



Chapitre 10

Mixer Python et OWL

1. Introduction

Dans ce chapitre, nous verrons comment mélanger méthodes Python et constructeurs logiques OWL au sein d'une même classe.

2. Ajouter des méthodes Python aux classes OWL

Avec Owlready, les classes OWL sont des classes Python (presque) comme les autres. Il est donc possible d'y inclure des méthodes. Voici un exemple simple qui permet de calculer le prix par comprimé d'un médicament à partir de son prix unitaire (par boîte) et du nombre de comprimés dans la boîte :

```
>>> from owlready2 import *
>>> onto = get_ontology("http://test.org/medicament.owl#")
>>> with onto:
...     class Médicament(Thing): pass
...
...     class prix (Médicament >> float, FunctionalProperty): pass
...     class nb_comprimé(Médicament >> int , FunctionalProperty): pass
...
...     class Médicament(Thing):
...         def get_prix_par_comprimé(self):
...             return self.prix / self.nb_comprimé
```

Notez que la classe `Médicament` est définie deux fois : il s'agit d'une déclaration anticipée pour pouvoir l'utiliser dans les définitions des propriétés (voir chapitre Créer et modifier des ontologies en Python, section Définitions multiples et déclarations anticipées).

La méthode peut ensuite être appelée sur les individus de la classe :

```
>>> mon_médicament = Médicament(prix = 10.0, nb_comprimé = 5)
>>> mon_médicament.get_prix_par_comprimé()
2.0
```

Dans les ontologies, il est fréquent de n'utiliser que des classes et des sous-classes, en lieu et place des individus (c'est le cas de Gene Ontology par exemple), car le pouvoir de représentation des classes est supérieur. Dans ce cas, Python permet de définir des méthodes de classe qui seront appelées sur la classe (ou une de ses sous-classes) et qui prend en premier paramètre cette dernière.

Voici le même exemple que précédemment en utilisant des classes :

```
>>> with onto:
...     class Médicament(Thing): pass
...
...     class prix (Médicament >> float, FunctionalProperty): pass
...     class nb_comprimé(Médicament >> int , FunctionalProperty): pass
...
...     class Médicament(Thing):
...         @classmethod
...         def get_prix_par_comprimé(self):
...             return self.prix / self.nb_comprimé
```

La méthode peut ensuite être appelée sur la classe et ses sous-classes :

```
>>> class MonMédicament(Médicament): pass
>>> MonMédicament.prix = 10.0
>>> MonMédicament.nb_comprimé = 5
>>> MonMédicament.get_prix_par_comprimé()
2.0
```

Attention cependant, pour faire cohabiter les deux types de méthode (d'individu et de classe) ensemble, il est nécessaire d'utiliser des noms de méthodes différents.

3. Associer un module Python à une ontologie

Lorsque les ontologies ne sont pas créées entièrement en Python (comme dans l'exemple ci-dessus) mais chargées à partir de fichier OWL, les méthodes Python peuvent être définies dans un fichier Python `.py` séparé. Celui-ci peut être importé manuellement ou bien relié à l'ontologie via une annotation; Owlready importera alors automatiquement le module Python lorsque l'ontologie sera chargée.

Par exemple, le fichier `bacterie.py` suivant ajoute une méthode dans les classes Bactérie et Staphylocoque de l'ontologie des bactéries :

```
# Fichier bacterie.py
from owlready2 import *

onto = get_ontology("http://lesfleursdunormal.fr/static/ \
                    _downloads/bacterie.owl#")

with onto:
    class Bactérie(Thing):
        def méthode(self):
            print("C'est une bactérie !")

    class Staphylocoque(Thing):
        def méthode(self):
            print("C'est un staphylocoque !")
```

Notez que nous n'avons pas chargé l'ontologie des bactéries (avec `.load()`) car celle-ci sera chargée par le programme principal. Notez aussi que nous n'avons pas indiqué la superclasse de Staphylocoque (qui est Bactérie). En effet, celle-ci figure déjà dans le fichier OWL, ce n'est donc pas la peine de la renseigner une seconde fois ici! En revanche, il est nécessaire de mentionner `Thing` comme parent, pour que la nouvelle classe soit bien une classe OWL gérée par Owlready et non une classe Python usuelle. D'une manière générale, lorsque l'on crée un fichier Python séparé contenant les méthodes, il est préférable de n'y placer que ces dernières, et de conserver le reste de l'ontologie (superclasses, propriétés, relations, etc.) uniquement en OWL pour limiter les redondances.

