Owlready2允许给注释属性设置单个值。

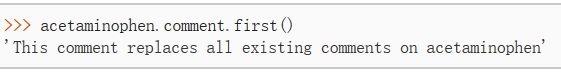


1. 查询注释

注释值可以使用.运算符来获取就如同它们是实体的属性：



如果你希望查询结果是单个值，可以使用list类型的.first()函数，它返回列表中的首个值，如果列表为空则返回None。



1. 删除注释

只需从注释列表中将特定注释移除即可：

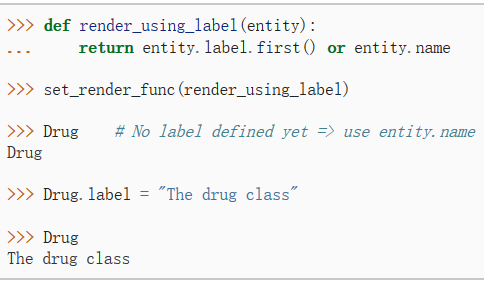
Drug.comment.remove("A second comment")。

或者直接给注释列表赋值一个空列表以删除全部注释：Drug.comment = []。

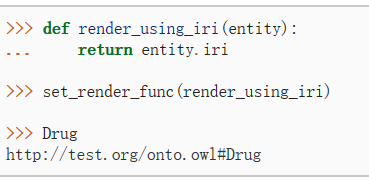
1. 定制化的实体呈递

全局函数set\_render\_func()可以指定Owlready2如何呈递实体。例如，当打印实体时如何去将他们转换成文本。这个函数接收一个函数作为参数，做参数的函数的参数是实体，返回值是字符串。

label注释经常被用来呈递实体，下面的例子使用它们的label注释呈递实体，如果label为空则使用实体名。

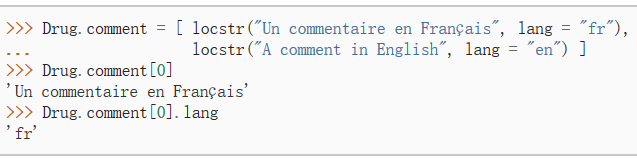


下面的例子使用实体的iri呈递实体：

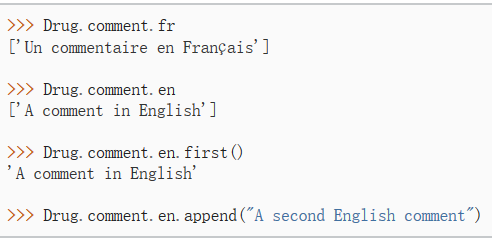


1. 特定语言的注释

可以使用locstr类型来指定注释的语言：



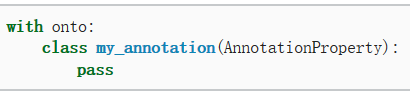
另外，值列表支持指定语言的子列表，子列表形式为‘.<语言代码>’如‘.en’、‘.fr’等。这些子列表包含普通的字符串，可以被修改。



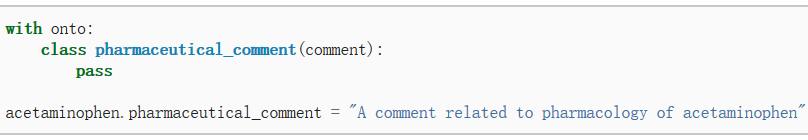
修改指定语言的子列表会自动更新值列表，但是反过来就不会：修改值列表不会更新指定语言子列表。

1. 创建新的注释类

AnnotationProperty类可以生成子类以创建新的注释类：



你也可以为已存在的注释类创建子类。



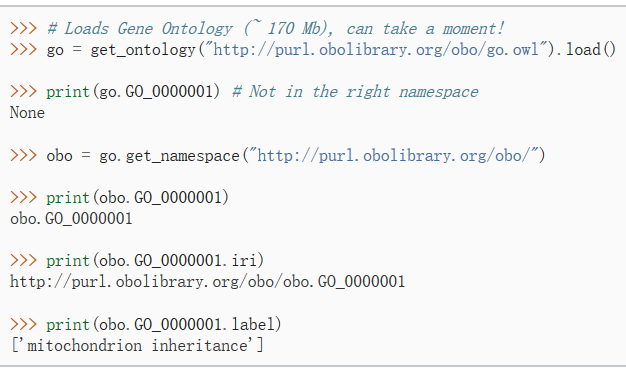
命名空间

本体可以定义位于其他命名空间的实体。一个例子是Gene本体：本体的IRI是‘<http://purl.obolibrary.org/obo/go.owl>’，但是本体的实体的IRI并不是如此形式‘<http://purl.obolibrary.org/obo/go.owl#GO_entity>’，而是‘<http://purl.obolibrary.org/obo/GO_entity>’。

