SimPhoNy documentation

Materials Data Science and Informatics team at Fraunhofer IWM

Dec 07, 2022

GETTING STARTED

CHAPTER

ONE

OVERVIEW

SimPhoNy is an ontology-based framework aimed at enabling interoperability between different simulation and data management tools, with a focus on materials science.

1.1 What can SimPhoNy be used for?

1.1.1 Manipulate ontology-based linked data, a format well suited for FAIR data principles

Linked data is a format for structured data that facilitates the interoperability among different data sources. In particular, the data is structured as a directed graph, consistent of nodes and labeled arcs. With SimPhoNy, you can not only manipulate this linked data, **but also transform existing non-linked data into linked data**.

To better understand the idea of linked data, take a quick glance at the toy example below. It shows data about a city from three different data sources: the city's traffic authority, a map from a city guide, and the university registry. As some of the concepts are present in multiple datasets, the linked data representation naturally joins all of them into a single one.

Linked data about a city from three different sources: the city's traffic authority, a map from a city guide, and the university registry. Each data source is represented using a different color and column.

Although the example above shows just plain linked data, in SimPhoNy, the linked data is enhanced with ontologies, which give **meaning** to the data. Specifically, SimPhoNy works with ontologies based on the Web Ontology Language, making the data compatible with the Semantic Web.

1.1.2 Fetch data from a database, run a simulation and immediately store the results

Ontology-based linked data is not only well suited for the interoperability of data, but also of software tools. In SimPhoNy, one can instantiate individuals from special ontology classes called *wrappers*. These wrappers are in fact a software interface between the core of SimPhoNy (ontology based) and external software tools, disguised to the user as an ontology class. We have already developed wrappers for a few database backends and popular simulation engines for materials science. You can have a look at the existing wrappers on our GitHub organization. If needed, you may even consider *developing your own*!

As a SimPhoNy user, you can see the data stored in the external software tools transparently as ontology individuals through the wrappers. In this way, moving data between different software tools becomes as simple as moving or copying it from one wrapper to another.

For example, linked data stored in a SQLite database can be used to run a simulation just by adding the ontology individuals contained in the SQLite wrapper to the Simulation Engine wrapper. Similarly, the ontology individuals representing the results can be simply added back into the database wrapper.

At this point, the results could be fetched again and for example, visualized with the help of a plotting library.

1.1.3 Couple simulation engines easily

Exactly in the same way that the data can be moved between a database and a simulation engine using their respective wrappers, it can also be moved between simulation engines.

This functionality facilitates the coupling and linking between such simulation engines. For example, in the domain of materials science, a certain engine might be useful for representing structures made up of atomistic particles (molecular dynamics), while another software tool could be focussed on representing bodies of fluids (fluid dynamics). As SimPhoNy can enable communication between the two tools, they could both be run and synced simultaneously to create more complex scenarios, such as a multi-scale simulation.

The concepts of coupling and linking illustrated in a video.

In order achieve that, it would be necessary to translate the input and output formats of both simulation engines. However, given that the necessary wrappers exist, and their ontologies are compatible, this task becomes relatively simple thanks to SimPhoNy! At the end of the coupling process, just add the results to a database wrapper to store them.

Coupling of two simulation engines, one that handles fluid dynamics (macroscopic behavior) and another that takes care of molecular dynamics (microscopic behavior).

CHAPTER

TWO

TERMINOLOGY

The name 'SimPhoNy' stems from the SimPhoNy EU-project in which it was originally developed (see more details here).

Here are some additional terms that are used throughout the documentation:

1. `backend`: a third party application or service. Simulation engines and databases are examples of backends.
2. `wrapper`: a plugin for OSP-core that adds support to a new backend. It must allow the user to interact with the backend through the same API as OSP-core.
3. `ontology`: an explicit, formal specification of a shared conceptualization. In the context of ontology, other relevant terms are:
    1. `class`: a concept. E.g., 'City', 'Experiment'.
    2. `attribute`: a property of a class that has a data type. E.g., 'name' of the type String which could be used as an attribute of 'City'.
    3. `individual`: an instance of a class. E.g., an instance of the class 'City' can be used to represent the city of Freiburg in which case it would have the attribute 'name' with the value 'Freiburg'.
    4. `relationship`: a type of a way in which one individual relates to another. E.g., 'Has-A' which could use to form the relationship 'Freiburg (City) Has-A Dreisam (River)'.
    5. `entity`: a general term that can refer to a class, a relationship, attribute, or an individual. E.g., 'City', 'name', 'Has-A', the Freiburg individual are all entities.
    6. `namespace`: an ontology identifier. E.g., 'city_ontology' which could be used as a namespace for the ontology that consists of the entities 'City', 'name' and 'Has-A'.
        - Each entity is uniquely identified by its name and the namespace it is contained in. We call <namespace name>.<entity name> the `qualified entity name`.
4. `CUDS`: Common Universal Data Structure. A data structure that is used to uniformly represent ontology concepts in programming code.
    - CUDS exposes an API that provides CRUD (Create, Read, Update and Delete) functionalities.
    - CUDS is a recursive data structure in that a CUDS object may contain other CUDS objects.
    - CUDS is the fundamental data type of OSP-core, a framework that establishes interoperability between software systems that are built on top of ontologies.

CHAPTER

THREE

FUNDAMENTAL CONCEPTS

In this section we will present some of the main concepts behind SimPhoNy.

3.1 General notions

3.1.1 Degrees of interoperability

There is a multitude of tools and programs out there, all with their own formats and protocols.

Every time a user wants to use one of these tools, they must familiarise themselves with the software. Furthermore, if they want to integrate multiple tools in one workflow, the must, in most cases, take care of the conversion on their own.

Based on how tools communicate with other tools, we can define 3 levels:

Compatibility

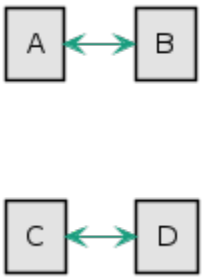


Fig. 1: Compatibility

When we say two tools are compatible, they are able to communicate with each other in a one to one basis. This means the tools must either use the same format, or be able to convert to the format of the other.

If we compare this to speaking languages, you could say `A` and `B`, or `C` and `D` speak the same language. However, `A` has no way to talk with `C` or `D`, for example.

De Facto Standard

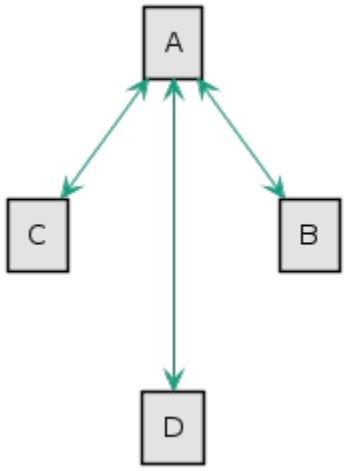


Fig. 2: De Facto Standard

In this case, the level of operability is higher. All tools know how to communicate with a tool whose format has become a de facto standard.

To continue with our language simile, `A` would be a translator that speaks the languages of `B`, `C` and `D`. If `B` wants to talk to `C`, they must first relay the message to `A`, and `A` will convert it to a format that `C` understands.

Interoperability

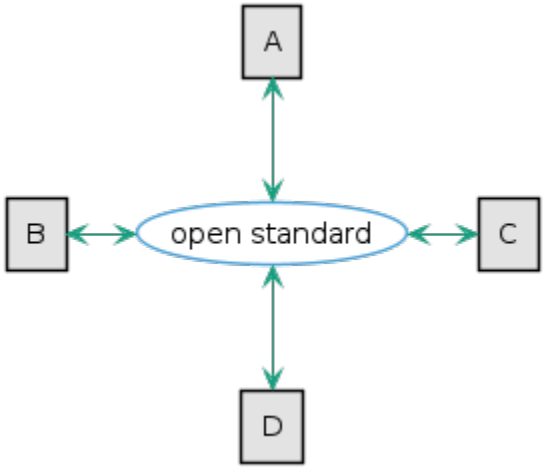


Fig. 3: Interoperability

The highest level of operability is interoperability. Here there is no need for all tools to go through the De Facto standard, because there is a format that is known by all of them and enables all components to communicate among themselves.

This final stage could be compared to all parties using an instant translator that can convert text from one language into any other.

Interoperability between software tools is one of the most important objectives of the SimPhoNy framework.

3.1.2 Semantic vs. syntactic

We can interpret a word as a specific sequence of characters without caring about the meaning itself. This way, a simulation engine parsing an input file will know that the integer written after the keyword `step` will be used to set the number of iterations the execution loop will run. It does nothing else, and could as easily use the sequence `ppp`.

However, for a person, the word `step` will be a sign representing a specific concept. It could be the number of rounds in a simulation, but also the consecutive instructions in an algorithm, the different levels in a stair or the motion a person makes when walking. Based on the domain, a person can also list other relevant concepts and relationships (e.g. when thinking of a stair, the `material` or the `width`).

Being able to know the semantic meaning of an instance, and hence its connection to other concepts, is one of the principles of SimPhoNy. For achieving this goal, ontologies play a major role.

3.1.3 Ontology

Important: An ontology is a formal specification of a shared conceptualization. [Borst, 1997]

Let's look at the individual components of this definition, starting from the end.

- *Conceptualization*, an ontology will work on the ideas and relationships in an area of interest.
- *Shared*, the ideas and concepts are perceived and agreed by multiple people.
- *Specification*, it will define and describe them in detail, following some predetermined rules and format.
- *Formal*, meaning it will follow a machine readable syntax.

In a simpler way, an ontology can be seen as the definition of concepts relevant to a given domain, as well as the relationships between them, in a way that a machine can interpret it.

For a deeper, more detailed analysis of the definition, refer to [Guarino, 2009].

Ontologies are more elaborated than taxonomies in that they can include multiple kinds of relationships (not just parent-child) between complex concepts in big domains.

EMMO

The Elementary Multiperspective Material Ontology (EMMO, previously the European Materials Modelling Ontology) is an ontology developed by the European Materials Modelling Council (EMMC). EMMO's goal is to define a representational system universal for scientists in the field of materials modelling to enable interoperability.

It has been designed from the bottom up, starting with the concepts of different domains and application fields and generalising into a middle and top level layers, and it is currently being further developed in multiple projects of the European Union.

SimPhoNy is being developed with the intention of being compatible with EMMO, and an easy installation of the ontology is available (further explained *here*).

There is also documentation available for developing an EMMO compliant ontology (requires login).

3.1.4 CUDS

CUDS, or Common Universal Data Structure, is the ontology compliant data format of OSP-core:

- **CUDS is an ontology individual**: each CUDS object is an instantiation of a class in the ontology. If we assume a food ontology that describes classes like pizza or pasta, a CUDS object could represent one specific pizza or pasta dish, that exists in the real world. Similar to ontology individuals, CUDS objects can be related with other individuals/CUDS by relations defined in the ontology. Like a *pizza* that 'hasPart' *tomato sauce*
- **CUDS is API**: To allow users to interact with the ontology individuals and their data, CUDS provides a CRUD API.
- **CUDS is a container**: Depending on the relationship connecting two CUDS objects, a certain instance can be seen as a container of other instances. We call a relationship that express containment an 'active relationship'. In the pizza example, 'hasPart' would be an 'active relationship'. If one would like to share the pizza CUDS object with others, one would like to share also the tomato sauce.
- **CUDS is RDF**: Internally a CUDS object is only an interface to an RDF-based triple store that contains the data of all CUDS objects.
- **CUDS is a node in a graph**: : CUDS being individuals in an RDF graph implies that each CUDS object can also be seen as a node in a graph. This does not conflict with the container perspective, instead we see it as to different views on the data.

3.2 Technologies and frameworks

3.2.1 RDF

RDF (Resource Description Framework) is a formal language for describing structured information used in the Semantic Web. Its first specification was published in 1999 and extended in 2004.

Knowledge is represented in directed graphs where the nodes are either ontological classes, instances of those classes or literals and the edges the relationships connecting them.

The graph is serialised in the form of triples of the form "subject-predicate-object"

- *Subject*: The IRI of the entity the triple refers to. Blank nodes have no IRI, but they are outside of the scope of this thesis.
- *Predicate*: IRI of the relationship from subject to object.
- *Object*: Literal or IRI of an entity

The following is an example of an RDF triple. This example will also be used to show the different serialisation formats of RDF. For the IRIs, `dbpedia`'s namespace was used.

Fig. 4: RDF triple sample

The most used formats for storing RDF data are:

- XML: Historically the most common format given the amount of libraries for handling it. It was released hand in hand with the RDF specification. Unfortunately, XML is best used with tree-like structures rather than graphs, which also makes it harder for humans to read.

  The example triple in XML is:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dbp="http://dbpedia.org/property/">
    <rdf:Description rdf:about="http://dbpedia.org/resource/The_Lord_of_the_Rings">
        <dbp:author rdf:resource="http://dbpedia.org/resource/J._R._R._Tolkien"/>
    </rdf:Description>
</rdf:RDF>
```

- N3: Notation3 is designed with human readability as a motivator. The RDF triples are written one per line, with the possibility to define common prefixes and other directives for simplicity.

  The previous example in N3 would be:

```
    @prefix dbo: <http://dbpedia.org/ontology/> .
    @prefix dbr: <http://dbpedia.org/resource/> .
    dbr:The_Lord_of_the_Rings  dbo:author  dbr:J._R._R._Tolkien .
```

- Turtle: Based on N3, it strips some of its syntax, making it easier to parse for machines. The recurring example would be exactly the same in Turtle as in N3.

- N-Triples: N-Triples are even simpler, without any of the syntactic sugar from N3 or Turtle. The triples are written one per line without prefixes. This makes it a very easy format to parse but complex to maintain/read by a human.

  The following representation should be in one line (it has been split for readability)

```
  <http://dbpedia.org/resource/The_Lord_of_the_Rings>
    <http://dbpedia.org/ontology/author>
    <http://dbpedia.org/resource/J._R._R._Tolkien> .
```

- JSON-LD: uses the commonly accepted web data scheme for serialising RDF triples. Easier than XML for humans, JSON has standard libraries in practically all programming languages.

```
The example in JSON is:
```

```
{
  "@id": "http://dbpedia.org/resource/The_Lord_of_the_Rings",
  "http://dbpedia.org/property/author": [
    { "@id": "http://dbpedia.org/resource/J._R._R._Tolkien" }
  ]
}
```

  SimPhoNy supports all the previous formats (plus a simpler custom YAML) as inputs in the ontology installation.

SPARQL

SPARQL (recursively SPARQL Protocol and RDF Query Language) is the most common query language for RDF. Queries are graph patterns (similar to the triples of Turtle) with variables for the parts of the pattern that make up the result.

Variables start with the identifier ? and represent concrete values that will be matched in the query process. They can appear in multiple locations in the patterns and those present in the `SELECT` clause will be returned as the query result.

The query for the author of *The Lord of the Rings* from our sample triples in SPARQL is:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
SELECT ?person WHERE {
    dbr:The_Lord_of_the_Rings  dbo:author  ?person .
}
```

The SPARQL query language offers multiple types of result sets and clauses, most of which won't be used for this Master's thesis. One which should be mentioned is the `FILTER` keyword. This will limit the result to those that evaluate `true` to the expression inside the brackets. For instance (omitting the prefix declaration for simplicity):

```
SELECT ?character WHERE {
    ?character dbp:affiliation dbr:The_Lord_of_the_Rings .
    ?character dbo:age ?age .
    FILTER(?age >= 100)
}
```

The previous query would return the characters from the book series with an age higher or equal to 100. (Note that while the query is correct, the result is empty, as such information is not stored on DBpedia).

For a very interesting and comprehensive introduction into RDF and SPARQL, see [Hitzler, 2009].

CHAPTER

FOUR

GENERAL ARCHITECTURE

The following architecture has the aim to cover and support the goals presented in the *overview section*.

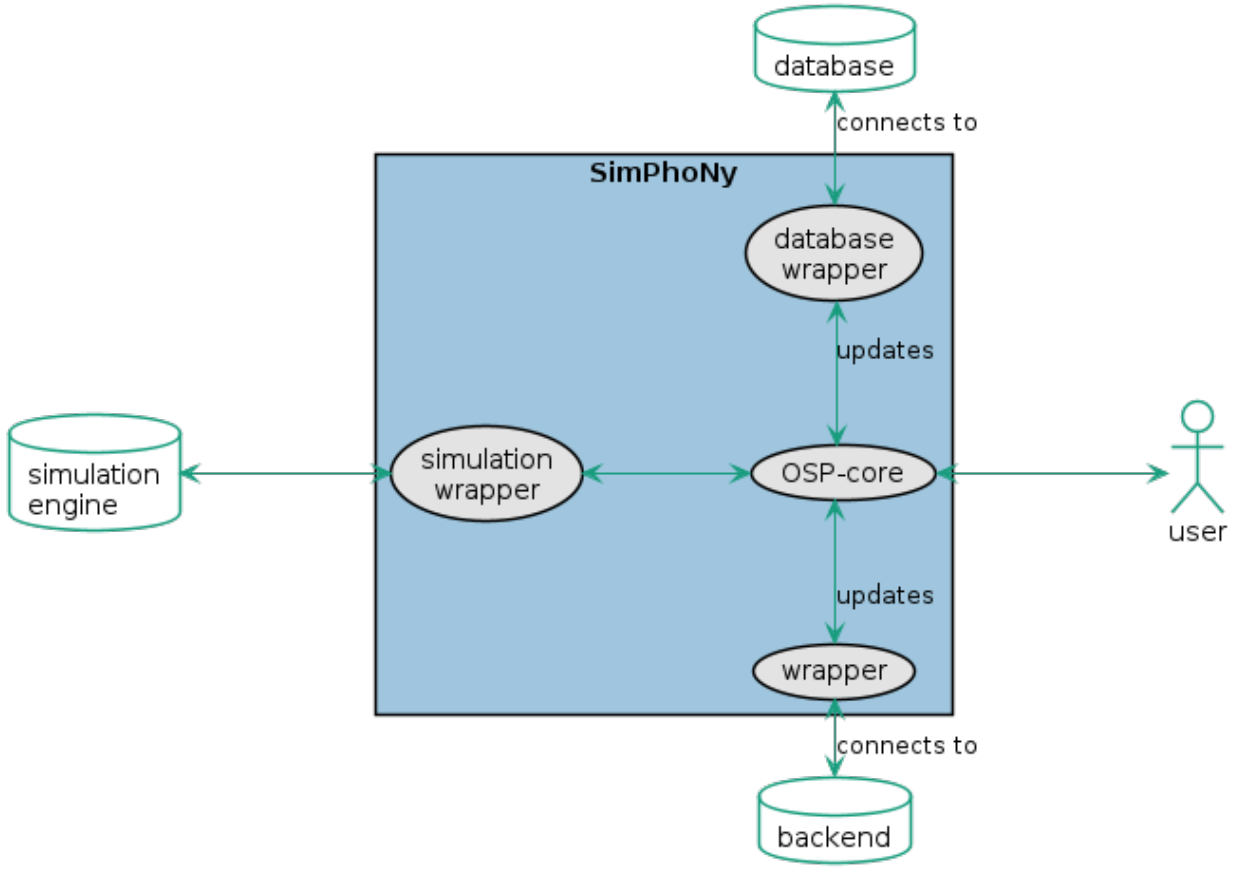


Fig. 1: Interoperability concept

As you can see, OSP-core provides the standard data format and API, and the wrappers take care of mapping that format to and from the backend specific syntax and API.

In order to simplify and generalise the usage as much as possible, the backend specific and syntactic knowledge should be abstracted to ontology concepts that encompass all third party tools.

For that, a 3 layer schema is used:

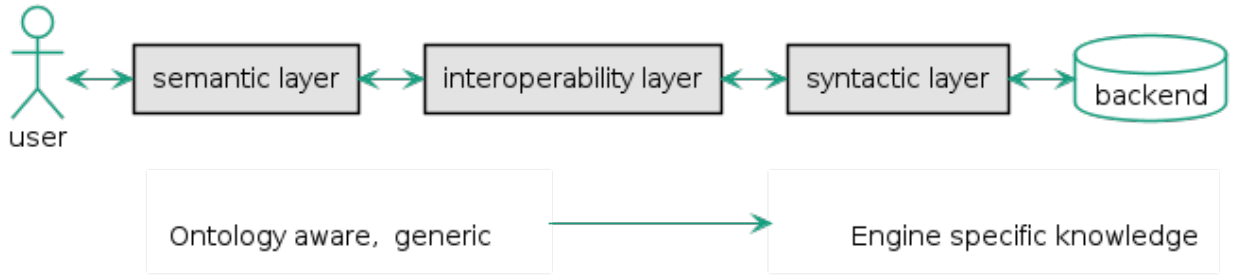


Fig. 2: Three layered design

- The *Semantic layer* are the classes generated from the ontology with the CUDS API.
- The *Interoperability layer* maps the changes in the semantic layer to calls in the syntactic layer.
- The *Syntactic layer* provides access to the backend.

The closer to the user, the closer to the ontology concepts. The abstraction is replaced by specificity when you move towards the backend.

For example, the City, Street or Neighborhood classes from the demonstrative *City Ontology* included in OSP-core, as well as the individuals that can be instantiated using them, would be part of the semantic layer. Any wrapper (e.g. the included SQLite wrapper), would be part of the interoperability layer. Finally, following the SQLite example, the sqlite3 library from python would be part of the syntactic layer.

For a full explanation on the architecture and design, go to *detailed design*.

4.1 OSP-core

OSP-core is the main component of the SimPhoNy framework. It is independent of any backend and provides the basic ontology based data structures for the seamless exchange of data between wrappers.

4.1.1 Ontology file

OSP-core requires an ontology file to create the appropriate CUDS classes.

Said ontology must be either in a YAML format as defined by *our specification* or *one of the supported owl ontologies*.

```
---
version: "0.0.3"

namespace: "city"

ontology:

  encloses:
    subclass_of:
    - cuba.activeRelationship
    inverse: city.isEnclosedBy

  isEnclosedBy:
    subclass_of:
```

(continues on next page)

(continued from previous page)

```
    - cuba.passiveRelationship
    inverse: city.encloses

  hasInhabitant:
    subclass_of:
    - city.encloses

  ################

  CityWrapper:
    subclass_of:
    - cuba.Wrapper
    - city.hasPart:
        range: city.City
        cardinality: 1+
        exclusive: false

  ################

  City:
    subclass_of:
    - city.PopulatedPlace
    - city.hasPart:
        range: city.Neighborhood
        cardinality: many
        exclusive: true
    - city.isPartOf:
        range: city.CityWrapper
        cardinality: 0-1
        exclusive: true
    - city.hasMajor:
        range: city.Citizen
        cardinality: 0-1
        exclusive: true

  Building:
    subclass_of:
    - city.ArchitecturalStructure
    - city.hasPart:
        range: city.Address
        cardinality: 1
        exclusive: false
    - city.hasPart:
        range: city.Floor
        cardinality: many
        exclusive: false
    - city.isPartOf:
        range: city.Street
        cardinality: 1
        exclusive: true

  Citizen:
```

(continues on next page)

(continued from previous page)

```
    subclass_of:
    - city.Person
```

OSP-core can be used with EMMO (Elementary Multiperspective Material Ontology) out of the box. See more *here*.

4.1.2 Python classes

Upon installation of OSP-core, each ontology class (except from attributes and relationships) becomes a python class.

Since each ontology has a namespace, it can be used to import the classes and create cuds objects:

```
from osp.core.namespaces import cuba, another_namespace

entity = cuba.Entity()
other_entity = another_namespace.SomeOtherEntity()
```

4.1.3 Sessions

The *sessions* are the interoperability classes that connect to where the data is stored. *In the case of wrappers*, they take care of keeping consistency between the backends (e.g. databases) and the internal registry.

The `CoreSession` is the default one used when instantiating a new object in your workspace. When you add an object to a wrapper, a copy of the object is created in the registry belonging to the session of the wrapper.

4.2 Wrappers

Like we have mentioned in previous sections, wrappers allow the user to interact through the cuds API with different backends.

Since each backend is different, for more detailed documentation of each wrapper we suggest going through the different available repositories.

For more technical information regarding wrappers, particularly for wrapper developers, we recommend visiting *wrapper development*.

CHAPTER

FIVE

INSTALLATION

For the installation and usage of the framework Python 3.6 or higher is needed. We *highly* encourage the use of a virtual environment or a conda environment.

```
# virtual environment
python3 -m venv SimPhoNy
source SimPhoNy/bin/activate
```

```
# conda
conda create -n <env_name>
conda activate <env_name>
```

OSP-core is available on PyPI, so it can be installed using `pip`:

```
pip install osp-core
```

For an installation from source, see *here*.

After installing OSP-core, you can install your ontology namespaces. We provide the *pico* tool for that purpose.

```
pico install <path/to/ontology.yml>

# If you have issues using pico directly, you can use
python -m osp.core.pico install <path/to/ontology.yml>
```

5.1 Wrapper installation

Wrappers are currently not available on PyPI, so they must be installed from source. First, the repository is cloned:

```
git clone https://github.com/simphony/<some-wrapper>.git
cd some-wrapper
```

5.1.1 Local wrapper installation

With OSP-core installed, if the wrapper has its own ontology, it *must* be installed:

```
pico install <path/to/ontology.yml>
```

For the wrappers that require the installation of a backend, a `install_engine.sh` script is usually provided. It will automatically call `install_engine_requirements.sh`, where the engine specific requirements are installed.

```
./install_engine.sh
```

Now, the wrapper can be installed:

```
pip install .
```

5.1.2 Wrapper Docker image

Some wrappers also provided a Dockerfile for an automatic installation in a container. The dockerfile should contain the information needed to run it inside.

5.2 Installing OSP-core from source

If you are a developer or an advanced user, you might be interested in installing OSP-core from source.

To do so, first the repository must be cloned:

```
git clone https://github.com/simphony/osp-core.git
cd osp-core
```

The installation is based on setuptools:

```
# build and install (recommended)
pip install .

# alternative
python3 setup.py install
```

or:

```
# build for in-place development (recommended)
pip install -e .

# alternative
python3 setup.py develop
```

CHAPTER

SIX

UTILITIES

In this section we will compile a list of useful utility functions, tools and examples of their usage. These functions are part of OSP-core and are used as an extension of the *main API*.

6.1 pico

Our tool for installing ontologies is called `pico`. It is a recursive acronym that stands for *Pico Installs Cuds Ontologies*.

There are 3 main operations that can be done with pico:

- Install ontologies.
- List the installed ontologies.
- Remove installed ontologies.

`pico` can be used both from the *command-line* and *as a Python module within the Python shell*.

6.1.1 Using pico from the command line

There are different possible logging levels available, and they can be set via `--log-level <ERROR|WARNING|INFO|DEBUG>`. The default value is `INFO`.

pico installs

Usage:

- `pico install <path/to/ontology_yml_file.yml>`
- `pico install <path/to/ontology_yml_file1.yml> <path/to/ontology_yml_file2.yml> ...`
- `pico install city foaf emmo dcat2` (the installation of these specific well-known ontologies is available via this shortcut)

Behaviour:

- The ontology file is parsed, and the entities mapped to Python objects.
- The Python objects can be imported via their namespace `from osp.core.namespaces import namespace`.

Example:

```
(venv) user@PC:~$ pico install city
INFO [osp.core.ontology.installation]: Will install the following namespaces: ['city']
INFO [osp.core.ontology.yml.yml_parser]: Parsing YAML ontology file /.../osp-core/osp/
↪core/ontology/docs/city.ontology.yml
INFO [osp.core.ontology.yml.yml_parser]: You can now use `from osp.core.namespaces␣
↪import city`.
INFO [osp.core.ontology.parser]: Loaded 367 ontology triples in total
INFO [osp.core.ontology.installation]: Installation successful
```

pico lists

Usage: `pico list`

Behaviour:

- The installed namespaces and packages are printed out. A package can be uninstalled and can contain many namespaces. A namespace can be imported within the Python shell.

Example:

```
Packages:
        - qe
        - city
Namespaces:
        - xml
        - rdf
        - rdfs
        - xsd
        - cuba
        - owl
        - qe
        - city
```

pico uninstalls

Usage:

- `pico uninstall <package>`
- `pico uninstall all`

Behaviour:

- The specified packages are uninstalled.
- All packages except the uninstalled ones are re-installed.

Example:

```
(venv) user@PC:~$ pico uninstall city
INFO [osp.core.ontology.installation]: Will install the following namespaces: ['qe']
INFO [osp.core.ontology.yml.yml_parser]: Parsing YAML ontology file /home/<username>/.
↪osp_ontologies/qe.yml
INFO [osp.core.ontology.yml.yml_parser]: You can now use `from osp.core.namespaces␣
↪import qe`.
```

(continues on next page)

(continued from previous page)

```
INFO [osp.core.ontology.parser]: Loaded 205 ontology triples in total
INFO [osp.core.ontology.installation]: Uninstallation successful
```

Conflicts with other "pico" installations

Some Operating Systems might have a pre-existing tool called *pico*. In most cases, the previous commands should work, but if any problem arises, you can use the following alternative:

```
python -m osp.core.pico <command>
```

For example:

```
python -m osp.core.pico install city
```

6.1.2 Using pico as a Python module

`pico` can also be used within the Python shell. In particular, four functions are available to be imported from the `osp.core.pico` module,

```
from osp.core.pico import install, namespaces, packages, uninstall
```

that cover the three main operations that pico is meant to perform: installing ontologies (`install`), uninstalling ontologies (`uninstall`), and listing the installed ontologies (`packages`, `namespaces`).