3.1 Import manuel

Nous pouvons ensuite charger l'ontologie et importer manuellement le fichier bactérie.py :

```
>>> from owlready2 import *
>>> onto = get_ontology("bacterie.owl").load()
>>> import bacterie
```

Ensuite, nous créons un Staphylocoque et nous appelons notre méthode :

```
>>> ma_bacterie = onto.Staphylocoque()
>>> ma_bacterie.méthode()
C'est un staphylocoque !
```

3.2 Import automatique

Pour cela, il est nécessaire d'éditer l'ontologie avec Protégé et d'ajouter une annotation indiquant le nom du module Python associé. Celle-ci s'appelle `python_module` et est définie dans l'ontologie `owlready_ontology.owl` qu'il est nécessaire d'importer. Voici donc les différentes étapes :

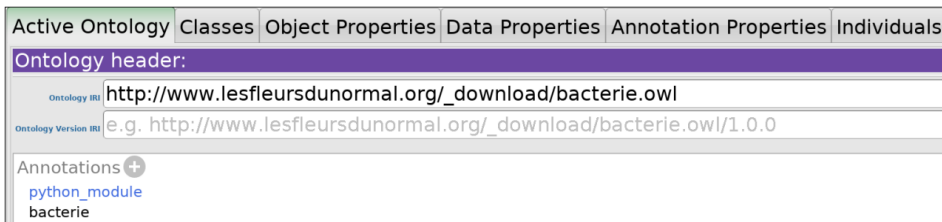


Figure 10.1 - Annotation `python_module` dans Protégé.

- ▣ Lancez Protégé et chargez l'ontologie des bactéries.
- ▣ Allez à l'onglet **Active Ontology** de Protégé.

■ Importez l'ontologie `owlready_ontology` en cliquant sur le bouton **+** à droite de **Direct imports**. L'ontologie peut être importée à partir de la copie locale qui figure dans le répertoire d'installation d'Owlready ou bien à partir de son IRI :

```
http://www.lesfleursdunormal.fr/static/_downloads/  
owlready_ontology.owl
```

■ Ajoutez une annotation dans la rubrique **Ontology header**. Le type d'annotation est `python_module` et la valeur est le nom du module, ici `bacterie` (voir Figure 10.1).

Désormais, nous n'avons plus besoin d'importer le module `bactérie`, Owlready le fait automatiquement à chaque fois que l'ontologie des bactéries est chargée. Dans l'exemple suivant, nous avons enregistré l'ontologie des bactéries (avec l'annotation `python_module`) dans un nouveau fichier OWL appelé `bacterie_owl_python.owl` :

```
>>> from owlready2 import *  
>>> onto = get_ontology("bacterie_owl_python.owl").load()  
>>> ma_bacterie = onto.Staphylocoque()  
>>> ma_bacterie.méthode()  
C'est un staphylocoque !
```

4. Polymorphisme sur inférence de type

Nous avons vu au chapitre Raisonnement automatique, section Raisonnement en monde ouvert que, lors du raisonnement, les classes des individus et les superclasses des classes pouvaient être modifiées. Dans ce cas, les méthodes disponibles peuvent changer. De plus, en cas de polymorphisme, c'est-à-dire lorsque plusieurs classes exploitent la même méthode de manière différente, l'implémentation de la méthode pour un individu ou une classe peut changer. C'est le « polymorphisme sur inférence de type ».

Voici un exemple simple :

```
>>> ma_bacterie = onto.Bactérie(gram_positif = True,  
...     a_pour_forme = onto.Ronde(),  
...     a_pour_regroupement = [onto.EnAmas()])  
>>> ma_bacterie.méthode()  
C'est une bactérie !
```

Nous avons créé une bactérie. Lorsque nous exécutons la méthode, c'est l'implémentation de la classe Bactérie qui est donc appelée. Nous allons maintenant appeler le raisonneur.

```
■ >>> sync_reasoner()
```

Le raisonneur a déduit que la bactérie est en fait un Staphylocoque (à partir de ses relations). À présent, si nous appelons la méthode, c'est l'implémentation du Staphylocoque qui est appelée :

```
■ >>> ma_bacterie.méthode()
    C'est un staphylocoque !
```

5. Introspection

L'introspection est une technique avancée de programmation objet qui consiste à analyser un objet sans le connaître, par exemple afin d'obtenir la liste de ses attributs et leurs valeurs.