1. 访问定义在其他命名空间的实体

实体在Owlready2中可以使用命名空间来访问。本体的.get\_namespace(base\_iri)方法返回一个base\_iri指定的命名空间。

命名空间之后可以使用‘.’来进行使用，和本体类似。



1. 在指定命名空间中创建类

当创建一个类或者属性的时候，命名空间属性可以用来建立类的full iri并且定义在哪个本体中创建这个类。类的iri与命名空间的base iri加上类名相同。

一个本体永远可以被当成命名空间使用，见‘类和类实例’这一章。如果你想在不同的iri中建立类你可以使用命名空间对象：



在上面的例子中，Drug类的iri是‘http://test.org/pharmaco/Drug’，但是类属于‘<http://test.org/onto>’这个本体。

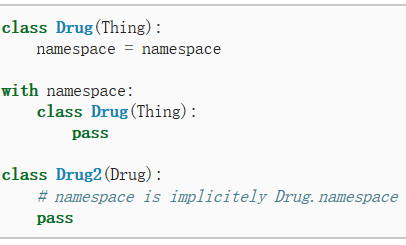
Owlready2拥有三种显示命名空间的方法：

（1）namespace类属性（attribute）；

（2）with namespace:语句；

（3）如果不提供，则命名空间从第一个父类处继承。

下面的例子阐明了以上三种方法：

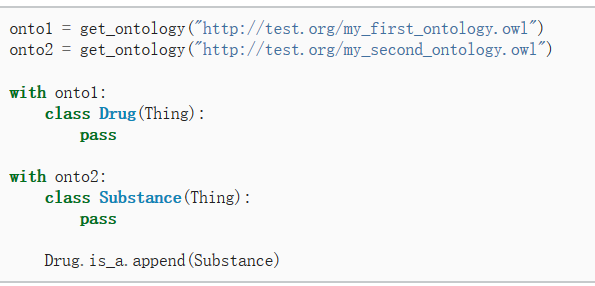


1. 更改定义在其他本体中的类

在OWL中一个本体可以修改一个在其他本体中定义的类。

在Owlready2中，以上特性可以使用‘with namespace:’语句实现。每一个被创建（删除）在‘with namespace:’语句之内的RDF三元组都存在于那个语句指定的namespace之内。

下面的例子在第一个本体中创建Drug类，并且声明Drug是第二个本体中的Substance的实例。



Worlds

Owlready2将所有的三元组存储在‘World’对象中，它也可以同时处理多个Worlds。‘default\_world’是默认使用的World。

1. 持久的world：将quadstore存储在SQLite3文件中

Owlready2使用一个优化的quadstore。这个quadstore默认存储在内存中，但是它也可以被存储在SQLite3文件中。这就保证了持久化：一个被加载的或创建的本体存储在文件中以供以后使用。这对于大型本体来说很有趣：加载一个大型本体需要时间，但是打开一个SQLite3文件只需要几分之一秒（即使很大）。并且如果你仅仅需要访问几个本体的实体的话，应当避免在内存中加载巨型本体。

World的.set\_backend()函数将SQLite3文件名关联到quadstore：

default\_world.set\_backend(filename = "/path/to/your/file.sqlite3")

如果quadstore非空，当调用.set\_backend()函数时RDF三元组会被自动地复制。然而这个操作会有很高的性能消耗（尤其是三元组很多时）

当时用持久化时必须调用World的.save()函数以保存SQLite3文件中的quadstore的实际状态。

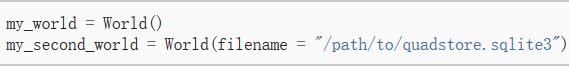


在文件中保存quadstore不会损耗Owlready2的性能（事实上，将quadstore保存在硬盘上Owlready2允许地会略快一些）

1. 使用多个孤立的Worlds

Owlready2支持多个、孤立的Worlds。如果你想要加载同一个本体的多个版本，例如本体在进行推理前后的版本，会十分有趣。

你可以使用World类来创建新的World：



本体在这之后就可以使用World的.get\_ontology()方法创建和加载本体（当在多个Worlds中使用时，这个方法会替代全局的get\_ontology()函数）：

onto = my\_world.get\_ontology("http://test.org/onto/").load()

World对象可以像伪字典一样使用以通过iri来访问实体（当在多个Worlds中使用时，这个方法会替代全局的IRIS伪字典）：

my\_world["http://test.org/onto/my\_iri"]

最后，推理器可以在指定的World中运行：

sync\_reasoner(my\_world)

1. 使用RDFlib执行SPARQL查询

Owlready2使用一个优化的RDF quadstore。它也可以被当做RDFlib图来访问：

graph = default\_world.as\_rdflib\_graph()

RDFlib图也可以被用来执行SPARQL查询：

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 .}"""))