Each function is used in a similar way to its command-line counterpart.

- `install`: accepts *one or more* positional arguments of string type, which can be either paths to `yml` ontology installation files or names of ontologies that can be installed via this shortcut. It is meant to clone the behavior of its command-line counterpart.
- `uninstall`: accepts *one or more* positional arguments of string type, which must be names of already installed ontology packages. It also clones the behavior of its command-line counterpart.
- `packages`: accepts no arguments and returns an iterator over the names of the installed packages.
- `namespaces`: accepts no arguments and returns an iterator yielding one `OntologyNamespace` object for each installed namespace.

Usage examples:

- `install('city', 'path/to/ontology_yml_file.yml')`, `install('foaf', 'dcat2')`
- `uninstall('city', 'foaf')`
- `print(list(packages()))`
- `print(list(namespaces()))`

6.1.3 Ontology installation folder

The installed ontologies are stored in the directory `~/.osp-ontologies` by default. On Windows, ~ usually refers to the path `C:\Users\<my username>`.

The installation directory can be changed by setting the environment variable `OSP_ONTOLOGIES_DIR`. Such action would move it to `$OSP_ONTOLOGIES_DIR/.osp-ontologies`.

6.2 Tips and tricks

The following are some utility functions and shortcuts for working with cuds. For those that are present in the util package, the import is `from osp.core import utils`.

- `utils.get_relationships_between(a, b)` returns a set with the relationships that connect `a` and `b` .
- `a.get_attributes()` returns a dictionary with the name and the value of the attributes of `a`.
- `a.is_a(oclass)` is True if the instance `a` is or inherits from `oclass`.
- `osp.core.get_entity("namespace.entity")` returns the class associated with an `entity` in a `namespace`. It can be used to instantiate objects.
- `[attr.argname for attr in oclass.attributes.keys()]` returns a list with the attributes of an `oclass`.
- `[attr.namespace for attr in oclass.attributes.keys()]` returns a list with the namespace of the attributes of an `oclass`.

6.3 Visualisation

There are two ways of visualising information about a Cuds structure, one as a text output to the standard output (*pretty print*), and another one as a dot graph (*cuds2dot*).

Another useful dot graph visualisation tool called *ontology2dot* is available for ontology YML files.

Warning: The graphic visualisation tools that generate a dot file require Graphviz to be installed in the system.

6.3.1 Pretty print

Location: `from osp.core.utils import pretty_print`.

Usage: `pretty_print(cuds_object)`

Behaviour:

- The UUID, oclass and attributes of the given object are printed.
- All the related objects are also printed in a recursive fashion.
- The relationship to the contained objects is stated.

Example:

```
>>> pretty_print(emmo_town)
Cuds object named <EMMO town>:
  uuid: 06b01f5a-e8c1-44a5-962d-ea0c726e97d0
  type: city.City
  superclasses: city.City, city.PopulatedPlace, city.GeographicalPlace, cuba.Entity
  values: coordinates: [42 42]
  description:
    To Be Determined

   |_Relationship city.hasInhabitant:
   | -  city.Citizen cuds object named <Emanuele Ghedini>:
   | .  uuid: f1bd9143-6472-4b24-94b5-1c5fc4c6e5b6
   | .  age: 25
   | -  city.Citizen cuds object named <Adham Hashibon>:
   | .  uuid: 3b774c96-1a0c-403b-b0d0-05d6cd38c52c
   | .  age: 25
   | -  city.Citizen cuds object named <Jesper Friis>:
   | .  uuid: 40d2335c-a335-4d07-b142-fb2b9b7581a7
   | .  age: 25
   | -  city.Citizen cuds object named <Gerhard Goldbeck>:
   | .  uuid: a5b9282a-ec10-462d-9aa1-9671d8bbe236
   | .  age: 25
   | -  city.Citizen cuds object named <Georg Schmitz>:
   | .  uuid: c7c87209-660f-4a54-9c37-7e50c3164bc9
   | .  age: 25
   | -  city.Citizen cuds object named <Anne de Baas>:
   |    uuid: d74cfbae-9699-4998-a1e2-8f495a874ced
   |    age: 25
   |_Relationship city.hasPart:
     -  city.Neighborhood cuds object named <Ontology>:
     .  uuid: 26c4767d-c0ea-4abb-b7b7-7e8702de5de3
     .  coordinates: [0 0]
     .   |_Relationship city.hasPart:
     .     -  city.Street cuds object named <Relationships>:
     .     .  uuid: 23b0ba0d-1601-4824-b6c7-7eb3fdc05a91
     .     .  coordinates: [0 0]
     .     -  city.Street cuds object named <Entities>:
     .        uuid: b69d40d0-b919-4df8-8334-b898e4beda83
     .        coordinates: [0 0]
     -  city.Neighborhood cuds object named <User cases>:
        uuid: 79a214f6-4eb1-4a3b-8908-306129583da1
        coordinates: [0 0]
```

6.3.2 Cuds2Dot

Location: `from osp.core.utils import Cuds2dot`.

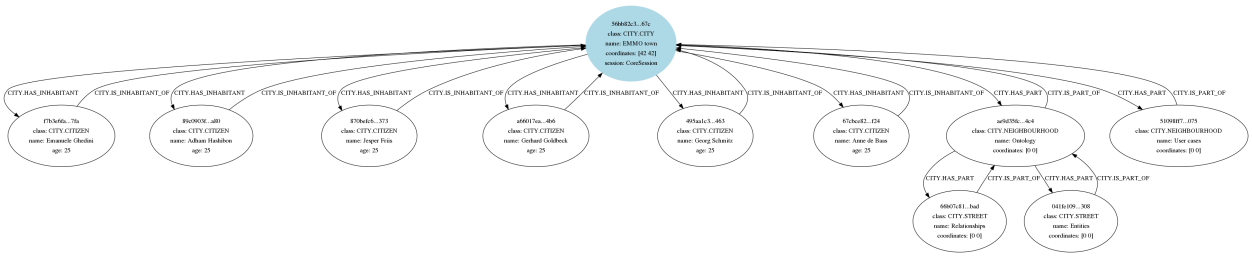
Usage: `Cuds2dot(cuds_object).render()`

Behaviour:

- Each entity is represented by a node.
- The relationships are the edges connecting them.
- The attributes, uuid and oclass are written inside the nodes.

Example:

```
>>> Cuds2dot(emmo_town).render()
```



6.3.3 Ontology2Dot

Location: console entry point `osp.core.tools.ontology2dot`.

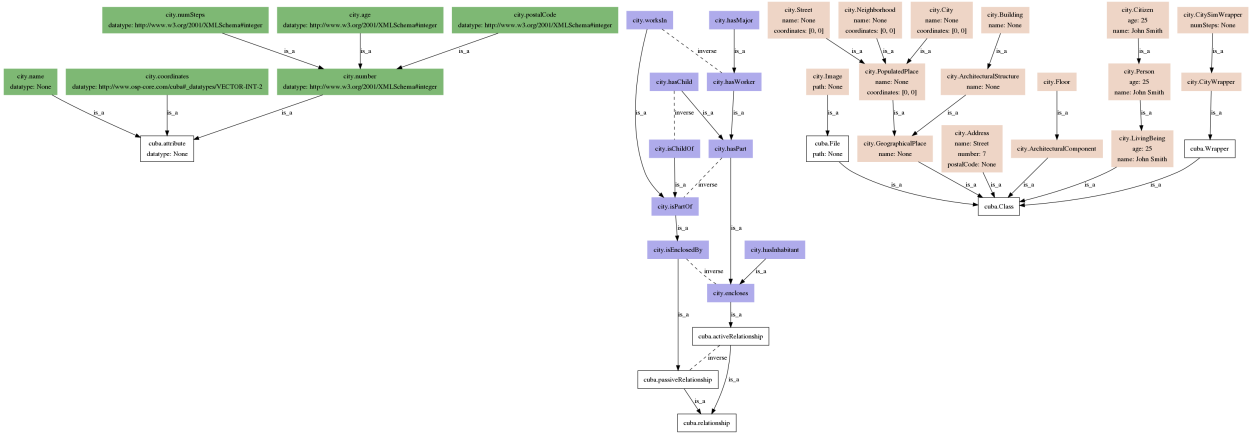
Usage: `ontology2dot`

Behaviour:

- Each ontology entity is represented by a box.
- Attributes and their default values are stated.
- Inheritance of entities is shown via *is_a*.
- The inverse of each relationship is also represented.

Example:

```
ontology2dot osp/core/ontology/yml/ontology.city.yml
```



6.4 Search

To make searching (in-depth) for a particular cuds object easier, we have implemented some simple search utility functions.

6.4.1 Simple search

Location: `import osp.core.utils.simple_search`

Usage:

- `find_cuds_object(criterion, root, rel, find_all, max_depth=float("inf"), current_depth=0, visited=None)` finds a cuds object under `root`, related via `rel` that returns `True` for `criterion` (boolean function). If `find_all` is set to `True`, it will return all elements, and not only the first found.
- `find_cuds_object_by_uid(uid, root, rel)` finds an element with given uid inside a cuds object by considering the given relationship.
- `find_cuds_objects_by_oclass(oclass, root, rel)` finds an element with given oclass inside a cuds object by considering the given relationship.
- `find_cuds_objects_by_attribute(attribute, value, root, rel)` finds a cuds object by attribute and value by only considering the given relationship.
- `find_relationships(find_rel, root, consider_rel, find_sub_rels=False)` finds the given relationship in the subtree of the given root.

Examples:

- To find all the inhabitants in a city with a given name:

```
queried_name = 'Pablo'
search.find_cuds_object(criterion = lambda x: queried_name in x.name,
                        root=city_cuds,
                        rel=city.hasInhabitant,
                        find_all=True)
```

- To find an object when the uid and relationship are known:

```
queried_uid = uuid.uuid4()
search.find_cuds_object_by_uid(uid=queried_uid,
                               root=city_cuds,
                               rel=city.get_default_rel())
```

- To find all the streets that are part of a city:

```
search.find_cuds_objects_by_oclass(oclass=city.Street,
                                   root=city_cuds,
                                   rel=city.hasPart)
```

- To find all the inhabitants with an attribute `age` with value `26`:

```
search.find_cuds_objects_by_attribute(attribute='age',
                                      value=26,
                                      root=city_cuds,
                                      rel=city.hasInhabitant)
```

6.5 Serialization JSON schema of CUDS objects

When you serialize a CUDS object using the `serialize()` method in the utils module, you will get a json document as a result. The method will traverse the hierarchical datastructure using Depth First Traversal. Therefore, its result is a json array composed of several flat CUDS objects.

This array can later be deserialized using the opposite `deserialize`.

The serialization is done via JSON-LD, with the schema used for the OSP API in Marketplace.

CHAPTER

SEVEN

TUTORIAL: CUDS API

This tutorial introduces the CUDS API. The code given here is based on this example.

Note that that this tutorial, as all others in this documentation, are Jupyter notebooks that can be downloaded by clicking on the "Edit on Github" button on the top right of the page.

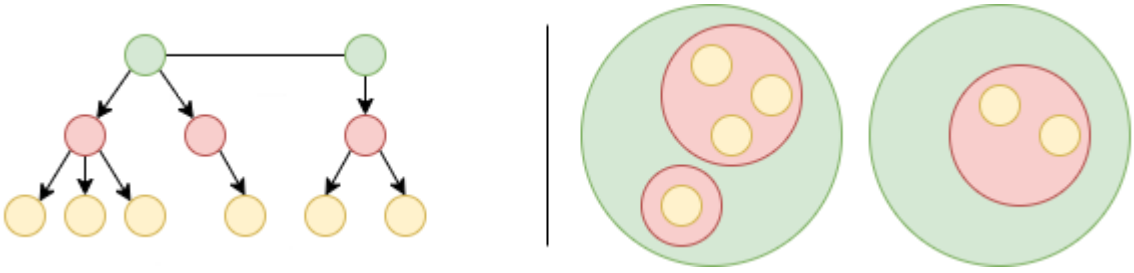
7.1 Background

CUDS stands for Common Universal Data Structure, and it is used to uniformly represent ontology individuals. In the python implementation of OSP-core, it means that every ontology individual is an instance of the `Cuds` class.

As every CUDS object is an ontology individual, each CUDS is related to an ontology class via the ontological `is a` relation, and can be connected to other CUDS objects through ontology relationships. In the python implementation of OSP-core, all such ontological classes are instances of the `OntologyClass` class, which is itself a subclass of `OntologyEntity`. The ontology entitites are organized in namespaces.

In OSP-core, the ontology relationships can be tagged as active or passive relationships. This is done in the ontology installation file. Such feature lets CUDS objects act as containers, so that content of a CUDS object consists of other CUDS objects. This means that a CUDS is a recursive data structure. Such active and passive relationships are directed, meaning that they have a source and a target. If a CUDS is the source of a connection via an active relationship to another CUDS, then the former contains the latter. Conversely, if a passive relationship is used, then the latter is contained in the former. Untagged ontology relationships do not define any containment.

Note: currently, each time a source CUDS object is connected to another target CUDS object through an active relationship, an inverse passive relationship is also created, connecting the target CUDS (acting as source) and the source CUDS (acting as target).



Containment in CUDS objects. On the graph view of the left hand side, the arrows depict active relationships, while the segment depicts any other untagged ontology relationship. On the right hand side, a containment view is provided.

The most important functionalities that the CUDS data structure exposes as python methods are the following:

- `add`: Connects the current CUDS object to one or more CUDS objects through a specific ontology relationship. If the chosen relationship is not an active relationship, one CUDS will not contain the other nor viceversa.
- `remove`: Despite its name, it does NOT delete the CUDS object itself. Instead, it just disconnects the current CUDS object from one or more CUDS objects.
- `get`: Returns the CUDS objects connected to the current CUDS object through a specific ontology relationship.
- `iter`: Similar to the `get` method, it just returns one CUDS objects at a time instead of all at once (python iterator), so that memory can be saved.
- `is_a`: Checks if the CUDS object is an instance of the given ontology class.

In addition, other important functionalities are exposed as python properties:

- `oclass`: The ontology class of the ontology individual represented by the CUDS object. If the individual belongs to multiple classes, only one of them is referenced by the property.
- `uid`: A unique ID identifying an ontology individual.
- `iri`: The Internationalized Resource Identifier of the CUDS object. It consists of a CUDS namespace prefix and the unique ID of the CUDS object. This will be further clarified in the tutorial.
- `attributes`: The values of the ontology attributes of an individual (also known as data properties in OWL) may also be accessed and modified as python properties of the CUDS objects. For example: `cuds_object.name = "Bob"`.

There are some advanced functionalities NOT covered in this tutorial. Among them, we highlight the `update` method, which is covered in the *wrapper tutorial*, where it can be seen in action. For a complete list of available methods and properties, check the API reference. That whole set of methods and attributes constitutes the CUDS API.

7.2 Let's get hands on

In this tutorial, we will work with the `city` namespace, the example namespace from OSP-core. It consists of concepts from the example city ontology.

The first step is to install the city ontology. Use the tool pico for this. If you want to know more about ontology installation, check the documentation on the pico ontology installation tool, YAML ontology installation files, and installing OWL ontologies.

[1]:
```
!pico install city
```

```
INFO 2021-04-01 13:40:41,433 [osp.core.ontology.installation]: Will install the␣
↪following namespaces: ['city']
INFO 2021-04-01 13:40:41,448 [osp.core.ontology.yml.yml_parser]: Parsing YAML ontology␣
↪file /home/jose/.local/lib/python3.9/site-packages/osp/core/ontology/docs/city.
↪ontology.yml
INFO 2021-04-01 13:40:41,476 [osp.core.ontology.yml.yml_parser]: You can now use `from␣
↪osp.core.namespaces import city`.
INFO 2021-04-01 13:40:41,476 [osp.core.ontology.parser]: Loaded 202 ontology triples in␣
↪total
INFO 2021-04-01 13:40:41,491 [osp.core.ontology.installation]: Installation successful
```

Then you can import the `city` namespace.

[2]:
```
# If you just installed the ontology from within this notebook and this line doesn't␣
↪work, please restart the kernel and run this cell again.
from osp.core.namespaces import city
```

You are now creating some CUDS objects that you are going to use to try out the functionalities of the CUDS data structure.

[3]:
```
c = city.City(name="Freiburg", coordinates=[47, 7])  # Ontology individual representing␣
↪the city of Freiburg.
p1 = city.Citizen(name="Peter")  # Ontology indidual representing a specific person,
↪"Peter".
p2 = city.Citizen(name="Anne")  # Ontology individual representing another specific␣
↪person, "Anne".
```

The names `c`, `p1`, `p2` are assigned to the newly created CUDS objects. The keyword arguments `name` and `coordinates` let you directly assign values for such ontology attributes (also known as data properties in OWL) to the new CUDS objects. The available ontology attributes for each ontology class depend on the specific class being instantiated. For example, the ontology attribute `name` is available for both the *City* and the *Citizen* ontology classes in the sample *City* ontology. The attribute `coordinates` is available for the *City* ontology class, but not for the *Citizen* class.

7.2.1 Functionalities exposed as python properties

Each CUDS object has a unique identifier (UID), which can be accessed using the `uid` property:

[4]:
```
print("uid of c: " + str(c.uid))
print("uid of p1: " + str(p1.uid))
print("uid of p2: " + str(p2.uid))
```

```
uid of c: e0b721ae-6004-4834-80f1-e6e979952d1f
uid of p1: 63874785-0de6-43ad-8669-999482501ad1
uid of p2: 39c7334f-ef62-4dbc-ae2e-390d7f3ca641
```

Similarly, each CUDS object has an IRI, which serves to reference it in the Semantic Web and improves the compatibility of the CUDS format with the Resource Description Framework data model. Note that the IRI of each CUDS object contains its unique identifier.

[5]:
```
print("IRI of c: " + str(c.iri))
print("IRI of p1: " + str(p1.iri))
print("IRI of p2: " + str(p2.iri))
```

```
IRI of c: http://www.osp-core.com/cuds#e0b721ae-6004-4834-80f1-e6e979952d1f
IRI of p1: http://www.osp-core.com/cuds#63874785-0de6-43ad-8669-999482501ad1
IRI of p2: http://www.osp-core.com/cuds#39c7334f-ef62-4dbc-ae2e-390d7f3ca641
```

The class of the ontology individual represented by the CUDS object can be queried as well:

[6]:
```
print("oclass of c: " + str(c.oclass))
print("oclass of p1: " + str(p1.oclass))
print("oclass of p2: " + str(p2.oclass))
```

```
oclass of c: city.City
oclass of p1: city.Citizen
oclass of p2: city.Citizen
```

The `uid`, `iri` and `oclass` properties **cannot be modified**.

Finally, the values of the ontology attributes of an individual can be easily **accessed and modified** using the dot notation.

[7]:
```
print(f"Name of c: {c.name}. Coordinates of c: {c.coordinates}." )
print("Name of p1: " + str(p1.name))
print("Name of p2: " + str(p2.name))

print(f"\nChange the name of {p1.name}.")
p1.name = "Bob"
print(f"Name of p1: {p1.name}.")
```

```
Name of c: Freiburg. Coordinates of c: [47  7].
Name of p1: Peter
Name of p2: Anne

Change the name of Peter.
Name of p1: Bob.
```

7.2.2 Functionalities exposed as python methods

Now, we may connect the two citizens to our city object:

[8]:
```
c.add(p1, rel=city.hasInhabitant)
c.add(p2, rel=city.hasInhabitant)
```

[8]:
```
<city.Citizen: 39c7334f-ef62-4dbc-ae2e-390d7f3ca641,  CoreSession: @0x7f3a5778ef70>
```

Note that the relationship type between the city and its two citizens in this case is 'hasInhabitant'. In our context, this means that Anne and Peter are Freiburg inhabitants. Moreover, in the city ontology, this relationship is defined as an active relationship. This means that Anne and Peter are not only connected to Freiburg, but are also contained in the Freiburg CUDS object.

Next, we would like to iterate over the objects contained in the city object. We do so by using the `iter` function:

[9]:
```
for el in c.iter():
    print("uid: " + str(el.uid))
```

```
uid: 63874785-0de6-43ad-8669-999482501ad1
uid: 39c7334f-ef62-4dbc-ae2e-390d7f3ca641
```

We can `get` a target object from a CUDS object if we have a UID of one of its immediate contained objects. This will not work if the target object is not contained in the CUDS object, but just connected to it.

[10]:
```
print(c.get(p1.uid))  # `p1`is contained in `c` because they are connected through an␣
↪active relationship.
print(p1.get(c.uid))  # `c` is connected to `p1`, but it is NOT contained in `p1`.
```

```
city.Citizen: 63874785-0de6-43ad-8669-999482501ad1
None
```

We can also filter the contained objects by type:

[11]:
```
print(c.get(oclass=city.Citizen))
```

```
[<city.Citizen: 63874785-0de6-43ad-8669-999482501ad1,  CoreSession: @0x7f3a5778ef70>,
↪<city.Citizen: 39c7334f-ef62-4dbc-ae2e-390d7f3ca641,  CoreSession: @0x7f3a5778ef70>]
```

We remove objects using the `remove()` function. Despite its name, this just disconnects the target object from the CUDS, but does NOT delete the target object from the memory.

[12]:
```
c.remove(p1)
# c.remove(p1.uid) also works!
print(p1)  # `p1` still exists,
print(c.get(p1.uid))  # but is no longer connected neither contained in `c`.
```

```
city.Citizen: 63874785-0de6-43ad-8669-999482501ad1
None
```

Let's close this tutorial by adding some neighborhoods in a loop,

[13]:
```
for i in range(6):
    c.add(city.Neighborhood(name="neighborhood %s" % i))
```

and then verifying that they are indeed neighborhoods, just to also try the `is_a` method.

[14]:
```
all(n.is_a(city.Neighborhood) for n in c.get(oclass=city.Neighborhood))
```

[14]:
```
True
```

The existing ontology individuals and the relationships among them have been depicted below, using the utility `pretty_print` (see the Utilities section). Note that some attributes that were not specified were set automatically to the default values specified in the ontology.

[15]:
```
from osp.core.utils import pretty_print
pretty_print(c)
```

```
- Cuds object named <Freiburg>:
  uuid: e0b721ae-6004-4834-80f1-e6e979952d1f
  type: city.City
  superclasses: city.City, city.GeographicalPlace, city.PopulatedPlace, cuba.Entity
  values: coordinates: [47  7]
  description:
    To Be Determined

   |_Relationship city.hasInhabitant:
   | -  city.Citizen cuds object named <Anne>:
   |    uuid: 39c7334f-ef62-4dbc-ae2e-390d7f3ca641
   |    age: 25
   |_Relationship city.hasPart:
     -  city.Neighborhood cuds object named <neighborhood 0>:
     .  uuid: 79eec4f9-55c5-464d-b6c3-dc4382e55476
     .  coordinates: [0 0]
     -  city.Neighborhood cuds object named <neighborhood 1>:
     .  uuid: 50fa8dac-2f5f-4b2c-ab8a-24f7f1b41f2c
     .  coordinates: [0 0]
     -  city.Neighborhood cuds object named <neighborhood 2>:
     .  uuid: f6c20bfa-c29b-46bb-a5f5-8bc88b3e3621
     .  coordinates: [0 0]
     -  city.Neighborhood cuds object named <neighborhood 3>:
     .  uuid: e161ea1c-40d8-4556-93ad-dfd4d178d669
     .  coordinates: [0 0]
     -  city.Neighborhood cuds object named <neighborhood 4>:
     .  uuid: a3af8db2-28cd-457b-84ed-b420be5212c2
     .  coordinates: [0 0]
     -  city.Neighborhood cuds object named <neighborhood 5>:
```

(continues on next page)

(continued from previous page)

```
        uuid: 8de47383-4416-44be-99dc-e5903b1d14dd
        coordinates: [0 0]
```

CHAPTER

EIGHT

TUTORIAL: SESSIONS AND VARIABLES

In this tutorial we will explain how objects and the variables assigned to them behave with different sessions. This tutorial is not available on binder since `wrapper_x` and `wrapper_y` are fictional implementations for educational purposes.

8.1 Background

Some use cases of OSP-core, like coupling and linking, require interaction between multiple sessions. Even using just one wrapper usually means working with the default CoreSession initially, and then the wrapper session.

When an object from one session is added to a different one, a deepcopy of the object is carried out. This means that both objects are technically the same at that point (same uid, oclass, relationships...), but reside in different memory areas and can theoretically differ in the future. The purpose of this behaviour is to allow selective synching and enable different sessions to have different states of an instance.

Variables that point to an object in the origin session will keep pointing to it (and not to the added session) unless explicitedly updated.

Here we will try to explain said behaviour with simple illustrative examples.

Note: This tutorial is not meant to be run like the others. The session classes and ontology entities are not real implementations. However, the behaviour shown will be the same as that of a real setup.

8.2 Let's show an example

We start by importing the necessary components, namely the session and a sample namespace:

[ ]:
```
from osp.core.namespaces import namespace
from osp.wrappers.wrapper_x import SessionX
from osp.wrappers.wrapper_y import SessionY
```

We can now instantiate the Session objects and bind them to wrapper instances:

[ ]:
```
sess_x = SessionX()
sess_y = SessionY()

wrapper_x = namespace.WrapperX(session=sess_x)
wrapper_y = namespace.WrapperY(session=sess_y)
```

Next, we will add some entities to the wrappers. For simplicity, we will use the `session` parameter available to all entities in their initialisation. This is just to explicitly work with `SessionX` and `SessionY`, without worrying about the default `CoreSession`.

[ ]:
```
a = namespace.A(session=sess_x, name="a")
wrapper_x.add(a)

b = namespace.B(session=sess_y, name="b")
wrapper_y.add(b)
```

Let's add `a` to `wrapper_y`:

[ ]:
```
wrapper_y.add(a)
```

After the previous statement, both wrappers have an identical version of a. However, they are not linked together. This means one can be changed and the other one won't be. Also, the variable `a` points to the instance inside `wrapper_x`, and there is no reference to the one inside `wrapper_y`. We can test that:

[4]:
```
a.name = "a updated"

a_in_y = wrapper_y.get(a.uid)

print("Name of a: ", a.name)
print("Name of a_in_y: ", a_in_y.name)
```

```
Name of a: a updated
Name of a_in_y: a
```

As you can see, changing the name through `a`, which points to the object in `SessionX`, only changed one version. Also bear in mind that `SessionX` and `SessionY` represent two arbitrary sessions, so this situation could arise when adding objects to a wrapper from the normal instantiation (remember that objects reside in the `CoreSession` by default).

In order to get a reference to the added object, you can assign the return of the call to `add` to a variable. For example:

[ ]:
```
b = wrapper_x.add(b)
```

That way we can modify the name of `b` in `wrapper_x` more easily:

[5]:
```
b.name = "b updated"

b_in_y = wrapper_y.get(b.uid)

print("Name of b: ", b.name)
print("Name of b_in_y: ", b_in_y.name)
```

```
Name of b: b updated
Name of b_in_y: b
```

8.3 Summary

In short, these are the things you should be aware of:

- Adding an object to a different session creates a deepcopy.
- Two versions of the same instance (same uid and type) in different sessions are not automatically synced.
- Variables point to an object in a session, and will not change when the objects are added somewhere else.
- If you want to quickly assign a variable to an object in a new session, you can capture the return of the `add` call.

CHAPTER

NINE

TUTORIAL: MULTIPLE WRAPPERS

This tutorial introduces the use of multiple data sources, and shows how can one exchange information between them. The code given here is based on this example and builds on the *introduction on the CUDS API*.

9.1 Background

One of the main strengths of CUDS objects is their ability to share information between different underlying data sources interchangeably. Using OSP-core's inner workings a data source can be represented as a CUDS object. A data source can be in turn a database, a simulation engine, or any other software package, which is able to either generate or store information.

We refer to a CUDS object, which represents an underlying data source as a **wrapper**, as it wraps around the data source. Wrappers use the CUDS API with the addition to some wrapper-specific methods, which will be discussed later on in this tutorial.

For a wrapper to be initialized, one needs some context for the underlying data source (e.g. location, credentials, etc.) for this we introduce an object called **session**. Conceptually a session can be thought as an interoperability level, or in simple terms it handles the transition from the user-friendly CUDS API to the more task-specific syntax data sources tend to have.

9.2 Let's get hands on

We start by importing the example namespace of OSP-core. If you haven't already, you should install the city ontology before:

[1]:
```
!pico install city
```

```
INFO 2020-12-02 11:54:26,244 [osp.core.ontology.installation]: Will install the␣
↪following namespaces: ['city']
INFO 2020-12-02 11:54:26,280 [osp.core.ontology.yml.yml_parser]: Parsing YAML ontology␣
↪file /mnt/c/Users/dea/Documents/Projects/simphony/osp-core/osp/core/ontology/docs/city.
↪ontology.yml
INFO 2020-12-02 11:54:26,331 [osp.core.ontology.yml.yml_parser]: You can now use `from␣
↪osp.core.namespaces import city`.
INFO 2020-12-02 11:54:26,333 [osp.core.ontology.parser]: Loaded 403 ontology triples in␣
↪total
INFO 2020-12-02 11:54:26,374 [osp.core.ontology.installation]: Installation successful
```

[2]:
```
from osp.core.namespaces import city
```

The `pretty_print` function is part of our utilities module and is a convinient way to output the tree-like structure of a CUDS object.