Pour l'introspection des individus, la méthode `get_properties()` permet d'obtenir la liste des propriétés pour lesquelles l'individu possède au moins une relation.

```
■ >>> onto.bactérie_inconnue.get_properties()
    {bacterie.a_pour_forme,
      bacterie.a_pour_regroupement,
      bacterie.gram_positif,
      bacterie.nb_colonies}
```

Il est ensuite possible d'obtenir et/ou de modifier ces relations. Les fonctions `getattr(objet, attribut)` et `setattr(objet, attribut, valeur)` de Python permettent de lire ou d'écrire un attribut d'un objet Python, lorsque le nom de l'attribut est connu dans une variable (voir chapitre Le langage Python : adoptez un serpent !, section Fonctions et opérateurs pour la programmation objet), par exemple :

```
■ >>> for prop in onto.bactérie_inconnue.get_properties():
    ...     print(prop.name, "=", getattr(onto.bactérie_inconnue, prop.python_name))
    a_pour_regroupement = [bacterie.en_amas1]
    a_pour_forme = bacterie.rondel
    gram_positif = True
    nb_colonies = 6
```

Les valeurs retournées sont les mêmes qu'avec la syntaxe « individu.propriété » : il s'agit d'une valeur unique pour les propriétés fonctionnelles et d'une liste de valeurs pour les autres. Cependant, lorsque l'on effectue l'introspection, il est souvent plus facile de traiter toutes les propriétés de manière générique, qu'elles soient fonctionnelles ou non. Dans ce cas, la syntaxe « propriété[individu] » est préférable, car elle retourne toujours une liste de valeurs, y compris pour les propriétés fonctionnelles. Par exemple :

```
>>> for prop in onto.bactérie_inconnue.get_properties():
...     print(prop.name, "=", prop[onto.bactérie_inconnue])
a_pour_regroupement = [bacterie.en_amas1]
a_pour_forme = [bacterie.rondel]
gram_positif = [True]
nb_colonies = [6]
```

Pour l'introspection des classes, la méthode `get_class_properties()` fonctionne de manière similaire à celle des individus. Elle retourne les propriétés pour lesquelles la classe possède au moins une restriction existentielle (ou universelle, selon le type de propriété de classe, voir chapitre Constructeurs et restrictions, propriétés de classes, section Restrictions comme propriétés de classe) :

```
>>> onto.Pseudomonas.get_class_properties()
{bacterie.gram_positif,
 bacterie.a_pour_forme,
 bacterie.a_pour_regroupement}
```

Owlready considère les classes parentes, mais aussi les classes équivalentes. La syntaxe « propriété[classe] » peut être utilisée pour obtenir et/ou modifier les restrictions existentielles des classes.

Enfin, les méthodes `INDIRECT_get_properties()` et `INDIRECT_get_class_properties()` fonctionnent de la même manière, mais retournent également les propriétés indirectes (c'est-à-dire héritées d'une classe parente).

De plus, la méthode `constructs()` permet de parcourir l'ensemble des constructeurs qui font référence à une classe ou à une propriété. Par exemple, nous pouvons chercher les constructeurs faisant référence à la classe `EnChaînette` :

```
>>> list(onto.EnChaînette.constructs())
[ bacterie.Bactérie
& bacterie.a_pour_forme.some(bacterie.Ronde)
& bacterie.a_pour_forme.only(bacterie.Ronde)
& bacterie.a_pour_regroupement.some(bacterie.EnChaînette)
& bacterie.a_pour_regroupement.only(Not(bacterie.Isolé))
& bacterie.gram_positif.value(True) ]
```

Nous en obtenons un seul, qui est une intersection incluant une restriction existentielle avec pour valeur la classe `EnChaînette`. Nous pouvons ensuite utiliser la méthode `subclasses()` de ce constructeur pour obtenir la liste de toutes les classes qui l'utilisent :

```
>>> constructeur = list(onto.EnChaînette.constructs())[0]
>>> constructeur.subclasses()
[bacterie.Streptocoque]
```

Nous retrouvons ainsi la classe `Streptocoque`, dans laquelle nous avons placé cette restriction (voir chapitre Les ontologies OWL, section Définitions (ou équivalences)).

6. Inverser les restrictions

Les restrictions permettent de définir des relations au niveau des classes de l'ontologie, par exemple « `Pseudomonas a_pour_forme some Allongée` ». Owlready permet d'accéder facilement à ces relations avec la syntaxe « `Classe.propriété` » :

```
>>> onto.Pseudomonas.a_pour_forme
bacterie.Allongée
```


Mais comment lire cette restriction existentielle « à l'envers », c'est-à-dire, à partir de la classe *Allongée*, remonter à la classe *Pseudomonas* ? Même si nous avons défini la propriété inverse, que nous pourrions appeler « *est_forme_de* », elle ne permettrait pas de répondre à notre question, comme le montre l'exemple suivant :

```
>>> with onto:
...     class est_forme_de(ObjectProperty):
...         inverse = onto.a_pour_forme

>>> onto.Allongée.est_forme_de
[]
```

En effet, d'un point de vue logique, les deux propositions suivantes sont différentes :

- « *Pseudomonas a_pour_forme some Allongée* »
- « *Allongée est_forme_de some Pseudomonas* »

La première indique que tout *Pseudomonas* a une forme *Allongée*, ce qui est vrai. La seconde indique que toute forme *Allongée* est la forme d'un *Pseudomonas*, ce qui n'est pas la même chose (et n'est pas vrai). Par exemple, la forme *Allongée* d'un ballon de rugby n'est pas la forme d'un *Pseudomonas*.

De même, pour les deux propositions suivantes :

- « *Noyau est_une_partie_de some Cellule* »
- « *Cellule a_pour_partie some Noyau* »

La première indique que tout noyau fait partie d'une cellule. La seconde indique que toute cellule a un noyau, ce qui n'est pas la même chose : en biologie, la première proposition est vraie tandis que la seconde est fausse (les bactéries sont des cellules sans noyaux).

Néanmoins, il est parfois utile de pouvoir lire les relations existentielles à l'envers. Owlready le permet en combinant les méthodes `constructs()` et `subclasses()` comme nous l'avons étudié à la section précédente. La méthode `inverse_restrictions()` permet d'automatiser cela :

```
>>> set(onto.Allongée.inverse_restrictions(onto.a_pour_forme))
{bacterie.Pseudomonas, bacterie.Bacille}
```

Notez que nous avons utilisé `set()` pour afficher le générateur retourné par `inverse_restrictions()`, en enlevant les doublons.

7. Exemple : utiliser Gene Ontology et gérer les relations « partie-de »

Gene Ontology (GO) est une ontologie très utilisée en bioinformatique (voir chapitre Accéder aux ontologies en Python, section Ontologie volumineuse et cache disque). GO se compose de trois parties : les processus biologiques, les fonctions moléculaires et les composants de la cellule. Cette troisième partie décrit les différents éléments d'une cellule : membranes, noyau, organites (tels que les mitochondries)... Elle est particulièrement complexe à gérer, car elle comprend à la fois une hiérarchie d'héritage « classique » avec des relations « est-un », mais aussi une hiérarchie de relation « partie-de ». Dans cette dernière, appelée méronymie, il s'agit de décomposer la cellule en sous-parties, puis en sous-sous-parties... La racine de cette hiérarchie est donc la cellule entière, et les feuilles, les parties indivisibles.

OWL et Owlready possèdent des relations et des méthodes pour gérer la hiérarchie d'héritage (`subclasses()`, `descendants()`, `ancestors()`... voir chapitre Accéder aux ontologies en Python, section Classes). En revanche, il n'existe pas de relation standard OWL pour la méronymie, ni de méthodes spécifiques dans Owlready. Nous allons voir ici comment ajouter aux classes GO des méthodes pour accéder aux sous-parties et aux superparties, en prenant en compte à la fois les relations « partie-de » et les relations « est-un ».

GO étant assez volumineuse (près de 200 Mo), le chargement prend plusieurs dizaines de secondes, voire quelques minutes, en fonction de la puissance de l'ordinateur et du temps de téléchargement du fichier OWL. Nous allons donc charger GO et stocker le quadstore Owlready dans un fichier (voir chapitre Accéder aux ontologies en Python, section Ontologie volumineuse et cache disque). De plus, nous utiliserons ici l'import manuel pour associer nos méthodes Python aux classes OWL (voir section Import manuel), afin de ne pas avoir à modifier GO en lui ajoutant une annotation `python_module`.

GO utilise des identifiants arbitraires qui ne sont pas directement compréhensibles par l'humain. Le tableau suivant récapitule les identifiants GO dont nous aurons besoin par la suite :