[3]:
```
from osp.core.utils import pretty_print
```

The `getpass` function is used to retrieve an input from an user. It prints a prompt, then reads input from the user until they press return.

[4]:
```
from getpass import getpass
```

The next statements imports the second data source session we will use in this example, namely a database, or more precisely the ORM toolkit SQLAlchemy which we will use in turn to connect to a PostgreSQL database. It will throws an error if it cannot find the `SQLAlchemyWrapperSession` as it needs to be installed from a separate repository. Please refer here for installation instructions.

[5]:
```
from osp.wrappers.sqlalchemy import SqlAlchemySession
```

Next we import a session object, which will provide the context to a simple simulation engine we developed for demonstrational purposes. It is already included in OSP-core.

[6]:
```
from osp.wrappers.simdummy import SimDummySession
```

The following lines prompt the user to enter the information needed to create a connection to a running instance of a PostgreSQL database, where the data of our simulation will be stored. Please make sure to point to an existing and running instance of PostgreSQL. To install PostgreSQL on your machine, please refer to their documentation.

The information is stored in a string `postgres_url`, which will later be passed to the `SQLAlchemyWrapperSession` object to initiate a connection with the data base.

[7]:
```
print("Input data to connect to Postgres table!")
user = input("User: ")
pwd = getpass("Password: ")
db_name = input("Database name: ")
host = input("Host: ")
port = int(input("Port [5432]: ") or 5432)
postgres_url = 'postgres://%s:%s@%s:%s/%s' % (user, pwd, host, port, db_name)
```

```
Input data to connect to Postgres table!
```

In the next lines we create our small ontology-loving **EMMO town** example. Please pay a closer attention to the fourth line as there you can see the power of the `add` method and how with one statement one can add multiple `CITIZEN` CUDS objects simultaneously.

[8]:
```
emmo_town = city.City(name='EMMO town')

emmo_town.add(city.Citizen(name='Emanuele Ghedini'), rel=city.hasInhabitant)
emmo_town.add(city.Citizen(name='Adham Hashibon'), rel=city.hasInhabitant)
emmo_town.add(city.Citizen(name='Jesper Friis'),
              city.Citizen(name='Gerhard Goldbeck'),
              city.Citizen(name='Georg Schmitz'),
              city.Citizen(name='Anne de Baas'),
              rel=city.hasInhabitant)

emmo_town.add(city.Neighborhood(name="Ontology"))
emmo_town.add(city.Neighborhood(name="User cases"))
```

[8]:
```
<city.Neighborhood: 034f396c-8b0a-411e-873e-482935688386,  CoreSession: @0x7f4ede777950>
```

Next we grow the `Ontology` neighbourhood by adding some streets to it: namely *relationships* and *entities* (puns are intended).

[9]:
```
ontology_uid = None
for neighbourhood in emmo_town.get(oclass=city.Neighborhood):
    if neighbourhood.name == "Ontology":
        ontology_uid = neighbourhood.uid
        neighbourhood.add(city.Street(name="Relationships"), rel=city.hasPart)
        neighbourhood.add(city.Street(name="Entities"), rel=city.hasPart)
```

Whenever you add a relationship to a CUDS object, OSP-core will always automatically add the `inverse` relationship. In our example case we can now retrieve from the "Ontology" neighbourhood `onto`, which town it belongs to.

[10]:
```
onto = emmo_town.get(ontology_uid)
print(onto.get(rel=city.isPartOf)[0].name + ' is my city!')
```

```
EMMO town is my city!
```

Let's now store our small city persistently in the PostgreSQL database. Working with session objects is similar to the way one is used to work with files in Python. And when you ponder a bit about it, it makes kind of sense, as what one intends to do is store some information in a data storage.

Using Python's `with` statement the connection to the database will be maintained only within the scope of the `with` statement. After exiting its scope the connection will be closed automatically. We advise the use of the `with` statement as it automatically manages opening and closing of database connections.

First we open a connection to a PostgreSQL database through our `SqlAlchemyWrapperSession` and assign it to the session variable. Then we assign to the `wrapper` variable a `CITY_WRAPPER` and pass it the needed context with the `session=session`. From there on, we treat the `wrapper` variable as a normal CUDS object with the only difference that its internal state is managed by the `SqlAlchemyWrapperSession` behind the scenes. On the last line, we see the benefits of that by simply executing the `commit` command onto the `session` object. This will trigger a series of events, with the end result being that our **EMMO town** will be stored in the database.

[12]:
```
with SqlAlchemySession(postgres_url) as session:
    wrapper = city.CityWrapper(session=session)
    wrapper.add(emmo_town)
    session.commit()
```

Next we show how one can use multiple data source wrappers simultaneously. First we open a connection to the PostgreSQL database through `SqlAlchemyWrapperSession` as shown above and assign it to the `db_session` variable. Then we retrieve the information we have previously stored to it using CUDS' `get` method and use the `pretty_print` function to output the information.

Then we open a connection to our demonstrational simulation engine through `DummySimWrapperSession` and assign it to the `sim_session` variable. As you can see opening the second simulation session is within the scope of the `SqlAlchemyWrapperSession`.

In the scope of `DummySimSession`, we initialize a `CitySimWrapper`, pass it the context from the `sim_session` and assign it to the `sim_wrapper` variable. The `CitySimWrapper` consumes a city and a person. The magic happens in the following `run` method, which recognise that **Peter** is a person and it then transforms him into a citizen of the **EMMO town**. This new information is then stored in the `sim_emmo_town` automatically within the `run` method. We then output the information about Peter, who is now a citizen of **EMMO town**.

Finally we `update` our now outdated **EMMO town** in the database by using the `update` command. It checks for any inconsistencies between the **EMMO town** stored in the database, `db_emmo_town`, and the modified by our simulation

engine town, `sim_emmo_town`. In our case it will find its new citizen **Peter** and it will add it to the **EMMO town** in the database instance of the town, `db_emmo_town`. This change is then in turn made persistent by calling the `commit` method, which will actually store the information the database.

[12]:
```
with SqlAlchemySession(postgres_url) as db_session:
    db_wrapper = city.CityWrapper(session=db_session)
    db_emmo_town = db_wrapper.get(emmo_town.uid)
    print("The database contains the following information about the city:")
    pretty_print(db_emmo_town)

    # Working with a Simulation wrapper
    with SimDummySession() as sim_session:
        sim_wrapper = city.CitySimWrapper(numSteps=1,
                                          session=sim_session)
        new_inhabitant = city.Person(age=31, name="Peter")
        sim_emmo_town, _ = sim_wrapper.add(db_emmo_town, new_inhabitant)
        sim_session.run()
        print("The city has a new inhabitant:")
        pretty_print(sim_emmo_town.get(new_inhabitant.uid))

    # update database
    db_wrapper.update(sim_emmo_town)
    db_session.commit()
```

```
The database contains the following information about the city:
- Cuds object named <EMMO town>:
  uuid: 96a2f49c-8009-456a-ad69-1cb1dea7f128
  type: city.City
  superclasses: city.City, city.GeographicalPlace, city.PopulatedPlace, cuba.Entity
  values: coordinates: [0 0]
  description:
    To Be Determined

   |_Relationship city.hasInhabitant:
   | -  city.Citizen cuds object named <Emanuele Ghedini>:
   | .  uuid: 08e05374-0187-4114-a114-085431aeebde
   | .  age: 25
   | -  city.Citizen cuds object named <Adham Hashibon>:
   | .  uuid: a084c4ba-c054-4afd-b194-b4e983743706
   | .  age: 25
   | -  city.Citizen cuds object named <Jesper Friis>:
   | .  uuid: 209926bc-6bcd-466c-a86a-807780e4789c
   | .  age: 25
   | -  city.Citizen cuds object named <Gerhard Goldbeck>:
   | .  uuid: 0a0a8488-7400-4f4e-9e44-dbf56782cd3c
   | .  age: 25
   | -  city.Citizen cuds object named <Georg Schmitz>:
   | .  uuid: cdfe72ec-fc15-4a14-a516-88b576bf0a27
   | .  age: 25
   | -  city.Citizen cuds object named <Anne de Baas>:
   |    uuid: 4ec1b1af-341f-4aa5-845d-fb3f9c502c9c
   |    age: 25
   |_Relationship city.hasPart:
```

(continues on next page)

(continued from previous page)

```
    -  city.Neighborhood cuds object named <Ontology>:
    .  uuid: e1632cf5-24e3-4afb-b72b-757ef62d5bef
    .  coordinates: [0 0]
    .   |_Relationship city.hasPart:
    .     -  city.Street cuds object named <Relationships>:
    .     .  uuid: 19a6a5a4-c22c-4be7-98c4-c69b3ee613d2
    .     .  coordinates: [0 0]
    .     -  city.Street cuds object named <Entities>:
    .        uuid: 681f7441-e711-4375-ab70-074234206faa
    .        coordinates: [0 0]
    -  city.Neighborhood cuds object named <User cases>:
       uuid: 574e4a68-92c8-4b0e-b41f-e0341b42b00b
       coordinates: [0 0]
The city has a new inhabitant:
- Cuds object named <Peter>:
  uuid: 7a263d53-9c4d-47ac-bb31-fa54d7920bcd
  type: city.Citizen
  superclasses: city.Citizen, city.LivingBeing, city.Person, cuba.Entity
  values: age: 32
  description:
    To Be Determined
```

Finally to ensure our database has successfully interpreted our addition to **Emmo town**, we check its contents and print them using `pretty_print`.

[13]:
```
with SqlAlchemySession(postgres_url) as db_session:
    db_wrapper = city.CityWrapper(session=db_session)
    db_emmo_town = db_wrapper.get(emmo_town.uid)
    print("The database contains the following information about the city:")
    pretty_print(db_emmo_town)
```

```
The database contains the following information about the city:
- Cuds object named <EMMO town>:
  uuid: 96a2f49c-8009-456a-ad69-1cb1dea7f128
  type: city.City
  superclasses: city.City, city.GeographicalPlace, city.PopulatedPlace, cuba.Entity
  values: coordinates: [0 0]
  description:
    To Be Determined

   |_Relationship city.hasInhabitant:
   | -  city.Citizen cuds object named <Emanuele Ghedini>:
   | .  uuid: 08e05374-0187-4114-a114-085431aeebde
   | .  age: 25
   | -  city.Citizen cuds object named <Adham Hashibon>:
   | .  uuid: a084c4ba-c054-4afd-b194-b4e983743706
   | .  age: 25
   | -  city.Citizen cuds object named <Jesper Friis>:
   | .  uuid: 209926bc-6bcd-466c-a86a-807780e4789c
   | .  age: 25
   | -  city.Citizen cuds object named <Gerhard Goldbeck>:
   | .  uuid: 0a0a8488-7400-4f4e-9e44-dbf56782cd3c
```

(continues on next page)

(continued from previous page)

```
| .  age: 25
| -  city.Citizen cuds object named <Georg Schmitz>:
| .  uuid: cdfe72ec-fc15-4a14-a516-88b576bf0a27
| .  age: 25
| -  city.Citizen cuds object named <Anne de Baas>:
| .  uuid: 4ec1b1af-341f-4aa5-845d-fb3f9c502c9c
| .  age: 25
| -  city.Citizen cuds object named <Peter>:
|    uuid: 7a263d53-9c4d-47ac-bb31-fa54d7920bcd
|    age: 32
|_Relationship city.hasPart:
  -  city.Neighborhood cuds object named <Ontology>:
  .  uuid: e1632cf5-24e3-4afb-b72b-757ef62d5bef
  .  coordinates: [0 0]
  .   |_Relationship city.hasPart:
  .     -  city.Street cuds object named <Relationships>:
  .     .  uuid: 19a6a5a4-c22c-4be7-98c4-c69b3ee613d2
  .     .  coordinates: [0 0]
  .     -  city.Street cuds object named <Entities>:
  .        uuid: 681f7441-e711-4375-ab70-074234206faa
  .        coordinates: [0 0]
  -  city.Neighborhood cuds object named <User cases>:
     uuid: 574e4a68-92c8-4b0e-b41f-e0341b42b00b
     coordinates: [0 0]
```

[ ]:

CHAPTER

TEN

TUTORIAL: IMPORT AND EXPORT

In this tutorial we will be covering the import and export capabilities of OSP-core. The utility functions that provide these functionalities are `import_cuds` and `export_cuds`, respectively.

Tip

The full API specifications of the import and export functions can be found in the *utilities API reference page*.

For our running example, we'll be using the *city ontology* that was already introduces in the *cuds API tutorial*. First, make sure the city ontology is installed. If not, run the following command:

[1]:
```
!pico install city
```

```
INFO 2021-05-31 10:55:51,839 [osp.core.ontology.installation]: Will install the␣
↪following namespaces: ['city']
INFO 2021-05-31 10:55:51,879 [osp.core.ontology.yml.yml_parser]: Parsing YAML ontology␣
↪file c:\users\yoav\anaconda3\envs\osp\lib\site-packages\osp\core\ontology\docs\city.
↪ontology.yml
INFO 2021-05-31 10:55:51,995 [osp.core.ontology.yml.yml_parser]: You can now use `from␣
↪osp.core.namespaces import city`.
INFO 2021-05-31 10:55:51,996 [osp.core.ontology.parser]: Loaded 7396 ontology triples in␣
↪total
INFO 2021-05-31 10:55:52,437 [osp.core.ontology.installation]: Installation successful
```

Next we create a few CUDS objects:

[2]:
```
from osp.core.namespaces import city

c = city.City(name="Freiburg", coordinates=[47, 7])
p1 = city.Citizen(name="Peter")
p2 = city.Citizen(name="Anne")
c.add(p1, rel=city.hasInhabitant)
c.add(p2, rel=city.hasInhabitant)
```

[2]:
```
<city.Citizen: 98f8ac8c-713d-4406-bc36-e0152f9e2ea3,  CoreSession: @0x221b0a05250>
```

Now we can use the `export_cuds` methods to export the data into a file:

[3]:
```
from osp.core.utils import export_cuds

export_cuds(c, file='./data.ttl', format='turtle')
```

This will create the file `data.ttl` with the following content:

[4]:
```
from sys import platform

if platform == 'win32':
    !more data.ttl
else:
    !cat data.ttl
```

```
@prefix city: <http://www.osp-core.com/city#> .
@prefix cuba: <http://www.osp-core.com/cuba#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

cuba:_serialization rdf:first "47398674-720b-4765-9047-b5351ed175c0" .

<http://www.osp-core.com/cuds#8a90e2b3-7cca-4103-9eba-aab55e5903b1> a city:Citizen ;
    city:INVERSE_OF_hasInhabitant <http://www.osp-core.com/cuds#47398674-720b-4765-9047-
↪b5351ed175c0> ;
    city:age 25 ;
    city:name "Peter" .

<http://www.osp-core.com/cuds#98f8ac8c-713d-4406-bc36-e0152f9e2ea3> a city:Citizen ;
    city:INVERSE_OF_hasInhabitant <http://www.osp-core.com/cuds#47398674-720b-4765-9047-
↪b5351ed175c0> ;
    city:age 25 ;
    city:name "Anne" .

<http://www.osp-core.com/cuds#47398674-720b-4765-9047-b5351ed175c0> a city:City ;
    city:coordinates "[47, 7]"^^<http://www.osp-core.com/cuba#_datatypes/VECTOR-INT-2> ;
    city:hasInhabitant <http://www.osp-core.com/cuds#8a90e2b3-7cca-4103-9eba-
↪aab55e5903b1>,
        <http://www.osp-core.com/cuds#98f8ac8c-713d-4406-bc36-e0152f9e2ea3> ;
    city:name "Freiburg" .
```

You can change the format by entering a different value for the parameter `format`. The supported formats are “xml”, “n3”, “turtle”, “nt”, “pretty-xml”, “trix”, “trig” and “nquads”.

To import data, we can use the `import` method. Let’s assume we wish to import data into an SQLite session. The following code will help us to achieve our aim:

[5]:
```
from osp.wrappers.sqlite import SqliteSession
from osp.core.utils import import_cuds

with SqliteSession("test.db") as session:
    wrapper = city.CityWrapper(session=session)
    c = import_cuds('./data.ttl')
    wrapper.add(c)
    session.commit()
```

Now we can verify the data was indeed imported:

[6]:
```
from osp.core.utils import pretty_print

with SqliteSession("test.db") as session:
```

(continues on next page)

(continued from previous page)

```
    wrapper = city.CityWrapper(session=session)
    pretty_print(wrapper)
```

```
- Cuds object:
  uid: 03015cb9-f88c-4ab1-9df9-bb52743b99de
  type: city.CityWrapper
  superclasses: city.CityWrapper, cuba.Entity, cuba.Wrapper
  description:
    To Be Determined

   |_Relationship city.hasPart:
     -  city.City cuds object named <Freiburg>:
     .  uid: 72595bc4-1b68-46a3-97e9-8f3de2650f2c
     .  coordinates: [47  7]
     .   |_Relationship city.hasInhabitant:
     .     -  city.Citizen cuds object named <Anne>:
     .     .  uid: 92a00459-0927-438c-a305-a26512ac7f03
     .     .  age: 25
     .     -  city.Citizen cuds object named <Peter>:
     .        uid: 27d1e83b-4ee9-4f4f-adb5-0b01a3cc2c1b
     .        age: 25
     -  city.City cuds object named <Freiburg>:
     .  uid: 47398674-720b-4765-9047-b5351ed175c0
     .  coordinates: [47  7]
     .   |_Relationship city.hasInhabitant:
     .     -  city.Citizen cuds object named <Anne>:
     .     .  uid: 98f8ac8c-713d-4406-bc36-e0152f9e2ea3
     .     .  age: 25
     .     -  city.Citizen cuds object named <Peter>:
     .        uid: 8a90e2b3-7cca-4103-9eba-aab55e5903b1
     .        age: 25
     -  city.City cuds object named <Freiburg>:
        uid: d886f8ce-1326-40f5-a98b-c4c893b8c085
        coordinates: [47  7]
         |_Relationship city.hasInhabitant:
           -  city.Citizen cuds object named <Anne>:
           .  uid: 2b5d0a3f-81a5-4746-aab9-40adcb65e71f
           .  age: 25
           -  city.Citizen cuds object named <Peter>:
              uid: 766b320a-7e9a-43ec-a696-96b4f9ee494d
              age: 25
```

Notes

1. The format is automatically inferred from the file extension. To specify it explicitly, you can add the `format` parameter, like so: `import_cuds('./data.ttl', format='turtle')`.
2. The `session` parameter is optional and inferred automatically from the context that created by the `with` statement (see the *tutorial on multiple wrappers* for more information). You can specify the session explicitly like so: `import_cuds('./data.ttl', session=session)`.

CHAPTER

ELEVEN

TUTORIAL: SIMLAMMPS WRAPPER

In this tutorial we will go through a simple example of how to use the wrapper for the LAMMPS simulation engine. You can find the wrapper here.

11.1 Background

A wrapper for LAMMPS has been present in SimPhoNy since its initial version, and it is the first simulation engine we supported in version 3.

This wrapper is a good example of the *3-layered-design* where the Syntactic layer is a third party library. In this case we use PyLammps, a Python binding for LAMMPS created and maintained by the LAMMPS developers.

11.2 Let's get hands on

11.2.1 Installation

We will start by quickly going through the installation of this tool. Like we explain in the *wrapper development section*, there are two options:

- Using Docker: run `./docker_install.sh`.
- Local installation: remember that you must have a compatible version of *OSP-core installed*.

Install the ontology via `pico install simlammps.ontology.yml`.

Run `./install_engine.sh`.

- Note that you will be asked for a superuser password to install required libraries for the installation (make, libjpeg, libpng…)
- Currently we support Ubuntu and centOS.

Install the wrapper by running `python setup.py install`.

That should be all needed to use simlammps!

11.2.2 Simple example

This is an adaptation of simlammps/examples/small.py. As usual, we start importing the necessary components:

[1]:
```
from osp.core import simlammps_ontology
from osp.wrappers.simlammps import SimlammpsSession
```

We create the wrapper instance. All wrappers are created by defining their own *session class*.

There is no need to specify a syntactic layer (PyLammps). The session will generate one.

[2]:
```
simlammps_session = SimlammpsSession()
simlammps = simlammps_ontology.SimlammpsWrapper(session=simlammps_session)
```

```
LAMMPS output is captured by PyLammps wrapper
```

Next, we can define some necessary settings for the run:

[3]:
```
# Define the simulation box
box = simlammps_ontology.SimulationBox()
face_x = simlammps_ontology.FaceX(vector=(10, 0, 0))
face_x.add(simlammps_ontology.Periodic())
face_y = simlammps_ontology.FaceY(vector=(0, 10, 0))
face_y.add(simlammps_ontology.Periodic())
face_z = simlammps_ontology.FaceZ(vector=(0, 0, 10))
face_z.add(simlammps_ontology.Periodic())
box.add(face_x, face_y, face_z)
simlammps.add(box)

# molecular dynamics model
md_nve = simlammps_ontology.MolecularDynamics()
simlammps.add(md_nve)

# solver component:
sp = simlammps_ontology.SolverParameter()

# integration time:
steps = 100
itime = simlammps_ontology.IntegrationTime(steps=steps)

sp.add(itime)
verlet = simlammps_ontology.Verlet()

sp.add(verlet)
simlammps.add(sp)

# Mass and material for the atoms
mass = simlammps_ontology.Mass(value=0.2)
material = simlammps_ontology.Material()

material.add(mass)
simlammps.add(material)

# Interatomic force as material relation
```

(continues on next page)

(continued from previous page)

```
lj = simlammps_ontology.LennardJones_6_12(cutoff_distance=2.5,
                                          energy_well_depth=1.0,
                                          van_der_waals_radius=1.0)
lj.add(material)
simlammps.add(lj)
```

[3]:
```
<SIMLAMMPS_ONTOLOGY.LENNARD_JONES_6_12: 9c9c1672-a9f2-4eba-85a4-060b56addf2a, ␣
↪SimlammpsSession: @0x7fac60dc7a90>
```

Now we add some atoms:

[4]:
```
particle = simlammps_ontology.Atom()
particle.add(material,
             simlammps_ontology.Position(vector=(1, 6, 3)),
             simlammps_ontology.Velocity(vector=(1, 0, 0)))
simlammps.add(particle)

particle = simlammps_ontology.Atom()
particle.add(material,
             simlammps_ontology.Position(vector=(2, 1, 4)),
             simlammps_ontology.Velocity(vector=(2, 0, 2)))
simlammps.add(particle)

particle = simlammps_ontology.Atom()
# The velocity is not required (the position is)
particle.add(material,
             simlammps_ontology.Position(vector=(7, 3, 0)))
simlammps.add(particle)
```

[4]:
```
<SIMLAMMPS_ONTOLOGY.ATOM: db96a76f-4d70-4e19-b460-46ee286f831e,  SimlammpsSession:␣
↪@0x7fac60dc7a90>
```

To run the simulation, we call the `run()` method of the session. The run method sends the information to the engine, and tells it to run the number of steps defined in the Integration Time entity (100):

[5]:
```
simlammps.session.run()
```

Since we will run the simulation a couple of times, we can define a simple function for showing the position and velocities of the atoms:

[6]:
```
def print_info():
    for atom in simlammps.iter(oclass=simlammps_ontology.Atom):
        # Remember that Cuds.get(oclass) returns a list
        # We now all atoms have one (and only one) position
        position = atom.get(oclass=simlammps_ontology.Position)[0]
        # But the atoms might not have a velocity
        velocity = atom.get(oclass=simlammps_ontology.Velocity)
        print("Atom " + str(atom.uid) + ":")
        print(" - Position: " + str(position.vector))
        if velocity:
            print(" - Velocity: " + str(velocity[0].vector))
```

Now we can easily print the results of the run:

[7]:
```
print_info()
```

```
Atom fd4199d4-4d1a-425c-8010-60efca65bd1c:
 - Position: [1.5 6.  3. ]
 - Velocity: [1. 0. 0.]
Atom f9a32d14-b638-4796-9407-4b1ae6be43cb:
 - Position: [3. 1. 5.]
 - Velocity: [2. 0. 2.]
Atom db96a76f-4d70-4e19-b460-46ee286f831e:
 - Position: [7. 3. 0.]
```

Finally, let's change the velocities and run again, but now for 200 steps:

[8]:
```
from random import randint

for atom in simlammps.iter(oclass=simlammps_ontology.Atom):
    # But the atoms might not have a velocity
    velocity = atom.get(oclass=simlammps_ontology.Velocity)
    if velocity:
        velocity[0].vector = (randint(-3, 3), randint(-3, 3), randint(-3, 3))
    else:
        atom.add(simlammps_ontology.Velocity(vector = (randint(-3, 3), randint(-3, 3),␣
↪randint(-3, 3))))

solver_parameter = simlammps.get(oclass=simlammps_ontology.SolverParameter)[0]
integration_time = solver_parameter.get(oclass=simlammps_ontology.IntegrationTime)[0]
integration_time.steps = 200

simlammps.session.run()
print_info()
```

```
Atom fd4199d4-4d1a-425c-8010-60efca65bd1c:
 - Position: [0.5 4.  6. ]
 - Velocity: [-1. -2.  3.]
Atom f9a32d14-b638-4796-9407-4b1ae6be43cb:
 - Position: [6. 8. 8.]
 - Velocity: [ 3. -3.  3.]
Atom db96a76f-4d70-4e19-b460-46ee286f831e:
 - Position: [9. 4. 0.]
 - Velocity: [2. 1. 0.]
```

[ ]:
```
```

CHAPTER

TWELVE

TUTORIAL: QUANTUM ESPRESSO WRAPPER

In this tutorial we will go through a simple example of how to use the wrapper for the Quantum Espresso simulation engine. You can find the wrapper here.

12.1 Background

This is an example of a slightly different design based upon the input-output functionality of certain simulation engines such as Quantum Espresso and Gromacs.

12.2 Let's get hands-on

12.2.1 Installation

To run the local installation of Quantum Espresso, simply run `./install_engine.sh`. This should check for the prerequisites and compile the code for Quantum Espresso for you.

If the script runs into an error finding openmpi-bin or something like that, try running `apt-get update` and try again. Once the installation has completed, try running `pw.x` to see if the installation has succeeded. If this does not work, then try adding `export PATH=$PATH:/home/username/qe-6.1/bin/` at the end of `.bashrc` located at your home folder.

Once you have verified that `pw.x` works, install the ontology via `pico install ontology.simlammps.yml`, and make sure to run `python3 setup.py` located in the root of the quantum espresso wrapper folder.

That should be all needed to use Quantum Espresso!

12.2.2 Simple example

This is an adaptation of quantum-espresso-wrapper/examples/Simple.py. As usual, we need to import the necessary components:

[1]:
```
import numpy as np

from osp.core.namespaces import QE
from osp.core.utils import pretty_print
from osp.wrappers.quantumespresso.qe_session import qeSession
```

Next, we create simulation and its K points, which determine at what points it samples the cell

[2]:
```
sim = QE.Simulation()
k = QE.K_POINTS(vector = (7, 7, 7), unit = "")
```

Next, we create a cell, the element Silicon, a pseudopotential, an atom and the cell parameters. Note that the pseudopotential files should ALWAYS be located inside of a folder named `$PSEUDO_DIR` inside of wherever you are running the simulation.

[3]:
```
SiCell = QE.Cell()
Si = QE.Element(name = "Si")
SiPseudo = QE.PSEUDOPOTENTIAL(name = "Si.pbe-n-kjpaw_psl.1.0.0.UPF")
Si1 = QE.Atom()
SiParams = QE.CellParams()
celldm1 = QE.Celldm1(value = 5.43070, unit = "au")
```

Next, we connect these all to each other using the `add` method.

[4]:
```
Si.add(SiPseudo, Si1)
Si.add(QE.Mass(value = 28.085, unit = "amu"))
SiCell.add(Si1, SiParams)
Si1.add(QE.Position(vector = (0, 0, 0), unit = ""))
SiCell.add(celldm1)
```

[4]:
```
<qe.Celldm1: be8f3915-3eb7-4221-a441-345eda51832b,  CoreSession: @0x7feae9717370>
```

We specify the cell parameters:

[5]:
```
SiParams.add(QE.CellParameterX(vector = (0.5, 0.5, 0), unit = ""),
             QE.CellParameterY(vector = (0.5, 0, 0.5), unit = ""),
             QE.CellParameterZ(vector = (0, 0.5, 0.5), unit = ""))
```

[5]:
```
[<qe.CellParameterX: 0ebdeed9-1d8a-498d-94c9-bafccb05d652,  CoreSession: @0x7feae9717370>
↪,
 <qe.CellParameterY: fea8789c-8c07-49f9-9971-8d42bdd6ba3f,  CoreSession: @0x7feae9717370>
↪,
 <qe.CellParameterZ: 1474d106-4204-428d-827b-2d5e2cb4af51,  CoreSession: @0x7feae9717370>
↪]
```

And then we add everything created so far to the simulation:

[6]:
```
sim.add(SiCell)
sim.add(Si)
sim.add(k)
```

[6]:
```
<qe.K_POINTS: 6847a5f0-8d20-4f73-9eb3-043e78053182,  CoreSession: @0x7feae9717370>
```

While we're add it, let's add some variables to the simulation which we can check to see if they have been updated. They will not be taken into account when simulating, so they're there for control purposes.