Identifiant GO	Label	Description
GO_0005575	<i>cellular_component</i>	Composant de la cellule
BFO_0000050	<i>part of</i>	Partie de
BFO_0000051	<i>has_part</i>	A pour partie

```
# Fichier go_partie de.py
from owlready2 import *

default_world.set_backend(filename = "quadstore.sqlite3")
go = get_ontology("http://purl.obolibrary.org/obo/go.owl#").load()
obo = go.get_namespace("http://purl.obolibrary.org/obo/")
default_world.save()

def mon_rendu(entity):
    return "%s:%s" % (entity.name, entity.label.first())
set_render_func(mon_rendu)

with obo:
    class GO_0005575(Thing):
        @classmethod
        def sous_parties(self):
            resultats = list(self.BFO_0000051)
            resultats.extend(self.inverse_restrictions(obo.BFO_0000050))
            return resultats

        @classmethod
        def sous_parties_transitives(self):
            resultats = set()
            for descendant in self.descendants():
                resultats.add(descendant)
            for sous_partie in descendant.sous_parties():
                resultats.update(sous_partie.sous_parties_transitives())
            return resultats

        @classmethod
        def super_parties(self):
            resultats = list(self.BFO_0000050)
            resultats.extend(self.inverse_restrictions(obo.BFO_0000051))
            return resultats

        @classmethod
        def super_parties_transitives(self):
```

```

resultats = set()
for ancetre in self.ancestors():
    if not issubclass(ancetre, GO_0005575): continue
    resultats.add(ancetre)
for super_partie in ancetre.super_parties():
    if issubclass(ancetre, GO_0005575):
        resultats.update( \
            super_partie.super_parties_transitives())
return resultats

```

Ce module définit quatre méthodes de classes dans la classe `GO_0005575` (*cellular_component*) :

- `sous_parties()` permet de récupérer l'ensemble des sous-parties du composant. Cette méthode prend en compte les relations `BFO_0000051` (*has part*), mais aussi les relations `BFO_0000050` (*part of*) lues à l'envers, contrairement à ce que nous aurions obtenu avec `.INDIRECT_BFO_0000051` (voir chapitre Constructeurs et restrictions, propriétés de classe, section Restrictions comme propriétés de classe).
- `sous_parties_transitives()` retourne les sous-parties, en prenant en compte les classes filles et la transitivité (si A est une sous-partie de B et que B est une sous-partie de C, alors A est aussi une sous-partie de C).
- `super_parties()` et `super_parties_transitives()` fonctionnent de la même manière pour les superparties.

Nous pouvons ensuite importer ce module et accéder à GO et aux relations « partie-de ». Dans l'exemple suivant, nous interrogeons les relations « parties-de » du nucléole (*nucleolus* en anglais), qui est un composant situé dans le noyau (*nucleus*) de la cellule (*cell*).

```

>>> from owlready2 import *
>>> from go_partie_de import *

>>> nucleole = go.search(label = "nucleolus")[0]

>>> print(nucleole.sous_parties())
[GO_0005655:nucleolar ribonuclease P complex,
GO_0030685:nucleolar preribosome,
GO_0044452:nucleolar part,
GO_0044452:nucleolar part,
GO_0101019:nucleolar exosome (RNase complex)]

>>> print(nucleole.super_parties())
[GO_0031981:nuclear lumen]

```

Les relations directes (sans prendre en compte la transitivité) ne sont pas très informatives. Les relations transitives sont beaucoup plus riches :

```
>>> nucleole.sous_parties_transitives()
{GO_0034388:Pwp2p-containing subcomplex of 90S preribosome,
GO_0097424:nucleolus-associated heterochromatin,
GO_0005736:DNA-directed RNA polymerase I complex,
GO_0005731:nucleolus organizer region,
GO_0101019:nucleolar exosome (RNase complex),
[...] }

>>> nucleole.super_parties_transitives()
{GO_0031981:nuclear lumen,
GO_0005634:nucleus,
GO_0043226:organelle,
GO_0044464:cell part,
GO_0005623:cell,
GO_0005575:cellular_component,
[...] }
```

8. Exemple : un « site de rencontre » pour les protéines

À présent, nous allons utiliser les fonctionnalités du module `go_partie_de.py` pour réaliser un « site de rencontre » pour les protéines. Ce site permet d'entrer deux noms de protéines, et de déterminer dans quels compartiments de la cellule elles peuvent se rencontrer (si c'est possible !). D'un point de vue biologique, cela est important, car deux protéines qui n'ont pas de « site de rencontre » commun ne peuvent pas interagir ensemble.

Pour cela, nous utiliserons :

- Le module Python `Flask`, pour faire un site web dynamique (voir chapitre [Accéder aux ontologies en Python](#), section Exemple : créer un site web dynamique à partir d'une ontologie).