[7]:
```
sim.add(QE.Pressure(value = 100, unit = "kbar"))
sim.add(QE.StressTensor(tensor2 = np.zeros((3, 3)), unit = "kbar"))
```

[7]:
```
<qe.StressTensor: 2f302f8b-89b8-4d7a-a2a7-4f6e19737f00,  CoreSession: @0x7feae9717370>
```

Let's check out what this simulation looks like now with the `pretty_print` function:

[8]:
```
pretty_print(sim)
```

```
- Cuds object:
  uuid: 903145ad-50e3-46fc-9d28-1aa1ec364e8a
  type: qe.Simulation
  superclasses: cuba.Class, cuba.Entity, qe.Simulation
  description:
    All components of the simulation that are needed to run the model

   |_Relationship qe.HAS_PART:
     -  qe.Cell cuds object:
     .  uuid: 67a0fcb4-4977-49df-9bcb-13bca17b2763
     .   |_Relationship qe.HAS_PART:
     .     -  qe.Atom cuds object:
     .     .  uuid: 1bad1c25-609a-4bc0-8c65-3c0167cfdbe2
     .     .   |_Relationship qe.HAS_PART:
     .     .     -  qe.Position cuds object:
     .     .        uuid: c70afbb3-0012-488d-8059-b44d775c6b23
     .     .        vector: [0. 0. 0.]
     .     .        unit:
     .     -  qe.CellParams cuds object:
     .     .  uuid: f444899a-e850-4ab2-b79e-c91026523eb3
     .     .   |_Relationship qe.HAS_PART:
     .     .     -  qe.CellParameterX cuds object:
     .     .     .  uuid: 0ebdeed9-1d8a-498d-94c9-bafccb05d652
     .     .     .  vector: [0.5 0.5 0. ]
     .     .     .  unit:
     .     .     -  qe.CellParameterY cuds object:
     .     .     .  uuid: fea8789c-8c07-49f9-9971-8d42bdd6ba3f
     .     .     .  vector: [0.5 0.  0.5]
     .     .     .  unit:
     .     .     -  qe.CellParameterZ cuds object:
     .     .        uuid: 1474d106-4204-428d-827b-2d5e2cb4af51
     .     .        vector: [0.  0.5 0.5]
     .     .        unit:
     .     -  qe.Celldm1 cuds object:
     .        uuid: be8f3915-3eb7-4221-a441-345eda51832b
     .        unit: au
     .        value: 5.4307
     -  qe.Element cuds object named <Si>:
     .  uuid: 8628ceb7-1c02-4014-95a4-d9450aab4753
     .   |_Relationship qe.HAS_PART:
     .     -  qe.Atom cuds object:
     .     .  uuid: 1bad1c25-609a-4bc0-8c65-3c0167cfdbe2
     .     .  (already printed)
     .     -  qe.Mass cuds object:
     .     .  uuid: 1aee515a-4e12-40e6-bbd6-23bf5c95fe84
     .     .  unit: amu
     .     .  value: 28.085
     .     -  qe.PSEUDOPOTENTIAL cuds object named <Si.pbe-n-kjpaw_psl.1.0.0.UPF>:
     .        uuid: cb27bcb9-27c6-48a9-8f1e-8977b16567c5
     -  qe.K_POINTS cuds object:
     .  uuid: 6847a5f0-8d20-4f73-9eb3-043e78053182
```

(continues on next page)

(continued from previous page)

```
    .  vector: [7. 7. 7.]
    .  unit:
    -  qe.Pressure cuds object:
    .  uuid: d281d93d-fd10-41fd-868c-0cda3b510431
    .  unit: kbar
    .  value: 100.0
    -  qe.StressTensor cuds object:
       uuid: 2f302f8b-89b8-4d7a-a2a7-4f6e19737f00
       unit: kbar
       tensor2: [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Now, it's time to get the simulation running:

[12]:
```
session = qeSession()
quantum_espresso_wrapper = QE.QEWrapper(session = session)
quantum_espresso_wrapper.add(sim)
print("Running calculation...")

quantum_espresso_wrapper.session._run(simulation = sim, prefix = "si", command_type =
↪"pw.x", calculation_type = "scf", root = "", CONTROL = {'pseudo_dir': "'.'"})
```

```
Running calculation...
/mnt/c/iwm/docs/si.pwscf.in
pw.x -i /mnt/c/iwm/docs/si.pwscf.in > /mnt/c/iwm/docs/si.pwscf.out
```

Now let's check the results of our calculation:

[10]:
```
pretty_print(sim)
```

```
- Cuds object:
  uuid: 903145ad-50e3-46fc-9d28-1aa1ec364e8a
  type: qe.Simulation
  superclasses: cuba.Class, cuba.Entity, qe.Simulation
  description:
    All components of the simulation that are needed to run the model

   |_Relationship qe.HAS_PART:
     -  qe.Cell cuds object:
     .  uuid: 67a0fcb4-4977-49df-9bcb-13bca17b2763
     .   |_Relationship qe.HAS_PART:
     .     -  qe.Atom cuds object:
     .     .  uuid: 1bad1c25-609a-4bc0-8c65-3c0167cfdbe2
     .     .   |_Relationship qe.HAS_PART:
     .     .     -  qe.Position cuds object:
     .     .        uuid: c70afbb3-0012-488d-8059-b44d775c6b23
     .     .        vector: [0. 0. 0.]
     .     .        unit:
     .     -  qe.CellParams cuds object:
     .     .  uuid: f444899a-e850-4ab2-b79e-c91026523eb3
     .     .   |_Relationship qe.HAS_PART:
     .     .     -  qe.CellParameterX cuds object:
```

(continues on next page)

(continued from previous page)

```
    .     .     .  uuid: 0ebdeed9-1d8a-498d-94c9-bafccb05d652
    .     .     .  vector: [0.5 0.5 0. ]
    .     .     .  unit:
    .     .     -  qe.CellParameterY cuds object:
    .     .     .  uuid: fea8789c-8c07-49f9-9971-8d42bdd6ba3f
    .     .     .  vector: [0.5 0.  0.5]
    .     .     .  unit:
    .     .     -  qe.CellParameterZ cuds object:
    .     .        uuid: 1474d106-4204-428d-827b-2d5e2cb4af51
    .     .        vector: [0.  0.5 0.5]
    .     .        unit:
    .     -  qe.Celldm1 cuds object:
    .        uuid: be8f3915-3eb7-4221-a441-345eda51832b
    .        unit: au
    .        value: 5.4307
    -  qe.Element cuds object named <Si>:
    .  uuid: 8628ceb7-1c02-4014-95a4-d9450aab4753
    .   |_Relationship qe.HAS_PART:
    .     -  qe.Atom cuds object:
    .     .  uuid: 1bad1c25-609a-4bc0-8c65-3c0167cfdbe2
    .     .  (already printed)
    .     -  qe.Mass cuds object:
    .     .  uuid: 1aee515a-4e12-40e6-bbd6-23bf5c95fe84
    .     .  unit: amu
    .     .  value: 28.085
    .     -  qe.PSEUDOPOTENTIAL cuds object named <Si.pbe-n-kjpaw_psl.1.0.0.UPF>:
    .        uuid: cb27bcb9-27c6-48a9-8f1e-8977b16567c5
    -  qe.K_POINTS cuds object:
    .  uuid: 6847a5f0-8d20-4f73-9eb3-043e78053182
    .  vector: [7. 7. 7.]
    .  unit:
    -  qe.Pressure cuds object:
    .  uuid: d281d93d-fd10-41fd-868c-0cda3b510431
    .  unit: kbar
    .  value: 100.0
    -  qe.PwOut cuds object:
    .  uuid: 15c6637e-9124-44dd-a1d3-f225203c1bfc
    .  path: /mnt/c/iwm/docs/si.pwscf.out
    -  qe.StressTensor cuds object:
       uuid: 2f302f8b-89b8-4d7a-a2a7-4f6e19737f00
       unit: kbar
       tensor2: [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

As you can see, the original part of the cuds tree is still there, with everything mostly the same. The new parts are:

- The qe.PwOut cuds object. This is the output file of the simulation, in case there is something that the wrapper does not parse but that you would still like to see.
- The qe.TotalEnergy cuds object. This was parsed from the qe.PwOut file itself.
- The qe.Force cuds object. This represents the force exerted on the atom(s).

The updated parts are:

- The qe.Pressure cuds object, having changed in value from 100 kbar to 5723.64 kbar.
- The qe.StressTensor cuds object, which is no longer zero.

Let's see if we can do better and calculate some bands structures:

[21]:
```
quantum_espresso_wrapper.session._run(prefix = "si", command_type = "pw.x", calculation_
↪type = "bands")
quantum_espresso_wrapper.session._run(prefix = "si", command_type = "bands.x",␣
↪calculation_type = "")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-51324d6941c1> in <module>
----> 1 quantum_espresso_wrapper.session._run(prefix = "si", command_type = "pw.x",␣
↪calculation_type = "bands")
      2 quantum_espresso_wrapper.session._run(prefix = "si", command_type = "bands.x",␣
↪calculation_type = "")

TypeError: _run() missing 1 required positional argument: 'simulation'
```

Although the cuds structure won't have changed much by this, the data is there in the folder.

Now let's try to relax this cell. While it isn't a real cell, we can still perform the calculations to relax it to know what the movement of the atoms would be like if it were a real cell (warning, perform vc-relax type calculations with caution. These examples are designed to be lightweight and non-indicative of real-world applications).

[22]:
```
quantum_espresso_wrapper.session._run(simulation = sim, prefix = "si", command_type =
↪"pw.x", calculation_type = "relax", IONS = {'ion_dynamics': "'bfgs'"})
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-8a4fa9c53f25> in <module>
----> 1 quantum_espresso_wrapper.session._run(prefix = "si", command_type = "pw.x",␣
↪calculation_type = "relax", IONS = {'ion_dynamics': "'bfgs'"})

TypeError: _run() missing 1 required positional argument: 'simulation'
```

[23]:
```
pretty_print(sim)
```

```
- Cuds object:
  uuid: 90361daa-6905-4566-979e-11b3b0dd4e85
  type: qe.Simulation
  superclasses: cuba.Class, cuba.Entity, qe.Simulation
  description:
    All components of the simulation that are needed to run the model

   |_Relationship qe.HAS_PART:
     -  qe.Cell cuds object:
     .  uuid: f7548873-28e9-4d76-86da-6fddb687d29e
     .   |_Relationship qe.HAS_PART:
     .     -  qe.Atom cuds object:
     .     .  uuid: 53114a1f-ebbb-4e4b-a115-080925d9eaa8
     .     .   |_Relationship qe.HAS_PART:
     .     .     -  qe.Position cuds object:
     .     .        uuid: 48d4483b-7c72-4454-8041-581dc73fd216
```

(continues on next page)

(continued from previous page)

```
.     .        vector: [0. 0. 0.]
.     .        unit:
.     -  qe.CellParams cuds object:
.     .  uuid: 9d61e990-2509-474b-935e-618ca11bb40d
.     .   |_Relationship qe.HAS_PART:
.     .     -  qe.CellParameterX cuds object:
.     .     .  uuid: f2655054-efa7-4b39-9f0a-cf6453be68ec
.     .     .  vector: [0.5 0.5 0. ]
.     .     .  unit:
.     .     -  qe.CellParameterY cuds object:
.     .     .  uuid: 55647575-ea8f-4ef7-aee8-2a3333a4ec71
.     .     .  vector: [0.5 0.  0.5]
.     .     .  unit:
.     .     -  qe.CellParameterZ cuds object:
.     .        uuid: 92be9c63-ee80-46d9-8853-ccb562e94a5b
.     .        vector: [0.  0.5 0.5]
.     .        unit:
.     -  qe.Celldm1 cuds object:
.     .  uuid: db776c65-9d2e-448e-bc55-5fe0f9c7ee75
.     .  unit: au
.     .  value: 5.4307
.     -  qe.Element cuds object named <Si>:
.        uuid: 14dacecb-023c-4ace-9e83-35b0ecaa1032
.         |_Relationship qe.HAS_PART:
.           -  qe.Atom cuds object:
.           .  uuid: 53114a1f-ebbb-4e4b-a115-080925d9eaa8
.           .  (already printed)
.           -  qe.Atom cuds object:
.           .  uuid: c2094a19-8769-4298-a50a-be1f8befe5bf
.           .   |_Relationship qe.HAS_PART:
.           .     -  qe.Position cuds object:
.           .        uuid: 4087ce47-16f0-4449-b8c4-4577e6d265e2
.           .        vector: [0.25 0.25 0.26]
.           .        unit:
.           -  qe.Mass cuds object:
.           .  uuid: 5d57a768-d315-4f91-84a8-fcfad9aae382
.           .  unit: amu
.           .  value: 28.085
.           -  qe.PSEUDOPOTENTIAL cuds object named <Si.pbe-n-kjpaw_psl.1.0.0.UPF>:
.              uuid: ab064983-5cc9-418e-a9e7-3357c04388f5
-  qe.Element cuds object named <Si>:
.  uuid: 14dacecb-023c-4ace-9e83-35b0ecaa1032
.  (already printed)
-  qe.K_POINTS cuds object:
.  uuid: 38385f3b-128c-491f-91f4-44de15055d56
.  vector: [7. 7. 7.]
.  unit:
-  qe.Pressure cuds object:
.  uuid: 57f47fa8-4588-488f-8c89-fa8a0f37f567
.  unit: kbar
.  value: 100.0
-  qe.Pressure cuds object:
```

(continues on next page)

(continued from previous page)

```
    .  uuid: beac6977-01cf-4b4d-b89b-1b4d4eecf5c0
    .  unit: kbar
    .  value: 100.0
    -  qe.PwOut cuds object:
    .  uuid: 7890c86c-9f6f-42b3-a02c-cd57f5307c75
    .  path: si.pwscf.out
    -  qe.PwOut cuds object:
    .  uuid: 5c7e31f9-6e55-4ddf-af62-5a3e58811464
    .  path: si.pwscf.out
    -  qe.StressTensor cuds object:
    .  uuid: 16b33d51-0f6d-4451-a919-da494a13082f
    .  unit: kbar
    .  tensor2: [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
    -  qe.StressTensor cuds object:
       uuid: 590ad64e-d110-4711-90c6-0986bb53dafc
       unit: kbar
       tensor2: [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

In this example, the position hasn't changed. So let's spice things up a little bit and add another atom, and then relax:

[24]:
```
Si2 = QE.Atom()
Si2.add(QE.Position(vector = (0.25, 0.25, 0.26), unit = ""))
SiCell.add(Si)
Si.add(Si2)
pretty_print(sim)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-6957eba14ddb> in <module>
      1 Si2 = QE.Atom()
      2 Si2.add(QE.Position(vector = (0.25, 0.25, 0.26), unit = ""))
----> 3 SiCell.add(Si)
      4 Si.add(Si2)
      5 pretty_print(sim)

/mnt/c/IWM/osp-core-3.4.0-dev/osp/core/cuds.py in add(self, rel, *args)
    161             if rel in self._neighbors and arg.uid in self._neighbors[rel]:
    162                 message = '{!r} is already in the container'
--> 163                 raise ValueError(message.format(arg))
    164             if self.session != arg.session:
    165                 arg = self._recursive_store(arg, next(old_objects))

ValueError: <qe.Element: 14dacecb-023c-4ace-9e83-35b0ecaa1032,  CoreSession:␣
↪@0x7f3d2d38bfa0> is already in the container
```

[25]:
```
quantum_espresso_wrapper.session._run(simulation = sim, prefix = "si", command_type =
↪"pw.x", calculation_type = "relax", IONS = {'ion_dynamics': "'bfgs'"})
pretty_print(sim)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-8eda2c089407> in <module>
----> 1 quantum_espresso_wrapper.session._run(prefix = "si", command_type = "pw.x",␣
↪calculation_type = "relax", IONS = {'ion_dynamics': "'bfgs'"})
      2 pretty_print(sim)

TypeError: _run() missing 1 required positional argument: 'simulation'
```

CHAPTER

THIRTEEN

INTRODUCTION ON ONTOLOGIES

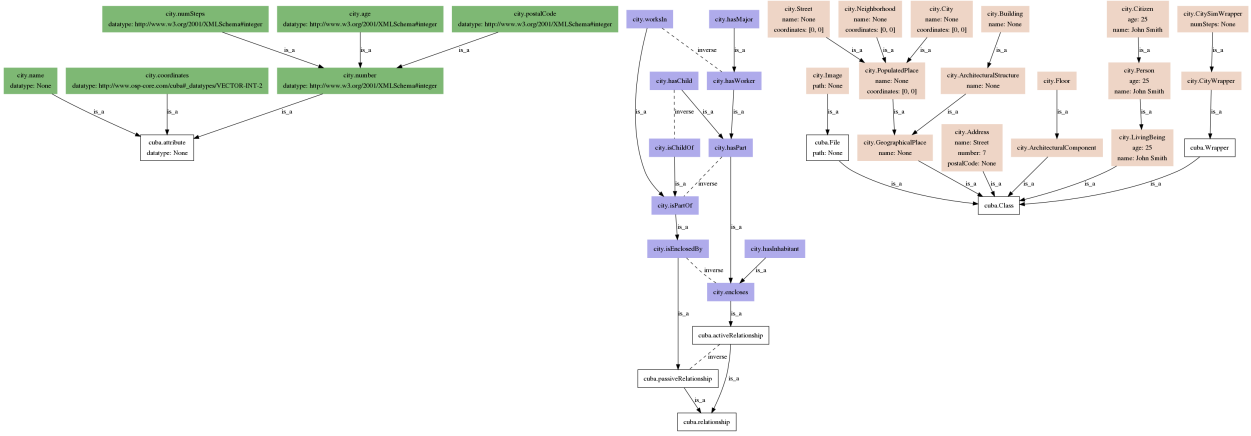
What is an ontology?

An ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application. (Source: http://tomgruber.org/writing/ontology-definition-2007.htm)

13.1 An example: the City ontology

OSP-core ships with one simple example ontology, called `city`. You can use it to play around and get familiar with OSP-core. We will also use it a lot in this documentation as an example.

The city ontology provides the concepts to describe people and buildings in a city. In this graph we show the different entities in the ontology. We used *Ontology2Dot* for that:



To use the city ontology you have to install it using the tool Pico:

```
pico install city
```

Take a look at our *examples* to see how you can build your own city!

CHAPTER

FOURTEEN

HOW TO WORK WITH ONTOLOGIES

OSP-core supports ontologies in the following formats:

- *OWL ontologies*
- *RDFS vocabularies* (limited support)
- *OSP-core YAML ontology format*

14.1 OWL ontologies and RDFS vocabularies

To install OWL ontologies or RDFS vocabularies in OSP-core, you have to create a configuration yaml file similar to the following one:

```
identifier: emmo
ontology_file: https://raw.githubusercontent.com/emmo-repo/EMMO/master/emmo-inferred.owl
format: turtle
reference_by_label: True
namespaces:
  mereotopology: http://emmo.info/emmo/top/mereotopology
  physical: http://emmo.info/emmo/top/physical
  top: http://emmo.info/emmo/top
  semiotics: http://emmo.info/emmo/top/semiotics
  perceptual: http://emmo.info/emmo/middle/perceptual
  reductionistic: http://emmo.info/emmo/middle/reductionistic
  holistic: http://emmo.info/emmo/middle/holistic
  physicalistic: http://emmo.info/emmo/middle/physicalistic
  math: http://emmo.info/emmo/middle/math
  properties: http://emmo.info/emmo/middle/properties
  materials: http://emmo.info/emmo/middle/materials
  metrology: http://emmo.info/emmo/middle/metrology
  models: http://emmo.info/emmo/middle/models
  manufacturing: http://emmo.info/emmo/middle/manufacturing
  isq: http://emmo.info/emmo/middle/isq
  siunits: http://emmo.info/emmo/middle/siunits
active_relationships:
  - http://emmo.info/emmo/top/mereotopology#EMMO_8c898653_1118_4682_9bbf_6cc334d16a99
  - http://emmo.info/emmo/top/semiotics#EMMO_60577dea_9019_4537_ac41_80b0fb563d41
default_relationship: http://emmo.info/emmo/top/mereotopology#EMMO_17e27c22_37e1_468c_
↪9dd7_95e137f73e7f
```

14.1.1 Keywords

identifier: Can be any alphanumerical string. It is the name of the package that contains multiple namespaces. Will be used for uninstallation: `pico uninstall emmo`. (In YAML ontologies this package name or identifier is the same as the namespace name).

ontology_file: Path to the inferred owl ontology. That means you should have executed a reasoner on your ontology, e.g. by using the `Export inferred axioms` functionality of Protégé.

format (optional): File format of the ontology file to be parsed. We support all the formats that RDFLib supports: XML (`xml`, `application/rdf+xml`, default), Turtle (`turtle`, `ttl`, `text/turtle`), N3 (`n3`,`text/n3`), NTriples (`nt`, `nt11`, `application/n-triples`), N-Quads (`nquads`, `application/n-quads`), TriX (`trix`, `application/trix`) and TriG (`trig`, `application/trig`). When not provided, it will be guessed from the file extension. However, such guess may not always be correct.

reference_by_label (default False): Whether the label should be used or the IRI suffix to reference entity from within OSP-core. In case of EMMO it is true, because IRI suffixes are not human friendly. In this case all labels should be unique and not contain whitespaces. If False, use dot notation to get by IRI square brackets (`__getitem__`) to get by label. The latter will return a list of all entities with the same label.

namespaces: mapping from namespace name (used to import the namespace) to iri prefix. If IRI doesn't end with "/" or "#", "#" will be added.

active relationships: List of iris of active relationships.

default relationship: The default relationship.

14.1.2 Installation

Name the yaml file as you would any yaml file `<name>.yml`, where `<name>` should be replaced by a user defined name.

Then you can use pico to install the tool Pico to install the ontology:

```
pico install </path/to/name.yml>
```

14.1.3 Limitations

At the moment, there are a few limitations on the supported features of OWL ontologies and RDFS vocabularies.

OWL ontologies

Not all predicates of OWL ontologies are taken into consideration. Among the used ones are:

- `RDF.type` to determine the type of the entities.
- `RDFS.label` to get the entities by label.
- `RDFS.isDefinedBy` to get a descriptions for the entities.
- `RDFS.subClassOf` / `RDFS.subPropertyOf` for subclasses.
- `OWL.inverseOf` for inverse relationships.
- `RDFS.range` to determine the datatype of `DataProperties`. These are the supported datatypes:
  - `XSD.boolean`
  - `XSD.integer`

  - `XSD.float`
  - `XSD.string`
- To get the attributes of an owl class, we use
  - The `RDFS.domain` of the `DatatypeProperties`, if it is a simple class.
  - Restrictions on the ontology classes.
  - Furthermore, all DataProperties are considered functional, see this issue.
- Restrictions and compositions are also supported. They can be consulted using the `axioms` attribute of ontology classes.

No reasoner is included. We plan to include a reasoner in the future.

We try to extend this list over time and support more of the OWL DL standard.

RDFS vocabularies

With respect to RDFS vocabularies, the `RDFS.Class` predicate is supported, but the `RDFS.Property` predicate is not. This means that the main limitation when using RDFS vocabularies is that only their classes are detected, but their properties are ignored.

14.2 OSP-core YAML ontology format

This section describes how you can create ontologies using YAML.

Tip: If you have an ontology where all entity names are in ALL_UPPERCASE, you can use the commandline tool `yaml2camelcase` that is shipped with OSP-core to transform it to an ontology with CamelCase entity names.

14.2.1 Introduction

In this file we will give a description of how an Ontology can be represented in a yaml file format and how to interpret such files. For simplicity reasons in the following we will give examples from the *** example ontology *** file which can be found in osp/core/ontology/yml/ontology.city.yml.

14.2.2 Naming of the files and installation

Name any ontology `<name>.ontology.yml`, where `<name>` should be replaced by a user defined name.

Then you can use pico to install the tool Pico to install the ontology:

```
pico install </path/to/my_ontology.ontology.yml>
```

14.2.3 Syntax of the .yml ontology

`version`: string

Contains semantic version Major.minor in format M.m with M and m positive integers. minor MUST be incremented when backward compatibility in the format is preserved. Major MUST be incremented when backward compatibility is removed. Due to the strict nature of the format, a change in minor is unlikely.

`namespace`: string

Defines the namespace of the current file. We recommend to use all_lowercase for the namespace name, with underscore as separation. All entities defined in this file will live in the namespace defined here.

`requirements`: List[string]

A list of namespaces that this namespace depends on.

`author`: string

Reference to the person(s) who created the file.

`ontology`: dict

Contains individual declarations for ontology entities as a mapping. Each key of the mapping is the name of an ontology entity. The key can contain letters, numbers and underscore. By convention, they should be in CamelCase. The name of ontology classes should start with an uppercase latter, while the name of relationships and attributes should start with a lower case letter. This key is later used the reference the entity from within OSP-core in a case sensitive manner. The value of the mapping is a mapping whose format is detailed in the *Ontology entities format*.

14.2.4 Ontology entities format

Every declaration of an ontology entity must have the following keys:

`description`: string

For human consumption. An ontological short description of the carried semantics. Should have the form: entity is a superclass_entity that has <differentiating> terms.

`subclass_of`: List[``**qualified entity name**``]. Its value is fixed on the ontology level.

The subclass keyword expresses an **ontological is-a** relation. MUST be a list of a fully qualified strings referring to another entity. Only the entity `cuba.entity` is allowed to have no superclass. See *CUBA namespace*.

If entity A is a subclass of B and B is subclass of C, then A is also subclass of C.

An ontology entity can be either a relationship, a cuds entity or an attribute. Depending on that the mapping can have further keys. For cuds entities these keys are described in *CUDS entities format* section. For relationship entities, these keys are described in *Relationship format* section. For attributes, these keys are described in *Attribute format* section.

14.2.5 The CUBA namespace

The CUBA namespace contains a set of Common Universal Basic entities, that have special meaning in OSP-core. The CUBA namespace is always installed in OSP-core.

`cuba.Entity`

The root for all ontology classes. Does not have a superclass.

`cuba.Nothing`

An ontology class, that is not allowed to have individuals.

`cuba.relationship`

The root of all relationships. Does not have a superclass.

`cuba.attribute`

The root of all attributes. Does not have a superclass.

`cuba.Wrapper`

The root of all wrappers. These are the bridge to simulation engines and databases. See the examples in examples/.

`cuba.activeRelationship`

The root of all active relationships. Active relationships express that one cuds object is in the container of another.

`cuba.passiveRelationship`

The inverse of `cuba.activeRelationship`.

14.2.6 Attribute format

Every attribute is a subclass of `cuba.attribute`. The declaration of an attribute is a special case of the declaration of an entity. It must have the keys described in *Ontology entities format*. It can additionally have the following keys:

`datatype`: string

It is an attribute of an entity in cases when the datatype of said entity is important.

Describes the datatype of the value that a certain entity can take. It can be one of the following:

- `BOOL`: a form of data with only two possible values (usually "true" or "false")
- `INT`: a positive or negative integer number.
- `FLOAT`: a number containing values on both sides of the decimal point
- `STRING`: a sequence of characters that can also contain spaces and numbers. The length can be specified with "STRING:LENGTH" (e.g. STRING:20 means the length of the string should be maximum 20 characters).
- `VECTOR:datatype:D1:D2:...:Dn`: a vector of the given dimensions (D1 x D2 x ... x Dn) and the given datatype. The dimensions are always fixed.

  For example, a VECTOR:INT:4:2:1 would be:
  { [(a), (b)], [(c), (d)], [(e), (f)], [(g), (h)] }
  where all elements (a, b, ...) are integers. (the different delimiters are only used for visual purposes).
  If no datatype is specified, it would be a `FLOAT` vector.

In case a datatype is not specified the default datatype is assumed to be STRING

For example: The datatype of entity numberOfOccurrences is INT.

Note: The implementation of the vectors is experimental and will be updated as soon as EMMO has established an appropriate wait of representing them

---

14.2.7 Class expressions

A class expression describes a subset of individuals. They are similar to classes, but do not have a name in the ontology. Class expressions will be used in *CUDS entities format* and *Relationship format*. They can be either:

- A ``**qualified entity name**`` of a class. In this case it corresponds to all individuals of referenced class.
- Requirements on the individual's relationships. For example:

```
city.hasInhabitant:
  cardinality: 1+
  range: city.Citizen
  exclusive: false
```

This describes the set of individuals that have at least one citizen as inhabitant. In general it describes the individuals which have some relationship to some object. It is a mapping from the ``**qualified entity name**`` of a relationship to the following keywords:

`range`

A class expression describing the individuals which are object of the relationship.

`cardinality`

The number of times individuals defined in `range` is allowed to be a object of the relationship. To define the `cardinality` we use the following syntax:

- many / * / 0+ (default): Zero to infinity target objects are allowed.
- some / + / 1+: At least one target object is required.
- ? / 0-1: At most one target object is allowed.
- a+: At least a target objects are required.
- a-b: At least a and at most b target objects are required (i.e. inclusive).
- a: Exactly a target objects are required.

`exclusive`

Whether the given `target` is the only allowed object.

- A composition of several class expressions. For example:

```
or:
  - city.City
  - city.Neighborhood
```

This is the union of all individuals that are a city or a neighbourhood. We use the keyword `or` for union, `and` for intersection and `not` for complement. After `or` and `and`, a list of class expressions for the union / intersection is expected. After `not` a single class expression is expected.

The definition of class expressions is recursive. For example:

```
or:
  - city.City
  - city.hasPart:
      cardinality: 1+
      range: city.Neighborhood
      exclusive: false
```

This describes the set of all individuals that are a city or have a neighbourhood.

14.2.8 CUDS classes format

The declaration of a cuds entity is a special case of the declaration of an entity. It must have the keys described in *Ontology entities format*. It can contain further information:

`attributes`: Dict[``**qualified entity name**``, default_value]

Expects a mapping from the ``**qualified entity name**`` of an attribute to its default. Each key must correspond to a subclass of `attribute`. For example:

```
Address:
  ...
  attributes:
    city.name: "Street"
    city.number:
```

Here, an address has a name and a number. The default name is "Street", the number has no default. If no default is given, the user is forced to specify a value each time he creates an individual.

`subclass_of`: List[Class expression]

In addition to qualified entity names of classes, class expressions are also allowed. These class expressions restrict the individuals allowed in the class. Only those individuals are allowed that are in the intersection of the class expressions. For example:

```
PopulatedPlace:
  description:
  subclass_of:
    - city.GeographicalPlace
    - city.hasInhabitant:
        range: city.Citizen
        cardinality: many
        exclusive: true
```

Here, a populated place is a geographical place which must have citizens as inhabitants.

`disjoint_with`: List[Class expression]

A list of class expressions that are not allowed to share any individuals with the cuds entity.

`equivalent_to`: List[Class expression]

A list of class expressions who contain the same individuals as the cuds entity. For example:

```
PopulatedPlace:
  description:
```

(continues on next page)

(continued from previous page)

```
  equivalent_to:
    - city.GeographicalPlace
    - city.hasInhabitant:
        range: city.Citizen
        cardinality: many
        exclusive: true
```

Here every geographical place that has citizens as inhabitants is automatically a populated place.

14.2.9 Relationship format

Every relationship is a subclass of `cuba.relationship`. The declaration of a relationship is a special case of the declaration of an entity. It must have the keys described in *Ontology entities format*. Furthermore, it can contain the following information:

`inverse`: ``**qualified entity name**`` or empty (None)

If CUDS object A is related to CUDS object B via relationship `rel`, then B is related with A via the inverse of `rel`.

For example: The inverse of `hasPart` is `isPartOf`.

If no inverse is given, OSP-core will automatically create one.

`domain`: Class expression

A class expression describing the individuals that are allowed to be a subject of the relationship. If multiple class expressions are given, the relationship's domain is the intersection of the class expressions.

`range`: Class expression

A class expression describing the individuals that are allowed to be object of the relationship. If multiple class expressions are given, the relationship's range is the intersection of the class expression.

`characteristics`: String

A list of characteristics of the relationship. The following characteristics are supported:

- reflexive
- symmetric
- transitive
- functional
- irreflexive
- asymmetric
- inversefunctional

A subclass of a relationship is called a sub-relationship.

14.2.10 Limitations

`Class expressions`, `domain`, `range`, `characteristics`, `equivalent_to`, `disjoint_with` are currently not parsed by OSP-core.

CHAPTER

FIFTEEN

INCLUDED ONTOLOGIES

As described on the *working with ontologies* section, to use an ontology you first have to install it, and to do so usually you have either to define a `yml` *configuration file* (for OWL ontologies and RDFS vocabularies) or provide the ontology in the *OSP-core YAML ontology format*.

However, in order to make using ontologies easier, we bundle a few of these files with OSP-core to enable rapid installation of common, well-known ontologies. Do not hesitate to *contact us* if you want your ontology to be shipped with SimPhoNy.

The included ontologies, together with their domains of application, are listed below.

- *Elementary Multiperspective Material Ontology (EMMO)* - Applied sciences
- *Dublin Core Metadata Initiative (DCMI)* - Metadata description
- *Data Catalog Vocabulary - Version 2 (DCAT2)* - Data catalogue information
- *Friend of a Friend (FOAF)* - People and information on the web
- *The PROV Ontology (PROV-O)* - Provenance information
- *The City Ontology* - Example ontology aimed at demonstrating the usage of SimPhoNy OSP-core

The ontologies can be installed providing the right package identifier to *pico*. You can find such package identifier and additional information on each ontology by clicking on the links from the list above.

15.1 Elementary Multiperspective Material Ontology (EMMO)

> EMMO is a multidisciplinary effort to develop a standard representational framework (the ontology) for applied sciences. It is based on physics, analytical philosophy and information and communication technologies. It has been instigated by materials science to provide a framework for knowledge capture that is consistent with scientific principles and methodologies. It is released under a Creative Commons CC BY 4.0 license.
>
> —About EMMO section, from the official EMMO GitHub repository

For a short introduction on this ontology, see the *fundamentals* section. To install the EMMO ontology, use

```
pico install emmo
```

and then just start creating cuds objects

```
>>> from osp.core.namespaces import math
>>> math.Numerical.attributes
{<OntologyAttribute math.hasNumericalData>: None}
```

(continues on next page)

(continued from previous page)

```
>>> x = math.Numerical(hasNumericalData=12)
>>> x
<math.Numerical: c11cc272-cdcf-421a-8838-5f177b065746,  CoreSession: @0x7f1987173190>
>>> x.hasNumericalData
12
```

15.2 Dublin Core Metadata Initiative (DCMI)

> The Dublin Core™ Metadata Initiative, or "DCMI", is an organization supporting innovation in metadata design and best practices across the metadata ecology. DCMI works openly, and it supported by a paid-membership model.
>
> —About DCMI

The Dublin Core™ Metadata Initiative has published, among others, the DCMI Metadata Terms specification, which establishes a set of core metadata terms enabling cross-domain description of resources on the web.

> Included are the fifteen terms of the Dublin Core™ Metadata Element Set (also known as "the Dublin Core") plus several dozen properties, classes, datatypes, and vocabulary encoding schemes. [...] These terms are intended to be used in combination with metadata terms from other, compatible vocabularies in the context of application profiles.
>
> —DCMI Metadata Terms

To install the `dcmitype` and `dcterms` RDFS vocabularies from the Dublin Core Metadata Initiative (DCMI), use

```
pico install dcmitype dcterms
```

Note that due to the fact that *RDFS properties are not supported by OSP-core*, the properties in these two vocabularies will be ignored. Only the classes will be detected.

15.3 Data Catalog Vocabulary - Version 2 (DCAT2)

> DCAT is an RDF vocabulary designed to facilitate interoperability between data catalogs published on the Web. [...]
>
> DCAT enables a publisher to describe datasets and data services in a catalog using a standard model and vocabulary that facilitates the consumption and aggregation of metadata from multiple catalogs. This can increase the discoverability of datasets and data services. It also makes it possible to have a decentralized approach to publishing data catalogs and makes federated search for datasets across catalogs in multiple sites possible using the same query mechanism and structure. Aggregated DCAT metadata can serve as a manifest file as part of the digital preservation process.
>
> —Data Catalog Vocabulary (DCAT) - Version 2

To install the DCAT2 ontology, use

```
pico install dcat2
```

15.4 Friend of a Friend (FOAF)

> FOAF is a project devoted to linking people and information using the Web. Regardless of whether information is in people's heads, in physical or digital documents, or in the form of factual data, it can be linked. FOAF integrates three kinds of network: social networks of human collaboration, friendship and association; representational networks that describe a simplified view of a cartoon universe in factual terms, and information networks that use Web-based linking to share independently published descriptions of this inter-connected world. FOAF does not compete with socially-oriented Web sites; rather it provides an approach in which different sites can tell different parts of the larger story, and by which users can retain some control over their information in a non-proprietary format.
>
> —FOAF Vocabulary Specification

To install the FOAF ontology, use

```
pico install foaf
```

15.5 The PROV Ontology (PROV-O)

> The PROV Ontology (PROV-O) expresses the PROV Data Model [PROV-DM] using the OWL2 Web Ontology Language (OWL2) [OWL2-OVERVIEW]. It provides a set of classes, properties, and restrictions that can be used to represent and interchange provenance information generated in different systems and under different contexts. It can also be specialized to create new classes and properties to model provenance information for different applications and domains.
>
> —PROV-O: The PROV Ontology

To install the PROV-O ontology, use

```
pico install prov
```

15.6 The City ontology

The City ontology is a *simple, example ontology* included with OSP-core. It provides a collection of concepts to describe people and buildings in a city, and is aimed at demonstrating the usage of SimPhoNy OSP-core.

To install the *city ontology*, use

```
pico install city
```

CHAPTER

SIXTEEN

TUTORIAL: ONTOLOGY INTERFACE

This tutorial introduces the interface to the installed ontologies. The code presented is based on this example.

16.1 Background

In an ontological framework, ontology entities are used as a knowledge representation form. Those can be further categorized in two groups: ontology individuals (assertional knowledge), and ontology classes, relationships and attributes (terminological knowledge).

In a *previous tutorial*, we have discussed how to work with CUDS objects, which represent ontology individuals. In this tutorial, we present the API of all the other entities instead: ontology classes, relationships and attributes. These are defined in an ontology installation file in *YAML* or *OWL* format. The presented API enables you to access the entities and navigate within an ontology.

In this tutorial, we will work both with the `city` namespace, the example namespace from OSP-core, and the `math` namespace from the Elementary Multiperspective Material Ontology (EMMO), for which an installation file is also provided with OSP-core.

Please install the ontologies running the commands below if you have not installed them yet.

[1]:
```
# Install the ontologies
!pico install city
!pico install emmo
```

```
INFO 2021-03-31 16:16:53,174 [osp.core.ontology.installation]: Will install the␣
↪following namespaces: ['city']
INFO 2021-03-31 16:16:53,187 [osp.core.ontology.yml.yml_parser]: Parsing YAML ontology␣
↪file /home/jose/.local/lib/python3.9/site-packages/osp/core/ontology/docs/city.
↪ontology.yml
INFO 2021-03-31 16:16:53,209 [osp.core.ontology.yml.yml_parser]: You can now use `from␣
↪osp.core.namespaces import city`.
INFO 2021-03-31 16:16:53,209 [osp.core.ontology.parser]: Loaded 202 ontology triples in␣
↪total
INFO 2021-03-31 16:16:53,223 [osp.core.ontology.installation]: Installation successful
INFO 2021-03-31 16:16:53,753 [osp.core.ontology.installation]: Will install the␣
↪following namespaces: ['emmo']
INFO 2021-03-31 16:16:53,756 [osp.core.ontology.parser]: Parsing /home/jose/.local/lib/
↪python3.9/site-packages/osp/core/ontology/docs/emmo.yml
INFO 2021-03-31 16:16:53,756 [osp.core.ontology.parser]: Downloading https://raw.
↪githubusercontent.com/emmo-repo/emmo-repo.github.io/master/versions/1.0.0-alpha2/emmo-
↪inferred.owl
```

(continues on next page)

(continued from previous page)

```
INFO 2021-03-31 16:16:54,299 [osp.core.ontology.parser]: Parsing /tmp/tmpvqey417g-emmo
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import mereotopology`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import physical`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import top`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import semiotics`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import perceptual`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import reductionistic`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import holistic`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import physicalistic`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import math`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import properties`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import materials`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import metrology`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import models`.
INFO 2021-03-31 16:16:54,897 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import manufacturing`.
INFO 2021-03-31 16:16:54,898 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import isq`.
INFO 2021-03-31 16:16:54,898 [osp.core.ontology.parser]: You can now use `from osp.core.
↪namespaces import siunits`.
INFO 2021-03-31 16:16:54,901 [osp.core.ontology.parser]: Loaded 4664 ontology triples in␣
↪total
INFO 2021-03-31 16:16:55,083 [osp.core.ontology.installation]: Installation successful
```

16.2 Accessing entities: the namespace object

To access ontology entities, we first need to know the aliases of the installed ontology namespaces. In each ontology *YAML installation file*, the namespace(s) that it contains is(are) stated at the top of the file. For example, at the top of the city ontology installation file you may find:

```
---
version: "0.0.3"

namespace: "city"

ontology:
   ...
```

Alternatively, you can use *pico ontology installation tool* to see the installed namespaces:

[2]:
```
!pico list
```

```
Packages:
        - city
        - emmo
Namespaces:
        - xml
        - rdf
        - rdfs
        - xsd
        - cuba
        - isq
        - ns1
        - ns2
        - owl
        - city
        - mereotopology
        - physical
        - top
        - semiotics
        - perceptual
        - reductionistic
        - holistic
        - physicalistic
        - math
        - properties
        - materials
        - metrology
        - models
        - manufacturing
        - siunits
```

Once we know the name of the namespace that we want to use, we import it in python. For this tutorial, we are importing the namespaces `city` and `math`. Through those imported namespace python objects, the entities within the namespaces can be accessed:

[3]:
```
from osp.core.namespaces import city
from osp.core.namespaces import math
```

There are several ways to access an ontology entity in OSP-core, which are summarized by the following list and will be demonstrated shortly after.

- By **suffix**. For example, for the namespace `city`, whose IRI is `http://www.osp-core.com/city#`, fetching by the suffix `Citizen` would return the ontology entity with IRI `http://www.osp-core.com/city#Citizen`.
- By **label**. Fetchs the entity by the label that has been assigned to it using either the `rdfs:label` or `skos:prefLabel` predicates.
- By **IRI**. The full IRI of an ontology entity is provided in order to fetch it.
- By **string**. Using a string, for example `"city.LivingBeing"`. This is only useful in some special cases.

The **most convenient way** to access an ontology entity is using the **dot notation** in python. For example, `city.Citizen`. This method is a shorthand for fetching by suffix or label:

- When the keyword `reference_by_label` is set to `True` (enabled) in the *ontology YAML installation file*, the dot notation is a shorthand for fetching by label. This keyword is **enabled** in the `math` namespace.
- When the keyword `reference_by_label` is set to `False` (disabled) or not set, the dot notation is a shorthand for fetching by suffix instead. This keyword is **disabled** in the `city` namespace.

To get a list of all the entities available within a namespace, run `list(namespace)`.

Tip

The dot notation supports IPython autocompletion. For example, when working on a Jupyter notebook, once the namespace has been imported, it is possible to get suggestions for the entity names by writing `namespace.` and pressing TAB.

Accessing an ontology entity by suffix

Let's fetch the Citizen class, whose IRI is `http://www.osp-core.com/city#Citizen`.

The keyword, `reference_by_label` is set to `False`, so one can just use the dot notation.

```
[4]: city.Citizen
```

```
[4]: <OntologyClass city.Citizen>
```

Another alternative is using the `get_from_suffix` method from the namespace object. This is useful when the suffix contains characters that Python does not accept as property names, such as spaces or dashes.

```
[5]: city.get_from_suffix('Citizen')
```

```
[5]: <OntologyClass city.Citizen>
```

Note that the suffix is case sensitive, and therefore the following would produce an error.

```
[6]: # city.citizen  # -> Fails.
```

Accessing an ontology entity by label

Let's fetch the Integer class, whose IRI is `http://emmo.info/emmo/middle/math#EMMO_f8bd64d5_5d3e_4ad4_a46e_c30714fecb7f`.

The keyword `reference_by_label` is set to `True`, so we just use the dot notation.

```
[7]: math.Integer
```

```
[7]: <OntologyClass math.Integer>
```

Another alternative is using the square bracket notation on the namespace object. This is useful when the suffix contains characters that Python does not accept as property names, such as spaces or dashes.

```
[8]: math['Integer']
```

```
[8]: <OntologyClass math.Integer>
```

Fetching by label is NOT case sensitive when using the dot notation, but it is when using square brackets, so the following behavior is expected.

```
[9]: # math['integer']  # -> Fails.
     math.integer  # -> Works.
```

```
[9]: <OntologyClass math.integer>
```

Accessing an ontology entity by IRI

This is only possible using the `get_from_iri` method from the namespace object. For example, let's fetch the Integer entity again.

[10]:
```
math.get_from_iri('http://emmo.info/emmo/middle/math#EMMO_f8bd64d5_5d3e_4ad4_a46e_
↪c30714fecb7f')
```

[10]:
```
<OntologyClass math.Integer>
```

Access entities using a string

Sometimes you only have a string refering to an entity. Using the `get_entity` function you can get the corresponding python object easily:

[11]:
```
from osp.core.namespaces import get_entity  # noqa: E402

print("\nYou can get an entity with a string")
print(get_entity("city.LivingBeing"))
print(get_entity("city.LivingBeing") == city.LivingBeing)
```

```
You can get an entity with a string
city.LivingBeing
True
```

16.3 Accessing an entity's name, IRI and namespace

Each ontology entity has an associated name which can be accessed using the `name` property.

[12]:
```
city.LivingBeing.name
```

[12]:
```
'LivingBeing'
```

The IRI of an entity might be accessed using the `iri` property.

[13]:
```
math.Real.iri
```

[13]:
```
rdflib.term.URIRef('http://emmo.info/emmo/middle/math#EMMO_18d180e4_5e3e_42f7_820c_
↪e08951223486')
```

In addition, it is possible to get the namespace object to which the entity belongs using the `namespace` property.

[14]:
```
math.Equation.namespace
```

[14]:
```
<math: http://emmo.info/emmo/middle/math#>
```

16.4 Accessing super- and subclasses

Using the properties `superclasses` and `subclasses` it is easy to navigate the ontology. Direct superclasses and subclasses can also be accessed:

[15]:
```
print("\nYou can access the superclasses and the subclasses")
print(city.LivingBeing.superclasses)
print(city.LivingBeing.subclasses)

print("\nYou can access the direct superclasses and subclasses")
print(city.LivingBeing.direct_superclasses)
print(city.LivingBeing.direct_subclasses)

print("\nYou can access a description of the entities")
print(city.LivingBeing.description)

print("\nYou can test if one entity is a subclass / superclass of another")
print(city.Person.is_subclass_of(city.LivingBeing))
print(city.LivingBeing.is_superclass_of(city.Person))
```

```
You can access the superclasses and the subclasses
{<OntologyClass cuba.Entity>, <OntologyClass city.LivingBeing>}
{<OntologyClass city.Person>, <OntologyClass city.Citizen>, <OntologyClass city.
↪LivingBeing>}

You can access the direct superclasses and subclasses
{<OntologyClass cuba.Entity>}
{<OntologyClass city.Person>}

You can access a description of the entities
A being that lives

You can test if one entity is a subclass / superclass of another
True
True
```

16.5 Testing the type of the entities

In the ontology, three types of entities can be defined: classes, relationships and attributes. OSP-core has its own vocabulary, the *CUBA namespace*, which describes, among other things, such entity types. Relationships are subclasses of `CUBA.RELATIONSHIP` and attributes are subclasses of `CUBA.ATTRIBUTE`. There are different Python objects for the different entity types. You can use both to check which type of entity you are dealing with:

[16]:
```
from osp.core.namespaces import cuba  # noqa: E402

# These are the classes for the ontology entities
from osp.core.ontology import (  # noqa: F401, E402
    OntologyEntity,
    OntologyClass,
    OntologyRelationship,
```

(continues on next page)

(continued from previous page)

```
    OntologyAttribute
)

print("\nYou can test if an entity is a class")
print(isinstance(city.LivingBeing, OntologyClass))
print(not city.LivingBeing.is_subclass_of(cuba.relationship)
      and not city.LivingBeing.is_subclass_of(cuba.attribute))

print("\nYou can test if an entity is a relationship")
print(isinstance(city.hasInhabitant, OntologyRelationship))
print(city.hasInhabitant.is_subclass_of(cuba.relationship))

print("\nYou can test if an entity is a attribute")
print(isinstance(city.name, OntologyAttribute))
print(city.name.is_subclass_of(cuba.attribute))
```

```
You can test if an entity is a class
True
True

You can test if an entity is a relationship
True
True

You can test if an entity is a attribute
True
True
```

16.6 Operations specific to ontology classes

The different types of entities differ in the operations they offer. For classes, you can access the attributes:

[17]:
```
print("\nYou can get the attributes of an ontology class and their defaults")
print(city.Citizen.attributes)

print("\nYou can get the non-inherited attributes and their defaults")
print(city.Citizen.own_attributes)
print(city.LivingBeing.own_attributes)
```

```
You can get the attributes of an ontology class and their defaults
{<OntologyAttribute city.name>: (rdflib.term.Literal('John Smith'), False, None),
 ↪<OntologyAttribute city.age>: (rdflib.term.Literal('25', datatype=rdflib.term.URIRef(
 ↪'http://www.w3.org/2001/XMLSchema#integer')), False, None)}

You can get the non-inherited attributes and their defaults
{}
{<OntologyAttribute city.name>: (rdflib.term.Literal('John Smith'), False, None),
 ↪<OntologyAttribute city.age>: (rdflib.term.Literal('25', datatype=rdflib.term.URIRef(
 ↪'http://www.w3.org/2001/XMLSchema#integer')), False, None)}
```

(continues on next page)

(continued from previous page)

In addition, OSP-core has special support for the `owl:Restriction` and `owl:Composition` classes of the Web Ontology Language (OWL) (check the OWL ontology specification for more details). Such OWL classes are represented by the python classes `Restriction` and `Composition`. See *operations specific to ontology axioms* for more information.

For example, in the city ontology, the citizens have a restriction on the name and age attributes: a citizen must have exactly one name and one age. These axioms can be accessed using the `axioms` property, which returns both the restriction and compositions affecting the class.

[18]:
```
tuple(str(x) for x in city.Citizen.axioms)
```

[18]:
```
('city.name QUANTIFIER.EXACTLY 1', 'city.age QUANTIFIER.EXACTLY 1')
```

16.7 Operations specific to ontology axioms

For restrictions, the quantifier, the target, the restriction type and the relationship/attribute (depending on whether it is a restriction of the relationship type or attribute type) may be accessed.

[19]:
```
restriction = city.Citizen.axioms[0]
print(restriction)
print(restriction.quantifier)
print(restriction.target)
print(restriction.rtype)
print(restriction.attribute)
```

```
city.name QUANTIFIER.EXACTLY 1
QUANTIFIER.EXACTLY
1
RTYPE.ATTRIBUTE_RESTRICTION
city.name
```

For compositions, both the operator and operands can be accesed.

[20]:
```
from osp.core.ontology.oclass_composition import Composition
composition = tuple(x for x in math.Integer.axioms if type(x) is Composition)[0]
print(composition)
print(composition.operator)
print(composition.operands)
```

```
(math.Mathematical OPERATOR.AND perceptual.Symbol)
OPERATOR.AND
[<OntologyClass math.Mathematical>, <OntologyClass perceptual.Symbol>]
```

16.8 Operations specific to ontology relationships

You can access the inverse of relationships.

[21]:
```
print("\nYou can get the inverse of a relationship")
print(city.hasInhabitant.inverse)
```

```
You can get the inverse of a relationship
city.INVERSE_OF_hasInhabitant
```

16.9 Operations specific to attributes

You can acces the datatype and the argument name of attributes.

[22]:
```
print("\nYou can get the argument name of an attribute. "
      "The argument name is used as keyword argument when instantiating CUDS objects.")
print(city.age.argname)

print("\nYou can get the datatype of attributes")
print(city.age.datatype)

print("\nYou can use the attribute to convert values "
      "to the datatype of the attribute")
result = city.age.convert_to_datatype("10")
print(type(result), result)

print("\nAnd likewise to convert values to the python basic type "
      "associated with the datatype of the attribute.")
result = city.name.convert_to_basic_type(5)
print(type(result), result)
```

```
You can get the argument name of an attribute. The argument name is used as keyword␣
↪argument when instantiating CUDS objects.
age

You can get the datatype of attributes
http://www.w3.org/2001/XMLSchema#integer

You can use the attribute to convert values to the datatype of the attribute
<class 'int'> 10

And likewise to convert values to the python basic type associated with the datatype of␣
↪the attribute.
<class 'str'> 5
```

Check the API Reference for more details on the methods **convert_to_datatype** and **convert_to_basic_type**.

16.10 Creating CUDS using ontology classes

You can call ontology classes to create CUDS objects. To learn more, have a look at the *CUDS API tutorial*.

[23]:
```
print("\nYou can instantiate CUDS objects using ontology classes")
print(city.Citizen(name="Test Person", age=42))

print("\nYou can check if a CUDS object is an instance of a ontology class")
print(city.Citizen(name="Test Person", age=42).is_a(city.Citizen))
print(city.Citizen(name="Test Person", age=42).is_a(city.LivingBeing))

print("\nYou can get the ontology class of a CUDS object.")
print(city.Citizen(name="Test Person", age=42).oclass)
```

```
You can instantiate CUDS objects using ontology classes
city.Citizen: e0947100-9c40-415f-92c8-a86b796dbb01

You can check if a CUDS object is an instance of a ontology class
True
True

You can get the ontology class of a CUDS object.
city.Citizen
```

CHAPTER

SEVENTEEN

WRAPPER DEVELOPMENT

For an skeleton structure of a wrapper, you can visit the wrapper development repo. For a tutorial on creating a simple wrapper, there is a *jupyter notebook* available.

17.1 Ontology

The end goal is to build one, unique and standard ontology with all the relevant entities and relationships. This ontology could use modules where the entities regarding a certain domain are present.

However, for the development of a wrapper, it is usually more practical to create a minimal temporary ontology with the entities required by a wrapper. Once the development is in a more stable stage, the development and merge of a correct ontology can be done, and should not require major changes in the code.

These are the requirements for a minimal wrapper ontology:

- Should contain an entity representing the wrapper. Said entity should inherit from (subclass, is_a) `cuba.Wrapper`.
- All attributes should subclass `cuba.attribute`.
- Top level entities should subclass `cuba.Entity`
- Active relationships should subclass `cuba.ActiveRelationship`
- Passive relationships should subclass `cuba.PassiveRelationship`

```
---
version: "M.m"

author: Parmenides <parmenides@ontology.creator>

namespace: some_new_wrapper_ontology

ontology:
  aRelationship:
    description: "default relationship"
    subclass_of:
      - cuba.activeRelationship
    inverse: some_new_wrapper_ontology.pihsnoitalerA
    default_rel: true

  pihsnoitalerA:
    description: "inverse of the default relationship"
```

(continues on next page)

(continued from previous page)

```
  subclass_of:
    - cuba.passiveRelationship
  inverse: some_new_wrapper_ontology.aRelationship

################

SomeNewWrapper:
  subclass_of:
    - cuba.Wrapper

value:
  subclass_of:
    - cuba.attribute

SomeEntity:
  subclass_of:
    - cuba.Entity
```

17.2 Coding

An advantage of the 3-layered-design that we follow is the modularity and conceptual separation. The closer to the user, the higher the abstraction.

This allows us to group and clearly define which components should and which ones should not be modified when creating a new wrapper.

- *Semantic layer*: Requires no work. As presented in the previous section, only an entity representing the wrapper has to be present in the ontology.
- *Interoperability layer*:
  - *Session class*: Represents the bulk of the work that a wrapper developer needs to do. A new class inheriting from the appropriate Session Abstract Base Class must be coded. It should at least implement all the inherited abstract methods.
    - `__str__(self)`: String representation of the wrapper.
    - `_apply_added(self, root_obj, buffer)`: Add all the elements in `buffer` to the engine.
    - `_apply_updated(self, root_obj, buffer)`: Update all the elements in `buffer` in the engine.
    - `_apply_deleted(self, root_obj, buffer)`: Remove all the elements in `buffer` from the engine.
    - `_load_from_backend(self, uids, expired=None)`: Loads the given uids (and the dependant entities) with the latest information from the backend. Only loads the directly related information, not all the children recursively. This method *must* be implemented for a wrapper to work.
    - Specific for a simulation:
      - `_run(self, root_cuds_object)`: Call the run method on the simulation.
    - Specific for a database:
      - `_initialize(self)`: Initialise the database.
      - `_load_first_level(self)`: Load the first level of children from the root from the database.

- `_init_transaction(self)`: Start the transaction.
- `_rollback_transaction(self)`: Rollback the transaction.
- `close(self)`: Close the connection.

- *Syntactic layer*: If none is available, one must be developed.

To facilitate the creation of the session class on the interoperability layer, there are several session abstract base classes that you can make your session inherit from, which already include some additional generic functions common to a few typical applications: databases, triplestores and simulation engines.

On the diagram below, you may observe a simplified session inheritance scheme for OSP-core. As a wrapper developer, you will most probably want to inherit from one of following abstract classes: `WrapperSession`, `DbWrapperSession`, `TripleStoreWrapperSession`, `SqlWrapperSession`, or `SimWrapperSession`. Your new wrapper session would be located of the OSP-core box, together among other wrapper sessions like the Simlammps, Sqlite or SqlAlchemy sessions.

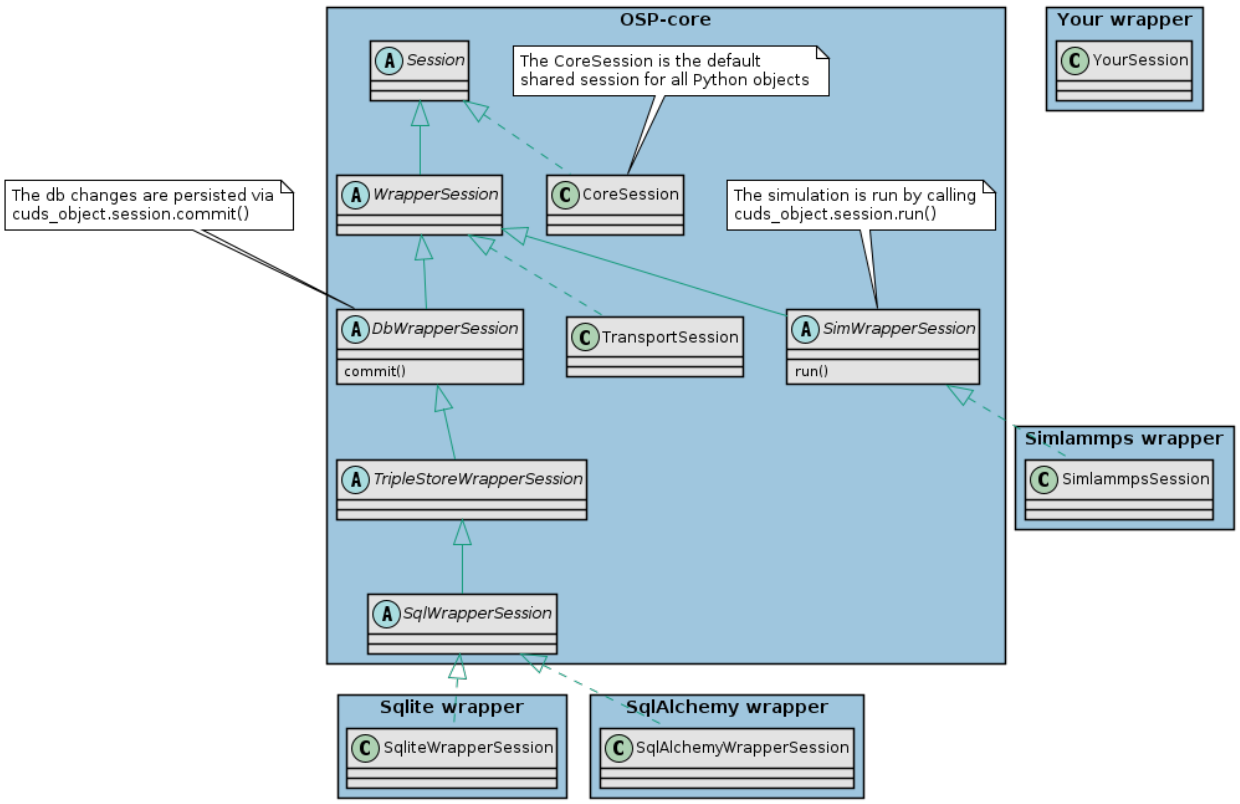


Fig. 1: Simplified session inheritance scheme

17.3 Engine installation

Most engines will require some sort of compilation or installation before being able to use them through Python.

To facilitate the installation of the backend to the end users, a shell script with the necessary commands should be made available. It is also recommended to split the installation of the engine from the installation of the engine requirements.