- Le module Python MyGene, pour effectuer des recherches sur le serveur MyGene et récupérer les concepts GO associés à chacune des deux protéines. Ce module permet de faire des recherches sur les gènes (et les protéines qu'ils codent). MyGene s'utilise de la manière suivante :

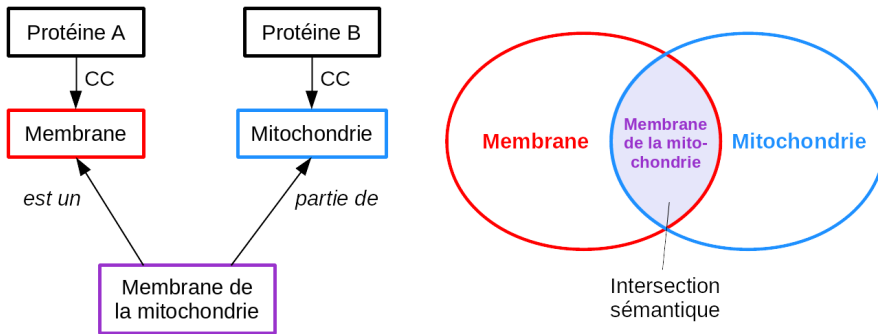
```
import mygene
mg = mygene.MyGeneInfo()
dico = mg.query('name:"<nom_de_gene>"',
               fields = "<champs_recherchés>",
               species = "<espèce>",
               size = <nombre_de_genes_à_rechercher>)
```

L'appel à MyGene retourne un dictionnaire contenant lui-même des listes et d'autres dictionnaires. Par exemple, nous pouvons rechercher l'ensemble des termes GO associés à l'insuline de la manière suivante :

```
>>> import mygene
>>> mg = mygene.MyGeneInfo()
>>> dico = mg.query('name:"insulin"',
...                 fields = "go.CC.id,go.MF.id,go.BP.id",
...                 species = "human", size = 1)
>>> dico
{'max_score': 13.233688, 'took': 17, 'total': 57,
 'hits': [{'_id': '3630', '_score': 13.233688,
           'go': {'BP': [{'id': 'GO:0002674'},
                        {'id': 'GO:0006006'}, [... ] ]}]}]}
```

Les champs `go.CC.id`, `go.MF.id` et `go.BP.id` représentent les trois grandes parties de GO (respectivement *Cellular Components*, *Molecular Functions* et *Biological Process*). Pour notre site de rencontre, nous n'utiliserons que les « CC ». Bien qu'issus de Gene Ontology, ceux-ci décrivent en fait la localisation dans la cellule du produit de gène, c'est-à-dire de la protéine (en général), et non du gène lui-même (les gènes restant normalement dans le noyau, pour les cellules eucaryotes). Plus d'informations sont disponibles sur le site de MyGene : <http://docs.mygene.info/en/latest/>

- Owlready et Gene Ontology (GO), pour réaliser l'intersection sémantique des termes GO décrivant les compartiments cellulaires des deux protéines. Une « simple » intersection (au sens ensembliste du terme) n'est pas suffisante : l'intersection doit prendre en compte à la fois les relations « est un » d'héritage et les relations « partie-de ». Par exemple, une protéine A présente uniquement dans les membranes et une protéine B présente uniquement dans les mitochondries pourront se rencontrer dans la membrane des mitochondries comme le montre le schéma suivant :



Le programme suivant décrit le site de rencontre pour protéine :

```
# Fichier site_rencontre.py
from owlready2 import *
from go_partie_de import *

from flask import Flask, request
app = Flask(__name__)

import mygene
mg = mygene.MyGeneInfo()

def chercher_proteine(nom_proteine):
    r = mg.query('name:"%s"' % nom_proteine, fields = "go.CC.id",
                species = "human", size = 1)

    if not "go" in r["hits"][0]: return set()

    cc = r["hits"][0]["go"]["CC"]
    if not isinstance(cc, list): cc = [cc]

    sites = set()
    for dico in cc:
        id_go = dico["id"]
```

```

        terme_go = obo[id_go.replace(":", "_")]
        if terme_go: sites.add(terme_go)

    return sites

def intersection_semantique(sites1, sites2):
    sous_parties1 = set()
    for site in sites1:
        sous_parties1.update(site.sous_parties_transitives())

    sous_parties2 = set()
    for site in sites2:
        sous_parties2.update(site.sous_parties_transitives())

    sites_communs = sous_parties1 & sous_parties2

    cache = { site : site.sous_parties_transitives()
              for site in sites_communs }

    sites_communs_sans_sous_parties = set()
    for site in sites_communs:
        for site2 in sites_communs:
            if (not site2 is site) and (site in cache[site2]): break
        else:
            sites_communs_sans_sous_parties.add(site)

    return sites_communs_sans_sous_parties

@app.route('/')
def page_saisie():
    html = """
<html><body>
  <form action="/resultat">
    Protéine 1: <input type="text" name="prot1"/><br/><br/>
    Protéine 2: <input type="text" name="prot2"/><br/><br/>
    <input type="submit"/>
  </form>
</body></html>"""
    return html

@app.route('/resultat')
def page_resultat():
    prot1 = request.args.get("prot1", "")
    prot2 = request.args.get("prot2", "")

    sites1 = chercher_proteine(prot1)
    sites2 = chercher_proteine(prot2)

```



```
sites_communs = intersection_semantique(sites1, sites2)

html = """<html><body>"""
html += """<h3>Site de la protéine 1 (%s)</h3>""" % prot1
if sites1:
    html += "<br/>".join(sorted(str(site) for site in sites1))
else:
    html += "(Aucun)<br/>"

html += """<h3>Site de la protéine 2 (%s)</h3>""" % prot2
if sites2:
    html += "<br/>".join(sorted(str(site) for site in sites2))
else:
    html += "(Aucun)<br/>"

html += """<h3>Sites de rencontre possibles</h3>"""
if sites_communs:
    html += "<br/>".join(sorted(str(site) for site in sites_communs))
else:
    html += "(Aucun)<br/>"

html += """</body></html>"""
return html

import werkzeug.serving
werkzeug.serving.run_simple("localhost", 5000, app)
```

Il commence par importer et initialiser l'ensemble des modules :

- Owlready.
- Le module `go_partie_de` que nous avons créé à la section précédente.
- Flask.
- MyGene.

Ensuite, la fonction `chercher_proteine()` est définie. Elle prend en entrée un nom de protéine (en anglais), tel que « insulin », et retourne l'ensemble des termes GO de type composant cellulaire (« CC ») qui lui sont associés dans MyGene. Pour cela, nous vérifions qu'au moins un résultat (*hit* en anglais) est trouvé, puis nous récupérons les « CC ». Si un seul CC est trouvé, MyGene le retourne, sinon, il s'agit d'une liste. Pour faciliter le traitement, nous créons systématiquement une liste appelée `cc`, puis nous la parcourons et extrayons l'identifiant GO. Les identifiants retournés par MyGene sont de la forme « GO:0002674 » et non « GO_0002674 » comme dans la version OWL de GO.

Nous remplaçons donc le « : » par un « _ ». Enfin, nous récupérons le concept de l'ontologie correspondant en utilisant l'espace de nommage `obo` (qui a été importé depuis le module `go_partie_de`).

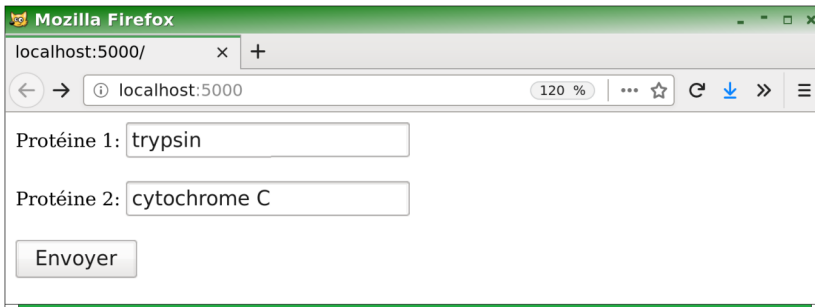
La fonction `intersection_semantique()` réalise l'intersection sémantique de deux ensembles contenant des concepts GO de composants cellulaires, en quatre étapes :