```
#!/bin/bash
#
# Author: Ada Lovelace <ada.lovelace@programmer.algorithm>
#
# Description: This script install the requirements for some engine
#              Used as part of the installation for SomeWrapper.
#
# Run Information: This script is called by install_engine.sh

echo "Installing necessary requirements for the engine"
platform=$(python3 -mplatform)

case $platform in
  *"Ubuntu"*)
    sudo apt-get update
    sudo apt-get install cmake
  ;;
  *"centos"*)
    sudo yum update
    sudo yum install make -y
    sudo yum install cmake -y
  ;;
  # Add other platforms here
esac
```

```
#!/bin/bash
#
# Author: Ada Lovelace <ada.lovelace@programmer.algorithm>
#
# Description: This script installs SomeEngine and its Python binding
#              Used as part of the installation for SomeWrapper.
#
# Run Information: This script is run manually.

###################################
### Install engine requirements ###
###################################
./install_engine_requirements.sh

#################################
### Download necessary files ###
#################################
echo "Checking out a recent stable version"
git clone some-repo.com/some-engine.git
cd some-engine
```

(continues on next page)

(continued from previous page)

```
  ############################
  ### Perform installation ###
  ############################
  cmake cmake
  make install

  ########################
  ### Test installation ###
  ########################
  {
      python3 -c 'from someEngine import engine; engine.test()'
  } || {
      echo "There was an error with the installation."
      echo "Please, try again or contact the developer."
  }
```

When the implementation of the wrapper is done, the user should be able to install all the necessary components via:

```
(env) user@computer:~/some_wrapper$ ./install_engine.sh
(env) user@computer:~/some_wrapper$ python setup.py install
```

17.3.1 Dockerfile with the engine

Apart from a system installation, we highly recommend providing a `Dockerfile` with the engine and other minimal requirements, in case the system installation is not possible or desired.

Similar to how OSP-core is the structure on top of which the wrappers are made, we designed a schema of Docker images where OSP-core is used as a base image.

Thus, OSP-core has an image (currently using Ubuntu) that should be tagged `simphony/osp-core:<VERSION>`. The Dockerfile of a wrapper will have that image in the `FROM` statement at the top, and take care of installing the engine requirements (and the wrapper itself).

To fix the tagging of the images and the versioning compatibility, the `Dockerfile` should be installed via the provided `docker_install.sh` script. It will tag the OSP-core image and call the Dockerfile in the root of the wrapper accordingly.

In terms of implementation, a wrapper developer needs to take care of the `Dockerfile`, making sure to leave the first two lines as they are in the wrapper development repo. `docker_install.sh` will only have to be modified with the proper tag for the wrapper image.

17.4 Continuous Integration

GitLab provides Continuous Integration via the `.gitlab-ci.yml` file. This should be used for checking both the style and the functionality of the code automatically after each commit.

If the wrapper requires the installation of an engine, it would probably be best to install it in a Docker image and push the image to Gitlab Container Registry so that the CI jobs use that image as the base system in which to run.

The `Dockerfile` for the Container Registry image will be very similar to the one used for installing the engine. However, here it might be useful to install other libraries like flake8 for style checks.

17.5 Utility functions for wrapper development

We have developed some functions that will probably come in handy when developing a wrapper. You can find them in osp.core.utils.wrapper_development.

17.6 Wrapper Examples

Some wrappers we are developing are:

- SQLAlchemy
- SQLite
- SimLammps
- SimGromacs
- SimOpenFoam
- Quantum Espresso

CHAPTER

EIGHTEEN

TUTORIAL: SIMPLE WRAPPER DEVELOPMENT

In this tutorial we will implement a very simple simulation wrapper. It can be used to understand which methods need to be implemented, and how.

The source files can be found here.

18.1 Background

Wrappers are the way to extend SimPhoNy to support other back-ends. For an in-depth explanation, you can go to the *wrapper development section* of the documentation. Here we will explain with more detail what has to be implemented.

18.2 Requirements

In order to run this code, you need to have the simple_ontology available here.

Remember that once you have OSP-core installed and the ontology file locally, you can simply run `pico install <path/to/ontology_file.yml>`

```
[ ]: # You can download and install the ontology by running this cell
!curl -s https://raw.githubusercontent.com/simphony/wrapper-development/master/osp/
↪wrappers/simple_simulation/simple_ontology.ontology.yml -o simple_ontology.ontology.yml
!pico install simple_ontology.ontology.yml
```

18.3 Let's get hands on

18.3.1 Syntactic layer

As you know, SimPhoNy consists of 3 layers, with the wrappers being relevant in the last 2 (interoperability and syntactic layers). The syntactic layer talks directly to the back-end in a way that it can be understood.

Since this wrapper aims to be as minimalistic as possible (while still being meaningfull), we have created a dummy syntactic layer that emulates talking to a simulation tool.

Note: In order to reduce the amount of code, the docstrings hav been erased. You can refer to the source file for the complete information.

[1]:
```
# This is the representation of an atom in the "engine"
class Atom():

    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity
```

The engine only works with atoms, setting and getting their position and velocities

[2]:
```
class SimulationEngine:
    def __init__(self):
        self.atoms = list()
        print("Engine instantiated!")

    def __str__(self):
        return "Some Engine Connection"

    def run(self, timesteps=1):
        print("Now the engine is running")
        for atom in self.atoms:
            atom.position += atom.velocity * timesteps

    def add_atom(self, position, velocity):
        print("Add atom %s with position %s and velocity %s"
              % (len(self.atoms), position, velocity))
        self.atoms.append(Atom(position, velocity))

    def update_position(self, idx, position):
        print("Update atom %s. Setting position to %s"
              % (idx, position))
        self.atoms[idx].position = position

    def update_velocity(self, idx, velocity):
        print("Update atom %s. Setting velocity to %s"
              % (idx, velocity))
        self.atoms[idx].velocity = velocity

    def get_velocity(self, idx):
        return self.atoms[idx].velocity

    def get_position(self, idx):
        return self.atoms[idx].position
```

18.3.2 Interoperability layer

Since a lot of 3rd-party tools come with a syntactic layer, the bulk of the work when developping a wrapper for SimPhoNy is here.

We will explain step by step all the code required.

First, we import the parent Simulation Wrapper Session and the namespace (ontology). The engine is not necessary since it is in the previous codebock.

[ ]:
```
from osp.core.session import SimWrapperSession
# from osp.wrappers.simple_simulation import SimulationEngine
from osp.core.namespaces import simple_ontology
```

Next, we will go through each of the methods.

Note: to be able to break the class into multiple blocks, we will use inheritance, to add a method each time. In truth, all the definitions should go under one same `class` definition.

The first method is the `__init__`. This method is called when a new object is instantiated. Here we will call the `__init__` method of the parent class and initialise the necessary elements.

Most simulation engines will have an internal way to keep track of, for example, particles. To make sure that the entities in the semantic layer are properly synched, we usually use a *mapper*. This could be anything from a list or dictionary to a more complex and sofisticated data structure.

[4]:
```
class SimpleSimulationSession(SimWrapperSession):

    def __init__(self, engine=None, **kwargs):
        super().__init__(engine or SimulationEngine(), **kwargs)
        self.mapper = dict()  # maps uuid to index in the backend
```

Next comes the output to the `str()` method. It will be a string returned in `__str__(self)`.

[5]:
```
class SimpleSimulationSession(SimpleSimulationSession):

    def __str__(self):
        return "Simple sample Wrapper Session"
```

When the `run()` or `commit()` method is called on the session, all the objects that have been added since the last run have to be sent to the back end. This is done through `_apply_added`. The method should iterate through all the entities in the buffer and trigger different actions depending on which type of entity it is.

Remember that we can check the type using the `is_a` method, or querying for the `oclass` attribute of an entity.

In this example, we will only contact the back end if an atom has been added. However, normal wrappers will have a lot more comparisons (`if` and `elif`) to determine which entity it is and act accordingly

[6]:
```
class SimpleSimulationSession(SimpleSimulationSession):

    # OVERRIDE
    def _apply_added(self, root_obj, buffer):
        """Adds the added cuds to the engine."""
        for obj in buffer.values():
            if obj.is_a(simple_ontology.Atom):
```

(continues on next page)

(continued from previous page)

```
                # Add the atom to the mapper
                self.mapper[obj.uid] = len(self.mapper)
                pos = obj.get(oclass=simple_ontology.Position)[0].value
                vel = obj.get(oclass=simple_ontology.Velocity)[0].value
                self._engine.add_atom(pos, vel)
```

Just like `_apply_added` is used to modify the engine with the new objects, `_apply_updated` changes the existing ones.

[7]:
```
class SimpleSimulationSession(SimpleSimulationSession):

    # OVERRIDE
    def _apply_updated(self, root_obj, buffer):
        """Updates the updated cuds in the engine."""
        for obj in buffer.values():

            # case 1: we change the velocity
            if obj.is_a(simple_ontology.Velocity):
                atom = obj.get(rel=simple_ontology.isPartOf)[0]
                idx = self.mapper[atom.uid]
                self._engine.update_velocity(idx, obj.value)

            # case 2: we change the position
            elif obj.is_a(simple_ontology.Position):
                atom = obj.get(rel=simple_ontology.isPartOf)[0]
                idx = self.mapper[atom.uid]
                self._engine.update_position(idx, obj.value)
```

Similarly to the previous methods, `_apply_deleted` should remove entities from the engine. In this specific case we left it empty to simplify the code (both in the session and the engine classes).

[8]:
```
class SimpleSimulationSession(SimpleSimulationSession):
    # OVERRIDE
    def _apply_deleted(self, root_obj, buffer):
        """Deletes the deleted cuds from the engine."""
```

The previous methods synchronise the engine with the cuds, i.e. the communication is from the semantic layer towards the syntactic. The way to update the cuds with the latest information from the engine is `_load_from_backend`.

It is most often called when the user calls the `get` on a cuds object that has potentially been changed by the engine.

When `_load_from_backend` is called for a given cuds object (through its uid), the method should: - Check if any of the attributes of the object has changed (like the *value* for a *position*). - Check if any new children cuds objects have been created (like a static *atom* that gets a new *velocity* when another bumps into it).

However, it does not have to be recursive and check for more than itself. This is because if the user queries any of the contained elements, that will trigger another call to `_load_from_backend`.

[9]:
```
class SimpleSimulationSession(SimpleSimulationSession):

    # OVERRIDE
    def _load_from_backend(self, uids, expired=None):
        """Loads the cuds object from the simulation engine"""
```

(continues on next page)

(continued from previous page)

```
        for uid in uids:
            if uid in self._registry:
                obj = self._registry.get(uid)

                # check whether user wants to load a position
                if obj.is_a(simple_ontology.Position):
                    atom = obj.get(rel=simple_ontology.isPartOf)[0]
                    idx = self.mapper[atom.uid]
                    pos = self._engine.get_position(idx)
                    obj.value = pos

                # check whether user wants to load a velocity
                elif obj.is_a(simple_ontology.Velocity):
                    atom = obj.get(rel=simple_ontology.isPartOf)[0]
                    idx = self.mapper[atom.uid]
                    vel = self._engine.get_velocity(idx)
                    obj.value = vel

                yield obj
```

The last method that needs to be overridden is `_run`. It simply has to call the `run` method of the engine. This could also need to send some information, like the number of steps. For that reason, the `root_cuds_object` is available for query.

[10]:
```
class SimpleSimulationSession(SimpleSimulationSession):

    # OVERRIDE
    def _run(self, root_cuds_object):
        """Call the run command of the engine."""
        self._engine.run()
```

Now we can run an example:

[11]:
```
from osp.core.utils import pretty_print
import numpy as np

m = simple_ontology.Material()
for i in range(3):
    a = m.add(simple_ontology.Atom())
    a.add(
        simple_ontology.Position(value=[i, i, i], unit="m"),
        simple_ontology.Velocity(value=np.random.random(3), unit="m/s")
    )

# Run a simulation
with SimpleSimulationSession() as session:
    w = simple_ontology.Wrapper(session=session)
    m = w.add(m)
    w.session.run()

    pretty_print(m)
```

(continues on next page)

(continued from previous page)

```
    for atom in m.get(rel=simple_ontology.hasPart):
        atom.get(oclass=simple_ontology.Velocity)[0].value = [0, 0, 0]
    w.session.run()

    pretty_print(m)
```

```
Engine instantiated!
Add atom 0 with position [0. 0. 0.] and velocity [0.63000616 0.38951439 0.12717548]
Add atom 1 with position [1. 1. 1.] and velocity [0.80816851 0.04562681 0.44983098]
Add atom 2 with position [2. 2. 2.] and velocity [0.3849223  0.50767213 0.82963311]
Now the engine is running
- Cuds object:
  uuid: a0a97dbe-584d-4764-b085-7b597e323d20
  type: simple_ontology.Material
  superclasses: cuba.Entity, simple_ontology.Material
  description:
    To Be Determined

   |_Relationship simple_ontology.hasPart:
     -  simple_ontology.Atom cuds object:
     .  uuid: 221a1793-c54a-4e42-bdeb-08921617fbac
     .   |_Relationship simple_ontology.hasPart:
     .     -  simple_ontology.Position cuds object:
     .     .  uuid: db17082e-d9d3-4a48-bce1-9402d4315200
     .     .  unit: m
     .     .  value: [0.63000616 0.38951439 0.12717548]
     .     -  simple_ontology.Velocity cuds object:
     .        uuid: fc7d778d-b18a-4b60-a6ab-ba855a2c2874
     .        value: [0.63000616 0.38951439 0.12717548]
     .        unit: m/s
     -  simple_ontology.Atom cuds object:
     .  uuid: 222df5d0-c0fe-435b-b5e2-0d5f7ebd32a9
     .   |_Relationship simple_ontology.hasPart:
     .     -  simple_ontology.Position cuds object:
     .     .  uuid: f0eb08c4-88a3-40de-893f-89473cd194e8
     .     .  value: [1.80816851 1.04562681 1.44983098]
     .     .  unit: m
     .     -  simple_ontology.Velocity cuds object:
     .        uuid: d3e7b5ce-3409-4a4e-bbdb-13a2addaee1c
     .        value: [0.80816851 0.04562681 0.44983098]
     .        unit: m/s
     -  simple_ontology.Atom cuds object:
        uuid: 13bfe4ee-32e8-4fbe-bad5-f98f46aa297a
         |_Relationship simple_ontology.hasPart:
           -  simple_ontology.Position cuds object:
           .  uuid: da5c35f0-afe5-4b56-b5fa-631b72ee32ad
           .  value: [2.3849223  2.50767213 2.82963311]
           .  unit: m
           -  simple_ontology.Velocity cuds object:
              uuid: 9b6c2c1c-3e63-4144-b7c9-d6223c0b79f7
              value: [0.3849223  0.50767213 0.82963311]
```

(continues on next page)

(continued from previous page)

```
             unit: m/s
Update atom 0. Setting velocity to [0. 0. 0.]
Update atom 1. Setting velocity to [0. 0. 0.]
Update atom 2. Setting velocity to [0. 0. 0.]
Now the engine is running
- Cuds object:
  uuid: a0a97dbe-584d-4764-b085-7b597e323d20
  type: simple_ontology.Material
  superclasses: cuba.Entity, simple_ontology.Material
  description:
    To Be Determined

   |_Relationship simple_ontology.hasPart:
     -  simple_ontology.Atom cuds object:
     .  uuid: 221a1793-c54a-4e42-bdeb-08921617fbac
     .   |_Relationship simple_ontology.hasPart:
     .     -  simple_ontology.Position cuds object:
     .     .  uuid: db17082e-d9d3-4a48-bce1-9402d4315200
     .     .  unit: m
     .     .  value: [0.63000616 0.38951439 0.12717548]
     .     -  simple_ontology.Velocity cuds object:
     .        uuid: fc7d778d-b18a-4b60-a6ab-ba855a2c2874
     .        value: [0. 0. 0.]
     .        unit: m/s
     -  simple_ontology.Atom cuds object:
     .  uuid: 222df5d0-c0fe-435b-b5e2-0d5f7ebd32a9
     .   |_Relationship simple_ontology.hasPart:
     .     -  simple_ontology.Position cuds object:
     .     .  uuid: f0eb08c4-88a3-40de-893f-89473cd194e8
     .     .  value: [1.80816851 1.04562681 1.44983098]
     .     .  unit: m
     .     -  simple_ontology.Velocity cuds object:
     .        uuid: d3e7b5ce-3409-4a4e-bbdb-13a2addaee1c
     .        unit: m/s
     .        value: [0. 0. 0.]
     -  simple_ontology.Atom cuds object:
        uuid: 13bfe4ee-32e8-4fbe-bad5-f98f46aa297a
         |_Relationship simple_ontology.hasPart:
           -  simple_ontology.Position cuds object:
           .  uuid: da5c35f0-afe5-4b56-b5fa-631b72ee32ad
           .  value: [2.3849223  2.50767213 2.82963311]
           .  unit: m
           -  simple_ontology.Velocity cuds object:
              uuid: 9b6c2c1c-3e63-4144-b7c9-d6223c0b79f7
              value: [0. 0. 0.]
              unit: m/s
```

CHAPTER

NINETEEN

API REFERENCE

This document is for developers and/or advanced users of OSP-core, it contains all API details.

19.1 CUDS

class osp.core.cuds.**Cuds**(*attributes: Dict[*osp.core.ontology.attribute.OntologyAttribute*, Any]*, *oclass: Optional[*osp.core.ontology.oclass.OntologyClass*] = None*, *session: Optional[*osp.core.session.session.Session*] = None*, *iri: Optional[rdflib.term.URIRef] = None*, *uid: Optional[Union[uuid.UUID, rdflib.term.URIRef]] = None*, *extra_triples: Iterable[Tuple[Union[rdflib.term.URIRef, rdflib.term.BNode], Union[rdflib.term.URIRef, rdflib.term.BNode], Union[rdflib.term.URIRef, rdflib.term.BNode]]] = ()*)

Bases: `object`

A Common Universal Data Structure.

The CUDS object is an ontology individual that can be used like a container. It has attributes and is connected to other cuds objects via relationships.

add(**args:* osp.core.cuds.Cuds, *rel: Optional[*osp.core.ontology.relationship.OntologyRelationship*] = None*) → Union[*osp.core.cuds.Cuds*, List[*osp.core.cuds.Cuds*]]

Add CUDS objects to their respective relationship.

If the added objects are associated with the same session, only a link is created. Otherwise, the a deep-copy is made and added to the session of this Cuds object. Before adding, check for invalid keys to avoid inconsistencies later.

Parameters

- **args** (Cuds) – The objects to be added
- **rel** (OntologyRelationship) – The relationship between the objects.

Raises

- **TypeError** – Ne relationship given and no default specified.
- **ValueError** – Added a CUDS object that is already in the container.

Returns

The CUDS objects that have been added, associated with the session of the current CUDS object. Result type is a list, if more than one CUDS object is returned.

Return type Union[*Cuds*, List[*Cuds*]]

get(**uids: typing.Union[uuid.UUID, rdflib.term.URIRef], rel: osp.core.ontology.relationship.OntologyRelationship = <OntologyRelationship cuba.activeRelationship>, oclass: typing.Optional[osp.core.ontology.oclass.OntologyClass] = None, return_rel: bool = False*) → Union[*osp.core.cuds.Cuds*, List[*osp.core.cuds.Cuds*]]

Return the contained elements.

Filter elements by given type, uid or relationship. Expected calls are get(), get(*uids), get(rel), get(oclass), get(*indentifiers, rel), get(rel, oclass). If uids are specified:

The position of each element in the result is determined by to the position of the corresponding uid in the given list of uids. In this case, the result can contain None values if a given uid is not a child of this cuds_object. If only a single indentifier is given, only this one element is returned (i.e. no list).

If no uids are specified: The result is a collection, where the elements are ordered randomly.

Parameters

- **uids** (*Union[UUID, URIRef]*) – uids of the elements.
- **rel** (OntologyRelationship, *optional*) – Only return cuds_object which are connected by subclass of given relationship. Defaults to cuba.activeRelationship.
- **oclass** (OntologyClass, *optional*) – Only return elements which are a subclass of the given ontology class. Defaults to None.
- **return_rel** (*bool, optional*) – Whether to return the connecting relationship. Defaults to False.

Returns The queried objects.

Return type Union[*Cuds*, List[*Cuds*]]

get_attributes()

Get the attributes as a dictionary.

get_triples(*include_neighbor_types=False*)

Get the triples of the cuds object.

property iri: rdflib.term.URIRef

Get the IRI of the CUDS object.

is_a(*oclass*)

Check if the CUDS object is an instance of the given oclass.

Parameters **oclass** (OntologyClass) – Check if the CUDS object is an instance of this oclass.

Returns Whether the CUDS object is an instance of the given oclass.

Return type bool

iter(**uids: typing.Union[uuid.UUID, rdflib.term.URIRef], rel: osp.core.ontology.relationship.OntologyRelationship = <OntologyRelationship cuba.activeRelationship>, oclass: typing.Optional[osp.core.ontology.oclass.OntologyClass] = None, return_rel: bool = False*) → Iterator[*osp.core.cuds.Cuds*]

Iterate over the contained elements.

Only iterate over objects of a given type, uid or oclass.

Expected calls are iter(), iter(*uids), iter(rel), iter(oclass), iter(*uids, rel), iter(rel, oclass). If uids are specified:

The position of each element in the result is determined by to the position of the corresponding uid in the given list of uids. In this case, the result can contain None values if a given uid is not a child of this cuds_object.

If no uids are specified: The result is ordered randomly.

Parameters

- **uids** (*Union[UUID, URIRef]*) – uids of the elements.
- **rel** (*OntologyRelationship, optional*) – Only return cuds_object which are connected by subclass of given relationship. Defaults to cuba.activeRelationship.
- **oclass** (*OntologyClass, optional*) – Only return elements which are a subclass of the given ontology class. Defaults to None.
- **return_rel** (*bool, optional*) – Whether to return the connecting relationship. Defaults to False.

Returns The queried objects.

Return type Iterator[*Cuds*]

property oclass

Get the type of the cuds object.

property oclasses

Get the ontology classes of this CUDS object.

remove(**args: typing.Union[osp.core.cuds.Cuds, uuid.UUID, rdflib.term.URIRef], rel: osp.core.ontology.relationship.OntologyRelationship = <OntologyRelationship cuba.activeRelationship>, oclass: typing.Optional[osp.core.ontology.oclass.OntologyClass] = None*)

Remove elements from the CUDS object.

Expected calls are remove(), remove(*uids/Cuds), remove(rel), remove(oclass), remove(*uids/Cuds, rel), remove(rel, oclass)

Parameters

- **args** (*Union[Cuds, UUID, URIRef]*) – UUIDs of the elements to remove or the elements themselves.
- **rel** (*OntologyRelationship, optional*) – Only remove cuds_object which are connected by subclass of given relationship. Defaults to cuba.activeRelationship.
- **oclass** (*OntologyClass, optional*) – Only remove elements which are a subclass of the given ontology class. Defaults to None.

Raises **RuntimeError** – No CUDS object removed, because specified CUDS objects are not in the container of the current CUDS object directly.

property session: *osp.core.session.session.Session*

Get the session of the cuds object.

property uid: Union[rdflib.term.URIRef, uuid.UUID]

Get the uid of the CUDS object.

This is the public getter of the property.

update(**args:* osp.core.cuds.Cuds) → List[*osp.core.cuds.Cuds*]

Update the Cuds object.

Updates the object by providing updated versions of CUDS objects that are directly in the container of this CUDS object. The updated versions must be associated with a different session.

Parameters **args** (Cuds) – The updated versions to use to update the current object.

Raises

- **ValueError** – Provided a CUDS objects is not in the container of the current CUDS
- **ValueError** – Provided CUDS object is associated with the same session as the current CUDS object. Therefore it is not an updated version.

Returns

The CUDS objects that have been updated, associated with the session of the current CUDS object. Result type is a list, if more than one CUDS object is returned.

Return type Union[*Cuds*, List[*Cuds*]]

19.2 Ontology interface

class osp.core.ontology.namespace.**OntologyNamespace**(*name*, *namespace_registry*, *iri*)

Bases: object

A namespace in the ontology.

__contains__(*item*)

Check whether the given entity is part of the namespace.

Parameters **item** (*Union[str, rdflib.URIRef, OntologyEntity, rdflib.BNode]*) – The name, IRI of an entity, the entity itself or a blank node.

Returns

Whether the given entity name or IRI is part of the namespace. Blank nodes are never part of a namespace.

Return type bool

__eq__(*other*)

Check whether the two namespace objects are the same.

Parameters **other** (OntologyNamespace) – The namespace to compare with.

Returns Whether the given namespace is the same.

Return type bool

__getattr__(*name*)

Get an ontology entity from the registry by label or suffix.

Parameters **name** (*str*) – The label or namespace suffix of the ontology entity.

Raises **AttributeError** – Unknown label or suffix.

Returns The ontology entity.

Return type *OntologyEntity*

__getitem__(*label*)

Get an ontology entity from the registry by label.

Useful for entities whose labels contains characters which are not compatible with the Python syntax.

Parameters **label** (*str*) – The label of the ontology entity.

Raises **KeyError** – Unknown label.

Returns The ontology entity.

Return type *OntologyEntity*

__iter__()

Iterate over the ontology entities in the namespace.

Returns An iterator over the entities.

Return type Iterator[*OntologyEntity*]

get(*name*, *fallback=None*)

Get an ontology entity from the registry by suffix or label.

Parameters

- **name** (*str*) – The label or suffix of the ontology entity.
- **default** (*Any*) – The value to return if it doesn't exist.
- **fallback** (*Any*) – The fallback value, defaults to None.

Returns The ontology entity

Return type *OntologyEntity*

get_default_rel()

Get the default relationship of the namespace.

get_from_iri(*iri*, *_name=None*)

Get an ontology entity directly from its IRI.

For consistency, this method only returns entities from this namespace.

Parameters

- **iri** (*Union[str, rdlib.URIRef]*) – The iri of the ontology entity.
- **_name** (*str*) – Not mean to be provided by the user. Just passed to the *from_iri* method of the namespace registry.

Returns The ontology entity.

Return type *OntologyEntity*

Raises **KeyError** – When the iri does not belong to the namespace.

get_from_suffix(*suffix*, *case_sensitive=False*)

Get an ontology entity from its namespace suffix.

Parameters

- **suffix** (*str*) – Suffix of the ontology entity.
- **case_sensitive** (*bool*) – Whether to search also for the same suffix with different capitalization. By default, such a search is performed.

get_iri()

Get the IRI of the namespace.

get_name()

Get the name of the namespace.

class osp.core.ontology.entity.**OntologyEntity**(*namespace_registry*, *namespace_iri*, *name*, *iri_suffix*)

Bases: abc.ABC

Abstract superclass of any entity in the ontology.

property description

Get the description of the entity.

Returns The description of the entity

Return type str

property direct_subclasses

Get the direct subclasses of the entity.

Returns The direct subclasses of the entity

Return type Set[*OntologyEntity*]

property direct_superclasses

Get the direct superclass of the entity.

Returns The direct superclasses of the entity

Return type List[*OntologyEntity*]

get_triples()

Get the triples of the entity.

property iri

Get the IRI of the Entity.

is_subclass_of(*other*)

Perform a subclass check.

Parameters **other** (*Entity*) – The other entity.

Returns Whether self is a subclass of other.

Return type bool

is_superclass_of(*other*)

Perform a superclass check.

Parameters **other** (*Entity*) – The other entity.

Returns Whether self is a superclass of other.

Return type bool

property name

Get the name of the entity.

property namespace

Get the namespace object of the entity.

property subclasses

Get the subclasses of the entity.

Returns The direct subclasses of the entity

Return type Set[*OntologyEntity*]

property superclasses

Get the superclass of the entity.

Returns The direct superclasses of the entity

Return type Set[*OntologyEntity*]

property tblname

Get the name used in storage backends to store instances.

class osp.core.ontology.oclass.**OntologyClass**(*namespace_registry*, *namespace_iri*, *name*, *iri_suffix*)

Bases: *osp.core.ontology.entity.OntologyEntity*

A class defined in the ontology.

property attributes

Get all the attributes of this oclass.

Returns Mapping from attribute to default

Return type Dict[*OntologyAttribute*, Any]

property axioms

Get all the axioms for the ontology class.

Include axioms of superclasses.

Returns The list of axioms for the ontology class.

Return type List[*Restriction*]

get_attribute_by_argname(*name*)

Get the attribute object with the argname of the object.

Parameters **name** (*str*) – The argname of the attribute

Returns The attribute

Return type *OntologyAttribute*

get_attribute_identifier_by_argname(*name*)

Get the attribute identifier with the argname of the object.

Parameters **name** (*str*) – The argname of the attribute

Returns The attribute identifier.

Return type Identifier

property own_attributes

Get the non-inherited attributes of this oclass.

Returns Mapping from attribute to default

Return type Dict[*OntologyAttribute*, str]

class osp.core.ontology.oclass_restriction.**Restriction**(*bnode*, *namespace_registry*)

Bases: object

A class to represet restrictions on ontology classes.

property attribute

The attribute the restriction acts on.

Only for ATTRIBUTE_RESTRICTIONs.

Raises **AttributeError** – self is a RELATIONSHIP_RESTRICTIONs.

Returns The datatype of the attribute.

Return type UriRef

property quantifier

Get the quantifier of the restriction.

Returns The quantifier of the restriction.

Return type QUANTIFIER

property relationship

The relationship the RELATIONSHIP_RESTRICTION acts on.

Raises **AttributeError** – Called on an ATTRIBUTE_RESTRICTION.

Returns The relationship the restriction acts on.

Return type *OntologyRelationship*

property rtype

Return the type of restriction.

Whether the restriction acts on attributes or relationships.

Returns The type of restriction.

Return type RTYPE

property target

The target ontology class or datatype.

Returns The target class or datatype.

Return type Union[*OntologyClass*, UriRef]

class osp.core.ontology.oclass_composition.**Composition**(*bnode*, *namespace_registry*)

Bases: object

Combine multiple classes using logical formulae.

property operands

The individual classes the formula is composed of.

Returns The operands.

Return type Union[*OntologyClass*, *Composition*, *Restriction*]

property operator

The operator that connects the different classes in the formula.

Returns The operator Enum.

Return type OPERATOR

class osp.core.ontology.relationship.**OntologyRelationship**(*namespace_registry*, *namespace_iri*, *name*, *iri_suffix*)

Bases: *osp.core.ontology.entity.OntologyEntity*

A relationship defined in the ontology.

property inverse

Get the inverse of this relationship.

If it doesn't exist, add one to the graph.

Returns The inverse relationship.

Return type *OntologyRelationship*

class osp.core.ontology.attribute.**OntologyAttribute**(*namespace_registry*, *namespace_iri*, *name*, *iri_suffix*)

Bases: *osp.core.ontology.entity.OntologyEntity*

An attribute defined in the ontology.

property argname

Get the name of the attribute when used as an argument.

This name is used when construction a cuds object or accessing the attributes of a CUDS object.

convert_to_basic_type(*value*)

Convert from the datatype of the value to a python basic type.

Parameters **value** (*Any*) – The value to convert

Returns The converted value

Return type Any

convert_to_datatype(*value*)

Convert to the datatype of the value.

Parameters **value** (*Any*) – The value to convert

Returns The converted value

Return type Any

property datatype

Get the datatype of the attribute.

Returns IRI of the datatype

Return type URIRef

Raises **RuntimeError** – More than one datatype associated with the attribute. # TODO should be allowed

19.3 Sessions

class osp.core.session.session.**Session**

Bases: abc.ABC

Abstract Base Class for all Sessions.

Defines the common standard API and sets the registry.

close()

Close the connection to the backend.

delete_cuds_object(*cuds_object*)

Remove a CUDS object.

Will not delete the cuds objects contained.

Parameters **cuds_object** (Cuds) – The CUDS object to be deleted

prune(*rel=None*)

Remove all elements not reachable from the sessions root.

Only consider given relationship and its subclasses.

Parameters **rel** (*Relationship, optional*) – Only consider this relationship to calculate reachability.. Defaults to None.

class osp.core.session.core_session.**CoreSession**

Bases: *osp.core.session.session.Session*, osp.core.session.sparql_backend.SPARQLBackend

Core default session for all objects.

class CoreSessionSparqlBindingSet(*row*, *session*)

Bases: osp.core.session.sparql_backend.SparqlBindingSet

A row in the result. Mapping from variable to value.

class CoreSessionSparqlResult(*query_result*, *session*)

Bases: osp.core.session.sparql_backend.SparqlResult

The result of a SPARQL query on the core session.

close()

Close the connection.

class osp.core.session.wrapper_session.**WrapperSession**(*engine*)

Bases: *osp.core.session.session.Session*

Common class for all wrapper sessions.

Sets the engine and creates the sets with the changed elements

static compute_auth(*username*, *password*, *handshake*)

Will be called on the client, after the handshake.

This method should produce an object that is able to authenticate the user. The __init__() method of the session should have a keyword "auth", that will have the output of this function as a value. –> The user can be authenticated on __init__()

Parameters

- **username** (*str*) – The username as encoded in the URI.

- **password** (*str*) – The password as encoded in the URI.
- **handshake** (*Any*) – The result of the hanshake method.

Returns Any JSON serializable object that is able to authenticate the user.

Return type Any

expire(**cuds_or_uids*)

Let cuds_objects expire.

Expired objects will be reloaded lazily when attributed or relationships are accessed.

Parameters

- ***cuds_or_uids** (*Union[Cuds, UUID, URIRef]*) – The cuds_object
- **expire.** (*or uids to*) –

Returns The set of uids that became expired

Return type Set[UUID]

expire_all()

Let all cuds_objects of the session expire.

Expired objects will be reloaded lazily when attributed or relationships are accessed.

Returns The set of uids that became expired

Return type Set[UUID]

static handshake(*username*, *connection_id*)

Will be called on the server, before anything else.

Result of this method will be fed into compute_auth() below, that will be executed by the client.

Parameters

- **username** (*str*) – The username of the user, as encoded in the URL.
- **connection_id** (*UUID*) – A UUID for the connection.

Returns

Any JSON serializable object that should be fed into compute_auth().

Return type Any

log_buffer_status(*context*)

Log the current status of the buffers.

Parameters **context** (*BufferContext*) – whether to print user or engine buffers

refresh(**cuds_or_uids*)

Refresh cuds_objects.

Load possibly updated data of cuds_object from the backend.

Parameters ***cuds_or_uids** (*Union[Cuds, UUID]*) – The cuds_object or uids to refresh.

class osp.core.session.sim_wrapper_session.**SimWrapperSession**(*engine*, ***kwargs*)

Bases: *osp.core.session.wrapper_session.WrapperSession*

Abstract class used for simulation sessions.

Contains methods necessary for all simulation sessions.

class osp.core.session.db.db_wrapper_session.**DbWrapperSession**(*engine*)

Bases: *osp.core.session.wrapper_session.WrapperSession*

Abstract class for a DB Wrapper Session.

abstract close()

Close the connection to the database.

static compute_auth(*username*, *password*, *handshake*)

Will be called on the client, after the handshake.

This method should produce an object that is able to authenticate the user. The __init__() method of the session should have a keyword "auth", that will have the output of this function as a value. –> The user can be authenticated on __init__()

Parameters

- **username** (*str*) – The username as encoded in the URI.
- **password** (*str*) – The password as encoded in the URI.
- **handshake** (*Any*) – The result of the hanshake method.

Returns Any JSON serializable object that is able to authenticate the user.

Return type Any

class osp.core.session.db.sql_wrapper_session.**SqlWrapperSession**(*engine*)

Bases: osp.core.session.db.triplestore_wrapper_session.TripleStoreWrapperSession

Abstract class for an SQL DB Wrapper Session.

check_schema()

Raise an error if sql session has data in not-supported.

Parameters **()** (*sql_session*) – [description]

Raises **RuntimeError** – [description]

19.4 Registry

class osp.core.session.registry.**Registry**

Bases: dict

A dictionary that contains all local CUDS objects.

filter(*criterion*)

Filter the registry.

Return a dictionary that is a subset of the registry. It contains only cuds objects that satisfy the given criterion.

Parameters **criterion** (*Callable[Cuds, bool]*) – A function that decides whether a cuds object should be returned. If the function returns True on a cuds object it means the cuds object satisfies the criterion.

Returns

dict contains the cuds objects satisfying the criterion.

Return type Dict[Union[UUID, URIRef], *Cuds*]

`filter_by_attribute`(*attribute*, *value*)

Filter by attribute and value.

Parameters

- **attribute** (*str*) – The attribute to look for.
- **value** (*Any*) – The corresponding value to look for.

Returns

A subset of the registry, containing cuds objects with given attribute and value.

Return type Dict[Union[UUID, URIRef], *Cuds*]

`filter_by_oclass`(*oclass*)

Filter the registry by ontology class.

Parameters **oclass** (OntologyClass) – The oclass used for filtering.

Returns

A subset of the registry, containing cuds objects with given ontology class.

Return type Dict[Union[UUID, URIRef], *Cuds*]

`filter_by_relationships`(*relationship*, *consider_subrelationships=False*)

Filter the registry by relationships.

Return cuds objects containing the given relationship.

Parameters

- **relationship** (OntologyRelationship) – The relationship to filter by.
- **consider_subrelationships** (*bool, optional*) – Whether to return CUDS objects containing subrelationships of the given relationship. Defaults to False.

Returns

A subset of the registry, containing cuds objects with given relationship.

Return type Dict[Union[UUID, URIRef], *Cuds*]

`get`(*uid*)

Return the object corresponding to a given uid.

Parameters **uid** (*Union[UUID, URIRef]*) – The uid of the desired object.

Raises **ValueError** – Unsupported key provided (not a uid object).

Returns Cuds object with the uid.

Return type *Cuds*

`get_subtree`(*root*, *subtree=None*, *rel=None*, *skip=None*, *warning=None*)

Get all the elements in the subtree rooted at given root.

Only use the given relationship for traversal.

Parameters

- **root** (*Union[UUID, URIRef,* Cuds*]*) – The root of the subtree.
- **rel** (*Relationship, optional*) – The relationship used for traversal. Defaults to None. Defaults to None.
- **subtree** (*Set[*Cuds*]*) – Currently calculated subtree (this is a recursive algorithm).

- **skip** (*Set[Cuds], optional*) – The elements to skip. Defaults to None. Defaults to None.
- **warning** (*LargeDatasetWarning, optional*) – Raise a *LargeDatasetWarning* when the subtree is large. When *None*, no warning is raised. If you wish to raise the warning, a *LargeDatasetWarning* object must be provided.

Returns

The set of elements in the subtree rooted in the given uid.

Return type Set[*Cuds*]

prune(**roots*, *rel=None*)

Remove all elements in the registry that are not reachable.

Parameters **rel** (*Relationship, optional*) – Only consider this relationship. Defaults to None.

Returns The set of removed elements.

Return type List[*Cuds*]

put(*cuds_object*)

Add an object to the registry.

Parameters **cuds_object** (Cuds) – The cuds_object to put in the registry.

Raises **ValueError** – Unsupported object provided (not a Cuds object).

reset()

Delete the contents of the registry.

19.5 Utilities

class osp.core.utils.**Cuds2dot**(*root*)

Utility for creating a dot and png representation of CUDS objects.

render(*filename=None*, ***kwargs*)

Create the graph and save it to a dot and png file.

static shorten_uid(*uid*)

Shortens the string of a given uid.

Parameters **uid** (*UUID*) – uuid to shorten.

Returns 8 first and 3 last characters separated by '...'.

Return type str

osp.core.utils.**branch**(*cuds_object*, **args*, *rel=None*)

Like Cuds.add(), but returns the element you add to.

This makes it easier to create large CUDS structures.

Parameters

- **cuds_object** (Cuds) – the object to add to.
- **args** (Cuds) – object(s) to add
- **rel** (OntologyRelationship) – class of the relationship between the objects.

Raises **ValueError** – adding an element already there.

Returns The first argument.

Return type *Cuds*

osp.core.utils.**change_oclass**(*cuds_object*, *new_oclass*, *kwargs*, *_force=False*)

Change the oclass of a cuds object.

Only allowed if cuds object does not have any neighbors.

Parameters

- **cuds_object** (Cuds) – The cuds object to change the oclass of
- **new_oclass** (OntologyClass) – The new oclass of the cuds object
- **kwargs** (*Dict[str, Any]*) – The keyword arguments used to instantiate cuds object of the new oclass.

Returns The cuds object with the changed oclass

Return type *Cuds*

osp.core.utils.**check_arguments**(*types*, **args*)

Check that the arguments provided are of the certain type(s).

Parameters

- **types** (*Union[Type, Tuple[Type]]*) – tuple with all the allowed types
- **args** (*Any*) – instances to check

Raises **TypeError** – if the arguments are not of the correct type

osp.core.utils.**clone_cuds_object**(*cuds_object*)

Avoid that the session gets copied.

Returns A copy of self with the same session.

Return type *Cuds*

osp.core.utils.**create_from_cuds_object**(*cuds_object*, *session*)

Create a copy of the given cuds_object in a different session.

WARNING: Will not recursively copy children.

Parameters

- **cuds_object** (Cuds) – The cuds object to copy
- **session** (Session) – The session of the new Cuds object

Returns A copy of self with the given session.

Return type *Cuds*

osp.core.utils.**create_from_triples**(*triples*, *neighbor_triples*, *session*, *fix_neighbors=True*)

Create a CUDS object from triples.

Parameters

- **triples** (*List[Tuple]*) – The list of triples of the CUDS object to create.
- **neighbor_triples** (*List[Tuple]*) – A list of important triples of neighbors, most importantly their types.
- **session** (Session) – The session to create the CUDS object in.

- **fix_neighbors** (*bool*) – Whether to remove the link from the old neighbors to this cuds object, defaults to True.

osp.core.utils.**create_recycle**(*oclass*, *kwargs*, *session*, *uid*, *fix_neighbors=True*, *_force=False*)

Instantiate a cuds_object with a given session.

If cuds_object with same uid is already in the session, this object will be reused.

Parameters

- **oclass** (Cuds) – The OntologyClass of cuds_object to instantiate
- **kwargs** (*Dict[str, Any]*) – The kwargs of the cuds_object
- **session** (Session) – The session of the new Cuds object
- **uid** (*Union[UUID, URIRef]*) – The uid of the new Cuds object
- **fix_neighbors** (*bool*) – Whether to remove the link from the old neighbors to this cuds object, defaults to True
- **_force** (*bool*) – Skip sanity checks.

Returns The created cuds object.

Return type *Cuds*

osp.core.utils.**delete_cuds_object_recursively**(*cuds_object*, *rel=<OntologyRelationship cuba.activeRelationship>*, *max_depth=inf*)

Delete a cuds object and all the object inside of the container of it.

Parameters

- **cuds_object** (Cuds) – The CUDS object to recursively delete.
- **rel** (OntologyRelationship, *optional*) – The relationship used for traversal. Defaults to cuba.activeRelationship.
- **max_depth** (*int, optional*) – The maximum depth of the recursion. Defaults to float("inf"). Defaults to float("inf").

osp.core.utils.**export_cuds**(*cuds_or_session: typing.Optional = None*, *file: typing.Optional[typing.Union[str, typing.TextIO]] = None*, *format: str = 'text/turtle'*, *rel: osp.core.ontology.relationship.OntologyRelationship = <OntologyRelationship cuba.activeRelationship>*, *max_depth: float = inf*) → Optional[str]

Exports CUDS in a variety of formats (see the *format* argument).

Parameters

- **cuds_or_session** (*Union[Cuds, Session], optional*) – the (Cuds) CUDS object to export, or (Session) a session to serialize all of its CUDS objects. If no item is specified, then the current session is exported.
- **file** (*str, optional*) – either, (str) a path, to save the CUDS objects or, (TextIO) any file-like object (in string mode) that provides a *write()* method. If this argument is not specified, a string with the results will be returned instead.
- **format** (*str*) – the target format. Defaults to triples in turtle syntax.
- **rel** (OntologyRelationship) – the ontology relationship to use as containment relationship when exporting CUDS.
- **max_depth** (*float*) – maximum depth to search for children CUDS.

osp.core.utils.**get_neighbor_diff**(*cuds1*, *cuds2*, *mode='all'*)

Get the ids of neighbors of cuds1 which are no neighbors in cuds2.

Furthermore get the relationship the neighbors are connected with. Optionally filter the considered relationships.

Args; cuds1 (Cuds): A Cuds object. cuds2 (Cuds): A Cuds object. mode (str): one of "all", "active", "non-active", whether to consider

only.

active or non-active relationships.

Returns

List of Tuples that contain the found uids and relationships.

Return type List[Tuple[Union[UUID, URIRef], Relationship]]

osp.core.utils.**get_relationships_between**(*subj*, *obj*)

Get the set of relationships between two cuds objects.

Parameters

- **subj** (Cuds) – The subject
- **obj** (Cuds) – The object

Returns

The set of relationships between subject and object.

Return type Set[*OntologyRelationship*]

osp.core.utils.**import_cuds**(*path_or_filelike: Union[str, TextIO, dict, List[dict]]*, *session: Optional = None*, *format: str = None*)

Imports CUDS in various formats (see the *format* argument).

Parameters

- **path_or_filelike** (*Union[str, TextIO], optional*) – either, (str) the path of a file to import; (Union[List[dict], dict]) a dictionary representing the contents of

  a json file;

  (TextIO) any file-like object (in string mode) that provides a *read()* method. Note that it is possible to get such an object from any *str* object using the python standard library. For example, given the *str* object *string*, *import io; filelike = io.StringIO(string)* would create such an object. If not format is specified, it will be guessed.

- **session** (Session) – the session in which the imported data will be stored.
- **format** (*str, optional*) – the format of the content to import. The supported formats are *json* and the ones supported by RDFLib. See *https://rdflib.readthedocs.io/en/latest/plugin_parsers.html*. If no format is specified, then it will be guessed. Note that in some specific cases, the guess may be wrong. In such cases, try again specifying the format.

Returns (List[Cuds]): a list of cuds objects.

osp.core.utils.**post**(*url*, *cuds_object*, *rel=<OntologyRelationship cuba.activeRelationship>*, *max_depth=inf*)

Will send the given CUDS object to the given URL.

Will also send the CUDS object in the container recursively.

Parameters

- **url** (*string*) – The URL to send the CUDS object to
- **cuds_object** (Cuds) – The CUDS to send
- **max_depth** (*int, optional*) – The maximum depth to send CUDS objects recursively. Defaults to float("inf").

Returns Server response

osp.core.utils.**pretty_print**(*cuds_object*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Print the given cuds_object in a human readable way.

The uuid, the type, the ancestors and the description and the contents is printed.

Parameters

- **cuds_object** (Cuds) – container to be printed.
- **file** (*TextIOWrapper*) – The file to print to.

osp.core.utils.**remove_cuds_object**(*cuds_object*)

Remove a cuds_object from the data structure.

Removes the relationships to all neighbors. To delete it from the registry you must call the sessions prune method afterwards.

Parameters **cuds_object** (Cuds) – The cuds_object to remove.

osp.core.utils.**sparql**(*query_string: str*, *session: Optional = None*)

Performs a SPARQL query on a session (if supported by the session).

Parameters

- **query_string** (*str*) – A string with the SPARQL query to perform.
- **session** (Session, *optional*) – The session on which the SPARQL query will be performed. If no session is specified, then the current default session is used. This means that, when no session is specified, inside session *with* statements, the query will be performed on the session associated with such statement, while outside, it will be performed on the OSP-core default session, the core session.

Returns

A SparqlResult object, which can be iterated to obtain the output rows. Then for each *row*, the value for each query variable can be retrieved as follows: *row['variable']*.

Return type SparqlResult

Raises **NotImplementedError** – when the session does not support SPARQL queries.

19.5.1 pico

Pico is a commandline tool used to install ontologies.

osp.core.pico.**install**(**files: str*, *overwrite: bool = False*) → None

Install ontologies.

Parameters

- **files** – Paths of *yml* files describing the ontologies to install.
- **overwrite** – Whether to overwrite already installed ontologies.

osp.core.pico.**namespaces**() → Iterator[*OntologyNamespace*]

Returns namespace objects for all the installed namespaces.

osp.core.pico.**packages**() → Iterator[str]

Returns the names of all installed packages.

osp.core.pico.**uninstall**(**package_names: str*) → None

Uninstall ontologies.

Parameters **package_names** – Names of the ontology packages to uninstall.

CHAPTER

TWENTY

CONTRIBUTE

This section aims to explain how we develop and organise, in order to help those that want to contribute to SimPhoNy.

20.1 Background

20.1.1 Tools

The following are some of the technologies and concepts we use regularly. It might be useful to become familiar with them:

- Version control: Git, GitHub and GitLab
- Unittest
- Continuous integration
- Python virtual environments/conda
- Docker
- Benchmarks

20.1.2 Code Organisation

There are 3 main categories of repos:

- *OSP-core* contains the nucleus of SimPhoNy, the base on which the wrappers build.
- Each *wrapper* will be in its own repository on GitHub or GitLab, mimicking wrapper_development.
- *docs* holds the source for this documentation.

There are also 4 types of branches:

- `master/main` contains all the releases, and should always be stable.
- `dev` holds the code for the newest release that is being developed.
- `issue branch` is where an specific issue is being solved.
- `hotfix branch` is where a critical software bug detected on the stable release (more on this later) is being solved.

All wrappers and OSP-core are part of a common directory structure:

- *osp/*: contains all the SimPhoNy source code.
  - *core/*: OSP-core source code.

  - *wrappers/*: wrappers source code.
    * *wrapper_xyz/*: one folder per wrapper implementation.
- *tests/*: unittests of the code.
- *examples/*: simple examples of how to use a certain feature.

20.2 Developing workflow

- Every new feature or bug is defined in an issue and labelled accordingly. If there is something that is missing or needs improving, make an issue in the appropriate project.
- Generally, the issues are fixed by creating a new `issue branch` from the `dev` branch, committing to that branch and making a new Pull/Merge Request when done. An owner of the project should be tagged for review. They will review and merge the PR if the fix is correct, deleting the `issue branch` afterwards. The changes should be clearly explained in the issue/Pull Request.

Warning: If the issue is a critical software bug detected in the stable release, a `hotfix branch` should be created from the `master/main` branch instead.

After committing to such branch, a new Pull/Merge request (targeting `dev`) should be created. If the fix is correct, the project owner will merge the PR to `dev`, additionally merge the `hotfix branch` to `master/main`, and then delete the `hotfix branch`.

- Once the features for a release have been reached, `dev` will be merged to `master/main`. Every new commit in the `master/main` branch generally corresponds to a new release, which is labelled with a git tag matching its version number. An exception to this rule may apply, for example when several critical hotfixes are applied in a row, as it would then be better to just to publish a single release afterwards. In regard to version numbering, we adhere to the *Semantic versioning* rules.

In the next image it can be seen how the branches usually look during this workflow, and the different commits used to synchronise them:

20.3 Coding

20.3.1 Documenting

- All code must be properly documented with meaningful comments.
- For readability, we now follow the Google docstring format.
- If some behaviour is very complex, in-line comments can be used. However, proper naming and clear operations are always preferred.

20.3.2 Code style

- Code should follow PEP8 code style conventions.
- All Python code should be validated by the Flake8 tool. The validation is also enforced on the repository by the *continuous integration*. Click here to see the specific options with which Flake8 is launched.
- All Python code should be reformatted with black and isort. The use of said tools is enforced by the *continuous integration*. Therefore, we strongly recommend that you use the configuration file bundled with the repository to install the pre-commit framework, that automates the task using git pre-commit hooks.
- A few other style conventions are also enforced by the continuous integration through pre-commit (such as empty lines at the end of text files). If you decide not to use it, the CI will let you know what to correct.

20.3.3 Testing

- All complex functionality must be tested.
- If some implementation can not be checked through unittest, it should be at least manually run in different systems to assure an expected behaviour.

20.3.4 Continuous Integration

- We currently run the CI through Github Actions/GitLab CI.
- Code style conventions are enforced through the use of Flake8, black, isort, and various pre-commit hooks.
- Tests are automatically run for all pull requests.
- For the OSP-core code, benchmarks are run after every merge to `dev`. Benchmark results are available here. The CI will report a failure when a benchmark is 50% slower than in the previous run, in addition to automatically commenting on the commit.

20.3.5 Naming conventions

- Use `cuds_object` as the argument name of your methods (not `entity`, `cuds`, `cuds_instance`...).
- The official spelling is `OSP-core` (as opposed to *osp core*, *OSP-Core* or similar).

20.4 Contribute to OSP-core

If you are not a member of the SimPhoNy organisation, rather than creating a branch from `dev`, you will have to fork the repository and create the Pull Request.

20.5 Contribute to wrapper development

For a sample wrapper, visit the wrapper_development repo.

README files should include:

- Information regarding the purpose of the wrapper and the backend used.
- A compatibility matrix with OSP-core.
- Installation instructions.
- Folder structure.
- Any other necessary information for users and other developers.

20.6 Contribute to the docs

If you have any suggestion for this documentation, whether it is something that needs more explanation, is inaccurate or simply a note on anything that could be improved, you can open an issue here, and we will look into it!.

CHAPTER

TWENTYONE

DETAILED DESIGN

Here we will give an in-depth view of the design of the 3 layers.

For a more general overview, go to *general architecture*.

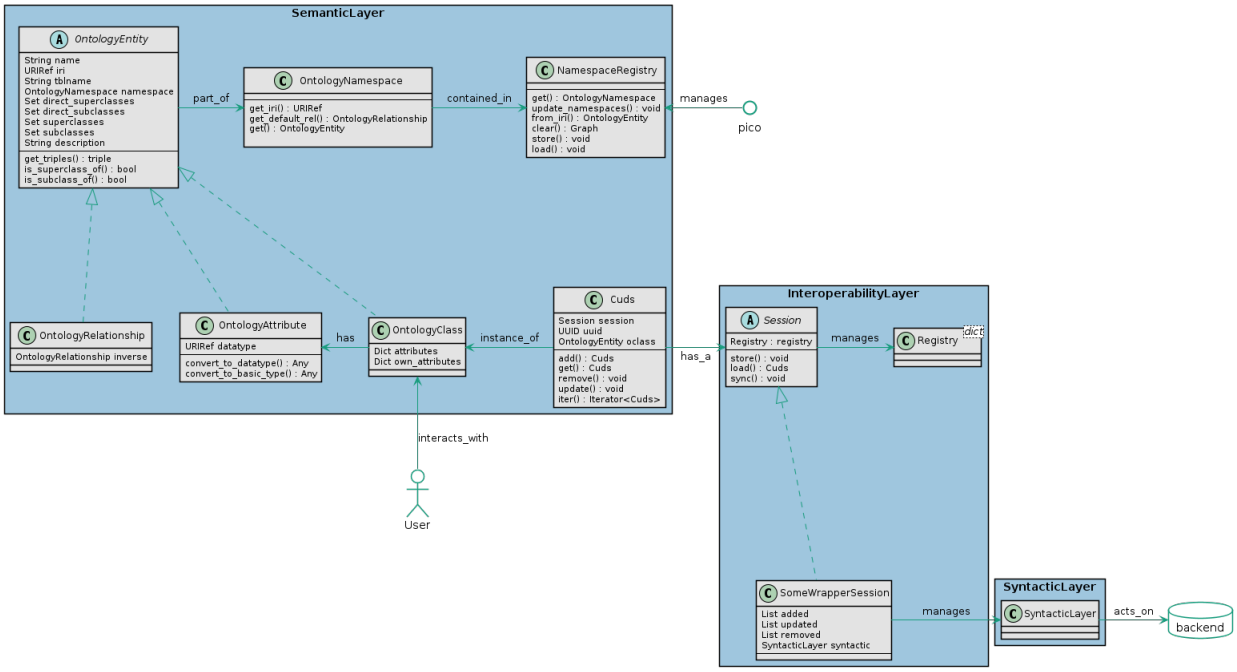


Fig. 1: Standard design

21.1 Semantic layer

The semantic layer is the representation of the classes of the ontology in a programming language.

When the user installs an ontology through `pico`, all ontology concepts are saved in a graph in `~/.osp_ontologies`.

The procedure is as follows:

- The `OntologyInstallationManager` receives a list of yml files with ontologies to install.
- It instantiates a `Parser`.
- The parser goes through the ontologies and creates an `OntologyClass` per entity.

- All the oclasses of the same namespace are grouped in an `OntologyNamespace`.
- All the registries are collected in the `NamespaceRegistry`.

Installing new ontologies loads the graph and adds new namespaces or modifies the existing ones.

When a class is instantiated, an individual is created. The graph is read, and an instance of the *Cuds* class with the ontology information is created.

Through the Cuds they realise the *Cuds API* which enables the user to work with them in a generic, simple way.

21.1.1 Cuds

Location: `osp.core.cuds`

It is the base class for all instances. Besides whatever might have been defined in the ontology, they all have 3 basic attributes:

- uid: instance of `uuid.UUID`, it serves to uniquely identify an instance.
- session: this is the link to the interoperability layer. By default all objects are in the `CoreSession`, unless they are in a wrapper.
- oclass: indicates the ontology class they originate from.

Cuds structure

Each cuds object contains the uids and oclass of the directly related entities, as well as the relationship that connects them. The actual related objects are kept in the *registry*.