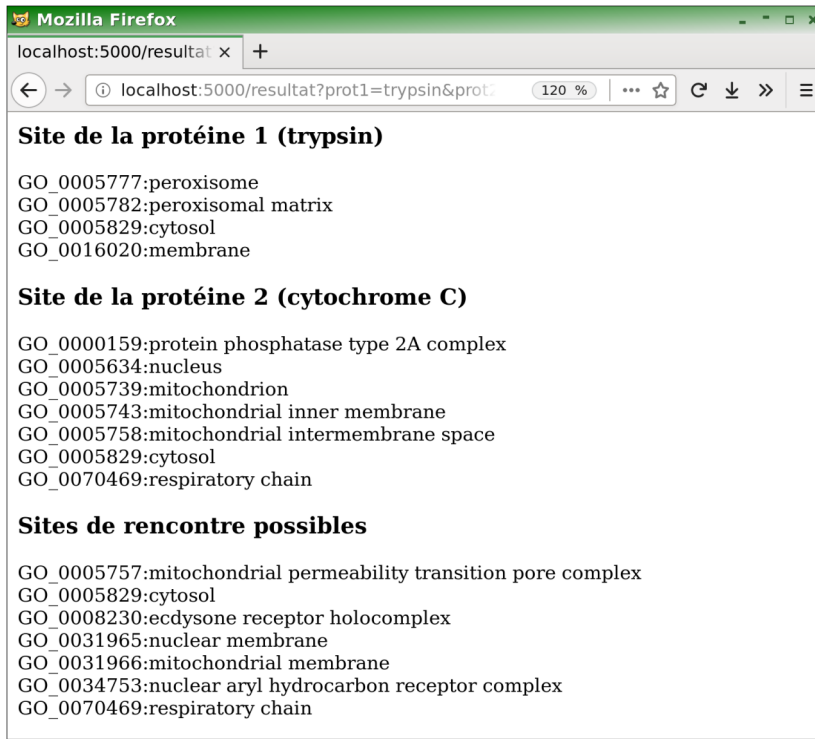
- Nous créons deux ensembles, `sous_parties1` et `sous_parties2`, contenant les composants associés à chacune des deux protéines ainsi que leurs sous-parties, de manière transitive. Pour cela, nous réutilisons la méthode statique `sous_parties_transitives()` que nous avons définie dans le module `go_partie_de` à la section précédente. Nous obtenons alors les ensembles de tous les endroits où peut être rencontrée chacune des deux protéines, en tenant compte des relations « est-un » et « partie-de ».
- Nous calculons l'intersection de ces deux ensembles avec l'opérateur « & » (voir chapitre Le langage Python : adoptez un serpent !, section Les ensembles (set) pour les ensembles en Python), et nous appelons le résultat `site_communs`.
- Il nous reste maintenant à simplifier l'ensemble `site_communs`. En effet, celui-ci comprend les concepts que nous recherchons, mais également tous leurs descendants et leurs sous-parties (dans l'exemple précédent avec « membrane » et « mitochondrie », nous avons donc « membrane de la mitochondrie » mais aussi « membrane interne de la mitochondrie » et « membrane externe de la mitochondrie »). Afin d'accélérer les traitements de l'étape suivante, nous créons tout d'abord un cache (à l'aide d'un dictionnaire). Celui-ci fait correspondre à chaque concept GO de `site_communs` l'ensemble de ses sous-parties (transitives).
- Nous créons un nouvel ensemble, `sites_communs_sans_sous_parties`, qui est vide au départ. Nous y ajoutons dedans tous les concepts de `site_communs` qui ne sont pas une sous-partie d'un autre concept de `site_communs`. Notez l'utilisation du « else de for », qui permet d'exécuter des instructions lorsque la boucle est allée jusqu'au bout (c'est-à-dire que le « break » n'a pas été rencontré, voir chapitre Le langage Python : adoptez un serpent !, section Boucles (for)). Enfin, nous retournons ce nouvel ensemble.

La suite du programme définit deux pages web avec Flask. La première (chemin « / ») est un formulaire basique avec deux champs textes pour entrer les noms des protéines et un bouton pour valider. La seconde (chemin « / resultat ») calcule et affiche le résultat. Elle appelle tout d'abord la fonction `chercher_proteine()` deux fois, une pour chaque protéine, puis la fonction `intersection_semantique()`. Enfin, elle génère une page web affichant les composants associés à la première protéine, à la seconde, et les composants où elles sont susceptibles de se rencontrer.

Afin de tester notre site de rencontre, voici quelques exemples de noms de protéines en anglais : trypsin, cytochrome C, insulín, insulín-degrading enzyme, insulín receptor, glucagon, hemoglobin, elastase, granzyme B, decorin, beta-2-microglobulin...

Les copies d'écran suivantes montrent le site de rencontre obtenu et son utilisation :





Mozilla Firefox

localhost:5000/resultat x +

localhost:5000/resultat?prot1=trypsin&prot: 120 %

Site de la protéine 1 (trypsin)

- GO_0005777:peroxisome
- GO_0005782:peroxisomal matrix
- GO_0005829:cytosol
- GO_0016020:membrane

Site de la protéine 2 (cytochrome C)

- GO_0000159:protein phosphatase type 2A complex
- GO_0005634:nucleus
- GO_0005739:mitochondrion
- GO_0005743:mitochondrial inner membrane
- GO_0005758:mitochondrial intermembrane space
- GO_0005829:cytosol
- GO_0070469:respiratory chain

Sites de rencontre possibles

- GO_0005757:mitochondrial permeability transition pore complex
- GO_0005829:cytosol
- GO_0008230:ecdysone receptor holocomplex
- GO_0031965:nuclear membrane
- GO_0031966:mitochondrial membrane
- GO_0034753:nuclear aryl hydrocarbon receptor complex
- GO_0070469:respiratory chain