```
a_cuds_object :=  {
   Relation1: {uid1: oclass, uid2: oclass},
   Relation2: {uid4: oclass},
   Relation3: {uid3: oclass, uid5: oclass},
   }
```

Note: This is an abstraction to show the general structure. The actual implementation is a bit more complex.

Cuds API

The governing idea behind the API design was to simplify as much as possible the usage.

This CRUD API is defined by 6 methods:

Create

```
from osp.core.namespaces import some_namespace

ontology_class = some_namespace.OntologyClass
relationship = some_namespace.relationship
cuds_obj = some_namespace.OntologyClass()
```

Add

```
# These will also add the opposed relationship to the new contained cuds object
cuds_obj.add(*other_cuds, rel=relationship)
cuds_obj.add(yet_another_cuds)  # Uses default relationship from ontology
```

The flow of information for the call of the `add` method would be:

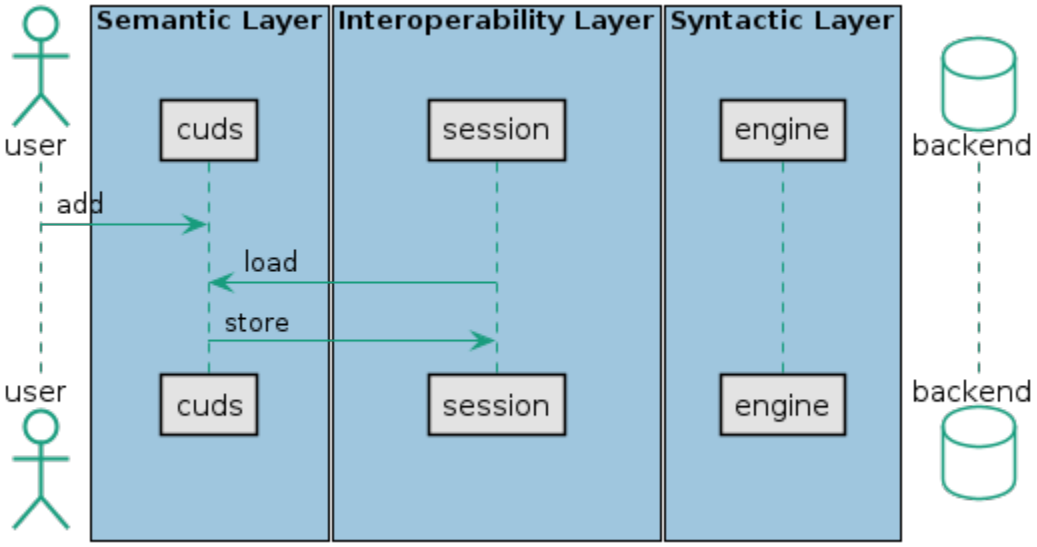


Fig. 2: add method call

As you can see, the information is sent to the next layer, but not all the way to the backend. This will be propagated when the user calls `session.run()` or `session.commit`. The registry is checked for a pre-existing object, in case something that is already there is being added.

Get

```
# Returns a list, unless only one uid was given
cuds_obj.get()                                  # All the contained cuds objects
cuds_obj.get(rel=relationship)                  # Entities under that␣
↪relationship
cuds_obj.get(*uids)                             # Searches elements for the uids
cuds_obj.get(*uids, rel=relationship)           # Faster, filters by␣
↪relationship
```

(continues on next page)

(continued from previous page)

```
cuds_obj.get(oclass=ontology_class)                    # Elements of that class
cuds_obj.get(rel=relationship, oclass=ontology_class)  # Filters by rel and oclass
```

In this case, the calls carried out by the `get` method are as follows:

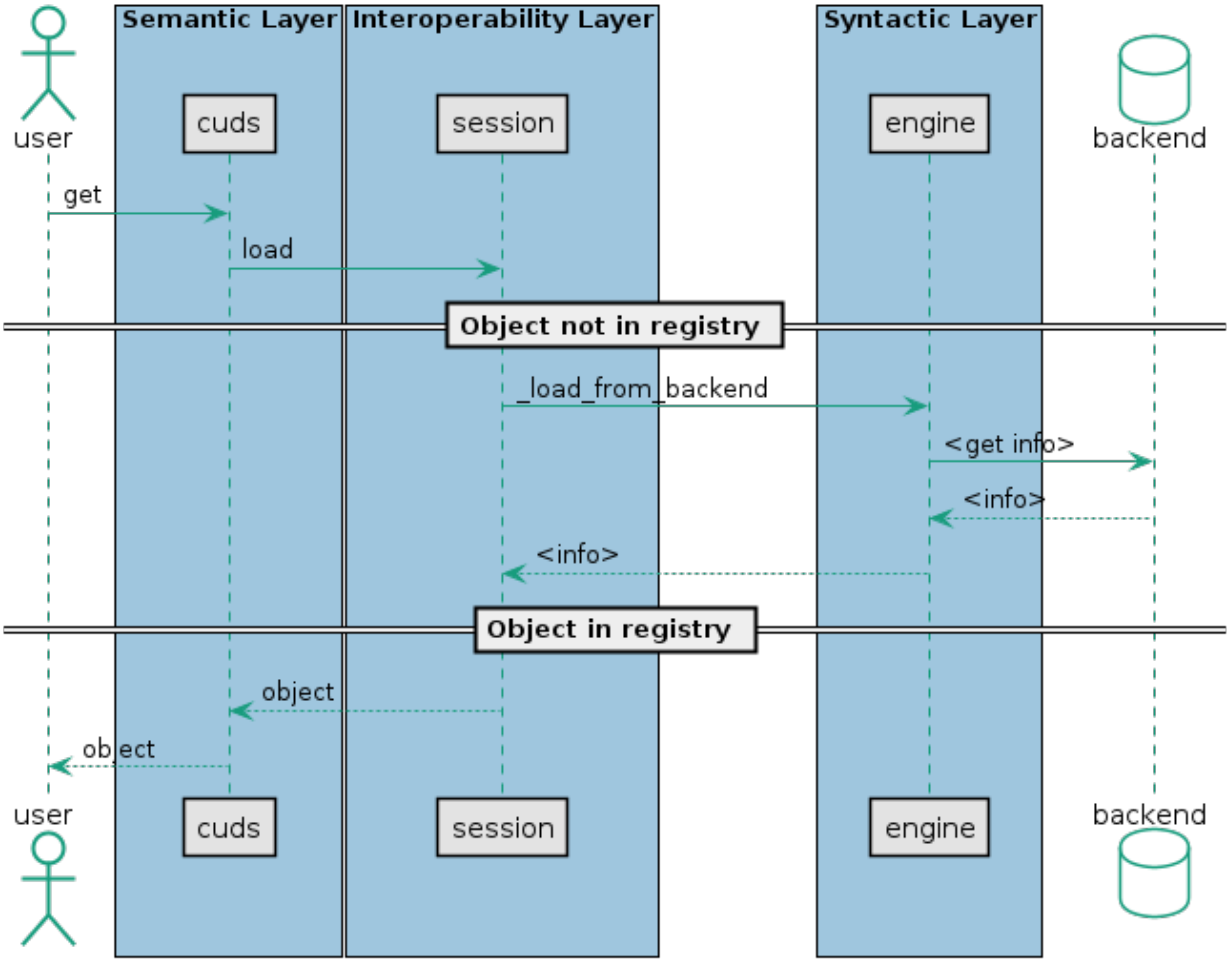


Fig. 3: get method call

Now the backend is contacted to make sure the user receives the latest available version of the objects being queried. This is done through `_load_from_backend()`.

Update

```
# Objects to update must exist already
cuds_obj.update(*cuds_objs)
```

A simple `update` call triggers the following behaviour:

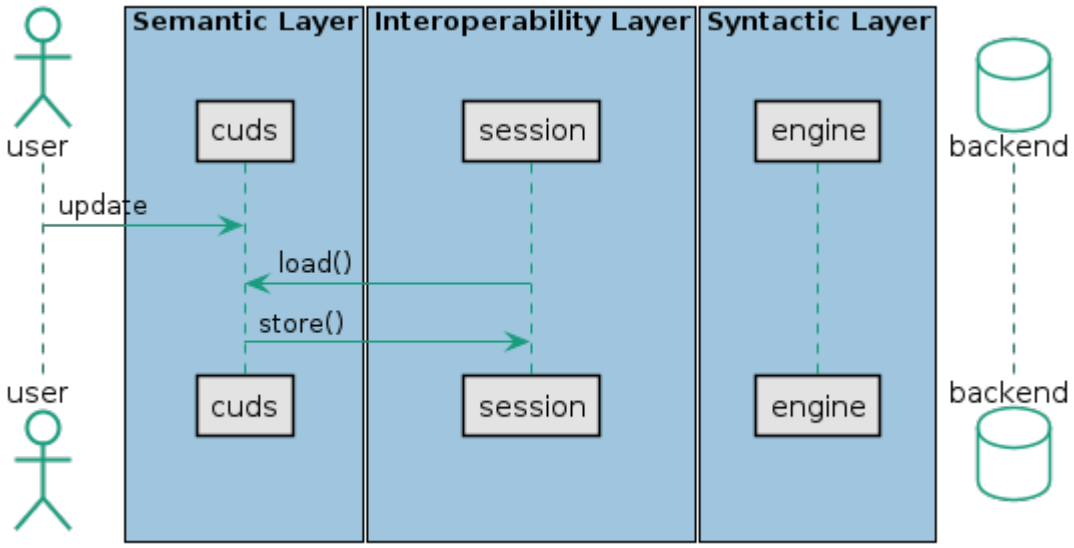


Fig. 4: update method call

You can see the calls are very much the same as with the `add` method. The difference is that the `update` requires the object to be there previously. And so the object is first loaded from the registry, then updated and stored.

Remove

```
# These will trigger the update in the opposed relationship of the erased element
cuds_obj.remove()                                          # Remove all
cuds_obj.remove(*uids/cuds_objs)                           # Remove objects with the given␣
↪uids
cuds_obj.remove(*uids/cuds_objs, rel=relationship)         # Faster, filters by␣
↪relationship
cuds_obj.remove(rel=relationship)                          # Delete all elements under a␣
↪relationship
cuds_obj.remove(oclass=ontology_class)                     # Delete all elements of a␣
↪certain class
cuds_obj.remove(rel=relationship, oclass=ontology_class) # Delete filtering by rel and␣
↪oclass
```

The sequence for a simple `remove` is:

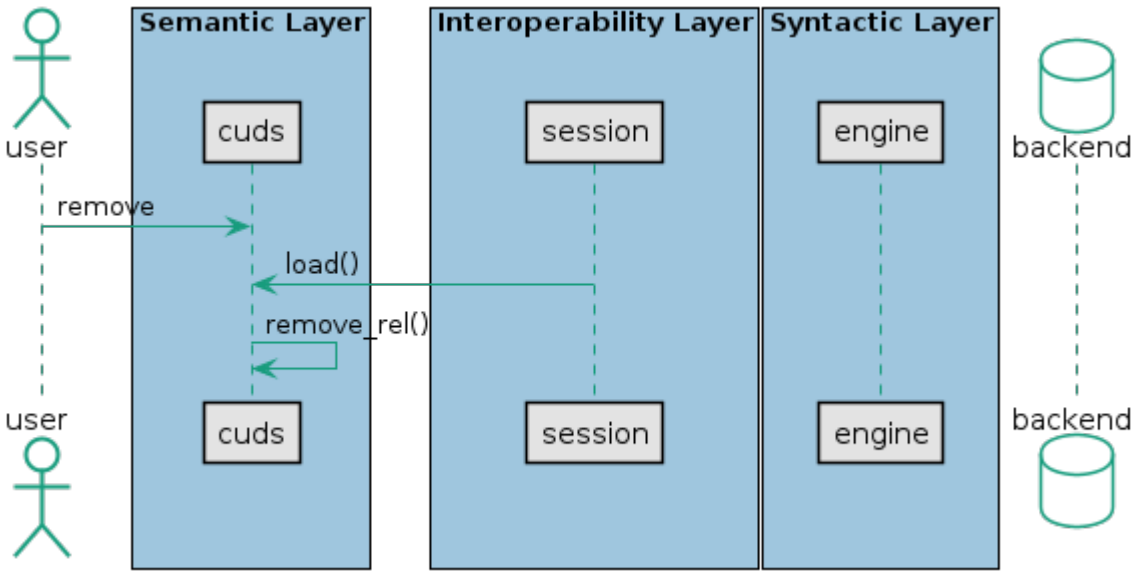


Fig. 5: remove method call

Here the registry is accessed to fetch the neighbours of the removed object and delete their links (relationships) to it.

Iterate

```
cuds_obj.iter()                                    # Iterates through all
cuds_obj.iter(oclass=ontology_class)               # Iterates filtering by the␣
↪ontology class
cuds_obj.iter(rel=relationship)                    # Iterates filtering by the␣
↪relationship
```

The general behaviour of the `iter` is:

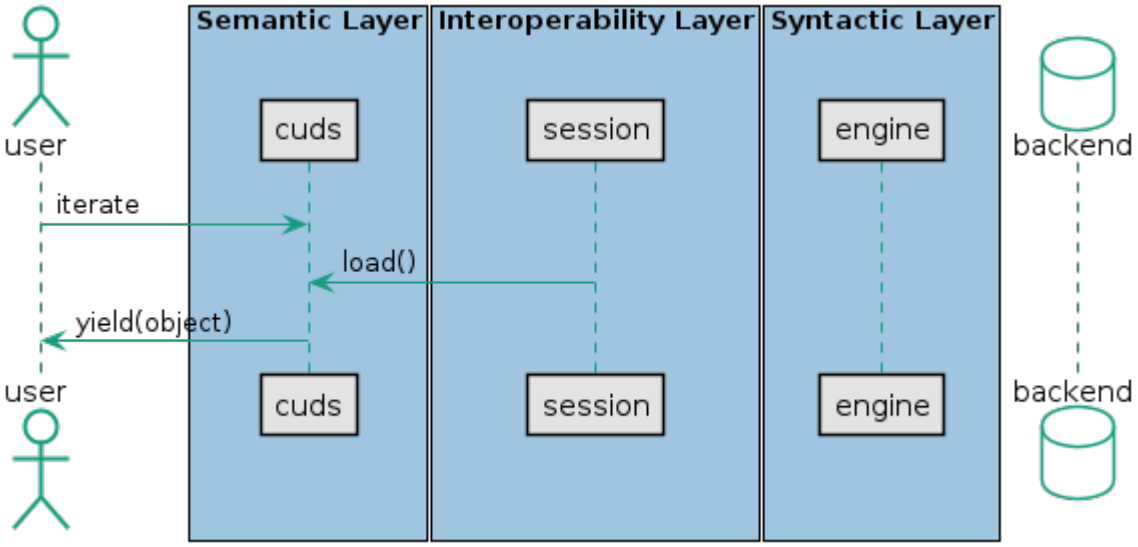


Fig. 6: iter method call

First the uids of all the objects to be iterated are gathered, and then they are yielded like a generator

Hint: There is also an `is_a` method for checking oclass inheritance.

Note: Be aware that the sequence diagrams shown represent simple use cases, and more complex scenarios are also possible (e.g. adding an object with children).

21.2 Interoperability layer

The interoperability layer takes care of the connection and translation between the semantic and syntactic parts. It also contains the storage of all the objects that share a session.

21.2.1 Registry

Location: `osp.core.session.registry`

This flat datastructure stores all the objects in the same workspace (session) by their uid. It is accessed through the session, and invisible to the user.

It also has functionality for pruning, resetting, or filtering its elements.

21.2.2 Session

Location: `osp.core.session`

The main purpose of session objects is to propagate the changes introduced by the user (and stored in the registry) to the backend, and update the registry with the modifications coming from the backend.

The backend is accessed via the Syntactic layer, through the `_engine` property.

To simplify and group functionality, we built an inheritance scheme:

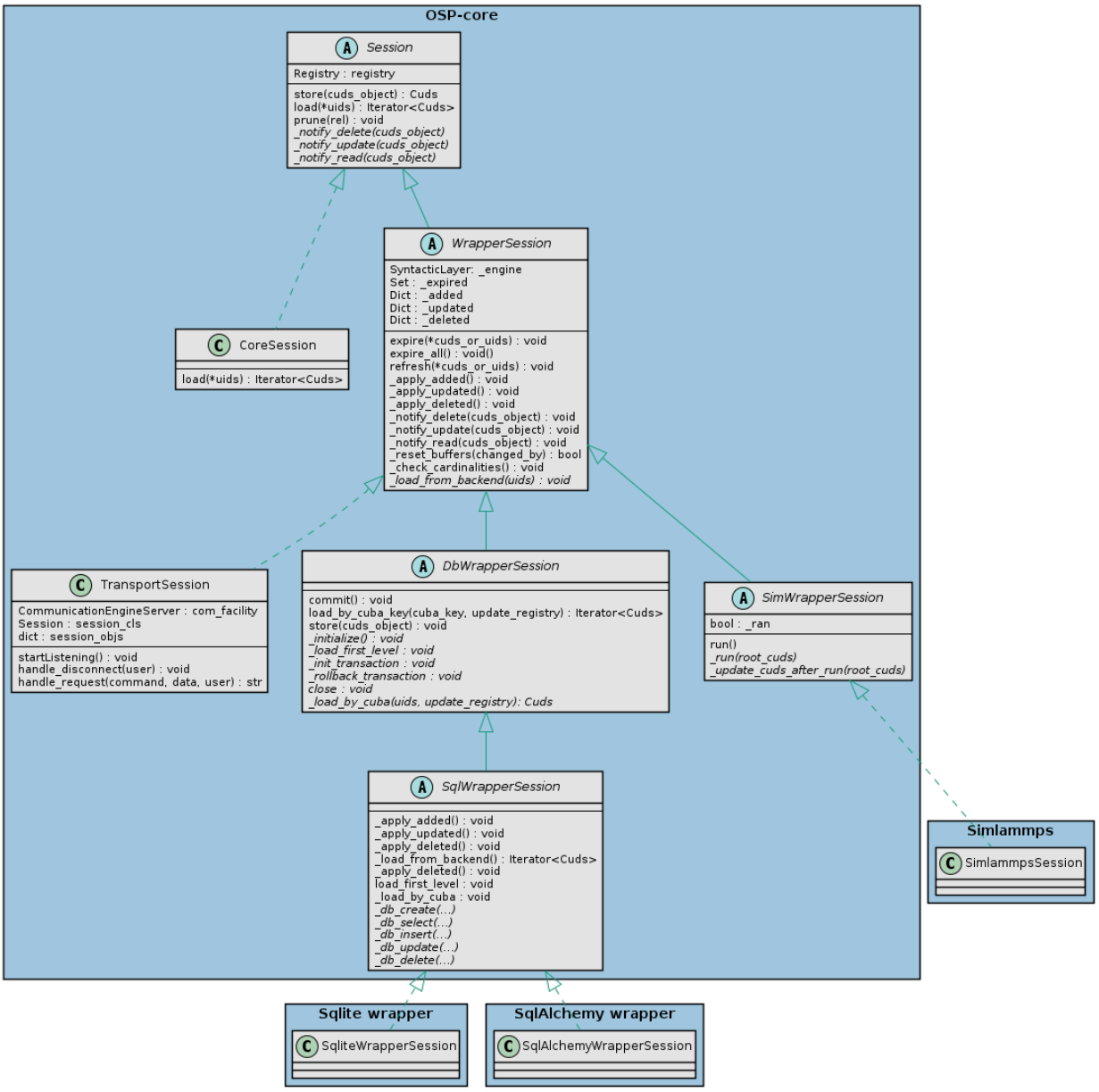


Fig. 7: Session inheritance scheme

Note: This is a reduced version and does not represent the entirety of the contained functions.

The simplest session, called `CoreSession`, is the default one for entities created in a python workspace and has no backend. It just accesses the registry to manage the operations made by users.

All wrappers will share `WrapperSession` as an ancestor. This will define which methods have to be implemented and `_engine` as the access point to a backend.

`SimWrapperSession` and `DbWrapperSession` further specify the behaviour of wrappers, defining the methods that trigger an action on the backend (`run` and `commit`, respectively).

Note: You might have noticed that the semantic layer defines `remove` in the API, but in the session and registry we use `delete`. The different between them is conceptual: `remove` is interpreted as detachment i.e. removal of edges, while `delete` implies the erasure of the node itself.

Buffers

Session classes under `WrapperSession` share 3 types of buffers, namely `added`, `updated` and `deleted`. The previous buffers are repeated twice, first for the user and then for the engine, so the number of buffers is actually 6.

As we have seen in the previous section, not all API calls trigger a change all the way to the backend. In fact, most of them do not. This is done to limit the traffic in the slower sections (networking or communicating with the engine).

On the other hand, the user should be able to access the latest version of the data (meaning the changes they might have just done), and the wrapper should know what changes have taken place since the last sync with the backend software (`commit` or `run`). In order to achieve these, the changes done by the user directly modify the semantic layer and are flagged in the buffers as changes to be propagated

Users or wrapper developers do not have to worry about updating this buffers, OSP-core handles them (both filling them up and emptying them).

However, these structures will be used in the different `_apply_<buffer>` methods when developing a wrapper (see *this* section of wrapper development).

Load from Backend

Similar to how the `_apply_<buffer>` methods are used to send information to the engine, `_load_from_backend` has the purpose of updating the semantic layer with the latest information from the backend.

You can see in the *`get` sequence diagram* that when the information has potentially changed in the backend (i.e the simulation has run, or a database has more data) the `get` has to fetch the latest version. To achieve this, OSP-core calls `_load_from_backend` with the list of desired uids, and the wrapper wrapper will update the objects in the registry with the relevant information and `yield` them.

21.2.3 Networking

Location: `osp.core.session.transport`

You may have noticed in the *session inheritance scheme* that there is `TransportSession` implementing the `WrapperSession`. This session class is the way to connect to engines that are located in other machines through web sockets.

The behaviour is as follows:

- The user instantiates a `TransportSessionClient` and provides the session class of the remote server, the hostname and the port.
- The `TransportSessionClient` will connect to a `TransportSessionServer` through a `CommunicationEngineClient`.
- The server has the wrapper package installed locally.
- `CommunicationEngineClient` and `CommunicationEngineServer` (one on each side) take care of the communication, so that:
  - The methods that the user would call on the remote wrapper are encoded with the relevant data (in json) and sent to the server.
  - The server deserialises the data and calls the method on the wrapper.
  - The results are serialised and sent back to the user´s local transport session.

The chosen implementation hides most of the work from the users and wrapper developers. The only difference between a local wrapper and a remote one is the line where the wrapper session is instantiated, from:

```
sess = SomeWrapperSession(parameter_a, parameter_b)
wrapper = AWrapperInstance(session=sess)
```

to:

```
# Once the server is properly setup
sess = TransportSessionClient(SomeWrapperSession, host, port,
                              parameter_a, parameter_b)
wrapper = AWrapperInstance(session=sess)
```

21.3 Syntactic layer

This layer is in direct communication with the backend. It has no ontological knowledge and must just provide a simple interface for the interoperability layer to interact with the wrapped application.

This means it may have to be a binding if the application is in a different language. It could also be a file generator/parser for backends that only allow file i/o. In other cases, (e.g. LAMMPS with PyLammps) it is provided by the backend itself, and requires no implementation.

Since the syntactic layer will greatly depend on the specific backend, no standardisation is provided there.

CHAPTER

TWENTYTWO

RELATED LINKS

Here are links to other projects relevant for this documentation.

SimPhoNy:

- GitLab's SimPhoNy group
- GitHub's SimPhoNy group
- OSP-core
- wrappers
- wrapper development

Technologies used:

- Docker, used for the CI and engines
- Sphinx, used for the documentation
- PlantUML, used for the diagrams

CHAPTER

TWENTYTHREE

ACKNOWLEDGEMENTS

SimPhoNy OSP-core and wrappers development is supported by the following Grants:

Project	Programme	Call ID	Grant Agreement ID
SimPhoNy	FP7	NMP-2013-1.4-1	604005
MarketPlace	Horizon 2020	H2020-NMBP-TO-IND-2016-2017	760173
FORCE	Horizon 2020	H2020-NMBP-TO-IND-2016-2017	721027
SimDOME	Horizon 2020	H2020-NMBP-TO-IND-2018-2020	814492
OYSTER	Horizon 2020	H2020-NMBP-2017-two-stage	760827
INTERSECT	Horizon 2020	H2020-NMBP-TO-IND-2018-2020	814487
ReaxPRO	Horizon 2020	H2020-NMBP-TO-IND-2018-2020	814416
APACHE	Horizon 2020	H2020-NMBP-ST-IND-2018	814496
NanoMECommons	Horizon 2020	H2020-NMBP-TO-IND-2020-twostage	952869
OntoTRANS	Horizon 2020	H2020-NMBP-TO-IND-2019	862136

Some of the explanations and background provided have been adapted from Pablo de Andres' master thesis on "Natural Language Search on an ontology-based data structure".

The OSP-core Python package originates from the European Project SimPhoNy (Project Nr. 604005). We would like to acknowledge and thank our project partners, especially Enthought, Inc, Centre Internacional de Mètodes Numèrics a l'Enginyeria (CIMNE) and the University of Jyväskylä, for their important contributions to some of the core concepts of OSP-core, which were originally demonstrated under the project https://github.com/simphony/simphony-common.

CHAPTER

TWENTYFOUR

COMPATIBILITY TABLE

The following table describes the compatibilities between the SimPhoNy docs and OSP-core up to version 2.5.1 of the documentation. For later releases, the version number of the documentation matches the version number of the OSP-core release to which it applies.

SimPhoNy docs	OSP-core
2.5.1	3.8.0
2.5.0	3.8.0
2.4.5	3.7.0
2.4.4	3.5.8-beta
2.4.3	3.5.5-beta
2.4.2	3.5.4-beta
2.4.1	3.5.3.1-beta
2.4.0	3.5.2-beta
2.3.x	3.4.0-beta
2.2.x	3.3.5-beta
2.1.x	3.3.0-beta

CHAPTER

TWENTYFIVE

LICENSE

CHAPTER

TWENTYSIX

CONTACT

If you see something wrong, missing, or in need of clarification, you can directly create an issue in here.

Any other questions, issues or comments can be directed to Pablo de Andrés, José Manuel Domínguez and Yoav Nahshon from the Materials Data Science and Informatics Team, Fraunhofer IWM.

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W