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 (such as simulation engines, databases and data repositories) using a knowledge graph as the common language. It is focused on the domain of 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 special "boxes" where linked data can be stored 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 a mere container for ontology entities. 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

INSTALLATION

For the installation and usage of SimPhoNy, Python 3.7 or higher is required. SimPhoNy is available on PyPI, so it can be installed using the `pip` package manager.

```
pip install simphony-osp
```

2.1 Wrapper installation

SimPhoNy Wrappers are distributed as separate Python packages. Typically, installing them also involves just installing a package either from the Python Package Index or a different source. However, sometimes it can be more complicated than that. Head to each wrapper's website for specific installation instructions.

2.2 Developers

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

```
git clone https://github.com/simphony/simphony-osp.git
pip install ./simphony-osp
```

Note that it is also possible to install Python packages in *development mode*, meaning that their source code can be edited in-place without needing to reinstall them to see the changes.

```
# development mode installation
pip install -e ./simphony-osp
```

CHAPTER

THREE

QUICKSTART

Tip

Using the button above, you can launch a Jupyter notebook to follow this tutorial without even having to install SimPhoNy.

This tutorial offers a quick first-contact with SimPhoNy. The learning objectives are:

- Convey the purpose of SimPhoNy
- Manage the installed ontologies
- Use the installed ontologies to instantiate ontology individuals
- Demonstrate the usage of wrappers to achieve interoperability

Following the tutorial on your own machine requires the installation of SimPhoNy and the SimLAMMPS wrapper. We recommend that you use the button above to follow the tutorial online using Binder.

3.1 Ontologies

SimPhoNy enables you to manage data that is based on ontologies. This means that all information is represented in terms of *ontology individuals*. Individuals belong to a specific *ontology class*, have specific *attributes* and can be connected to other individuals through *relationships*. Classes, attributes and relationships are defined in ontologies. Therefore, in order for SimPhoNy to be able to properly interpret the data, such ontologies need to be installed. For that purpose, SimPhoNy includes an ontology management tool called *pico*.

In this tutorial, you will work, among others, with the SimLAMMPS wrapper. This wrapper only understands data based on the SimLAMMPS ontology, which is included with it. Therefore, you will start by installing such ontology.

pico works with so-called "ontology packages". Ontology packages are just a pointer to an ontology file with some additional metadata defined on it, such as the namespaces that the ontology includes or a name for the package. You can learn to create your own packages here.

To install the desired ontology use the command `pico install` and provide the path to the ontology package.

[1]:
```
# if you are running the tutorial online using Binder, then the simlammps
# ontology is already pre-installed

# otherwise, download `simlammps.ttl` and `simlammps.yml` from
# https://github.com/simphony/simlammps/tree/v4.0.0/simphony_osp_simlammps
# and run

#!pico install simlammps.yml
```

pico will install the ontology. After the installation is complete, it is listed among the installed ontology packages when running the `pico list` command.

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

```
Packages:
        - simlammps
Namespaces:
        - simphony
        - owl
        - rdfs
        - simlammps
```

That's all! Everything is ready to start using SimPhoNy.

3.2 Data and sessions

The simplest way to start working with some data is the following.

1. Import an installed *ontology namespace*. Namespaces agglomerate the ontology classes, relationships, and attributes included in an ontology. Namespaces come bundled with ontology packages, so that one ontology package can provide several namespaces.
2. Retrieve a class from the namespace and use it to create an ontology individual.
3. Assign attributes or connect the individual with others.

In the example below, first the `simlammps` namespace from the previously installed ontology is imported, then two ontology individuals of classes *Atom* and *Position* are created. After that, the individual of class Atom is linked to the individual of class Position through the relationship *hasPart*, and finally some coordinates are assigned to the individual of class Position.

[3]:
```
from simphony_osp.namespaces import simlammps

atom, position = simlammps.Atom(), simlammps.Position(vector=[1, 0, 3])
atom[simlammps.hasPart] = position

atom.label = "My Atom"
position.label = "My Atom's position"
```

SimPhoNy includes a visualization tool that can draw a graph containing any ontology individuals you desire, their attributes, and the relationships connecting them. Using it, the dataset that has just been created may be visualized.

[4]:
```
from simphony_osp.tools import semantic2dot

semantic2dot(atom, position)
```

[4]:

But... Where are actually the atom and the position stored? If a new atom is created running `atom = simlammps.Atom()` again, what happens to the old atom, how can it be retrieved?

SimPhoNy stores data in the so-called *sessions*. You may think of a session as a "box" were ontology idividuals can be placed. The magic lies within the fact that sessions can provide *views* into different data sources and software products, thanks to the SimPhoNy Wrapper mechanism. This means that you see a "box" containing ontology entitites, but behind the scenes, SimPhoNy is translating this information so that it can be used by the underlying data source or software.

But still, this does not clarify the issue. To which session did the atom and the position go? When you do not specify any session, objects are created by default on the so-called *Core Session*, which is the default session of SimPhoNy. You may access the default session at any time by importing it.

[5]:
```
from simphony_osp.session import core_session

list(core_session)
```

[5]:
```
[<OntologyIndividual: My Atom 63f477ea-dfc5-48c7-8227-60b71cc65cb6>,
 <OntologyIndividual: My Atom's position b9074f9e-6bd4-4eef-b8ae-78fd4d1d86cb>]
```

Now it is clear that the atom and its position are stored on the Core Session. Be aware that the Core Session is only meant to serve as a way to quickly test SimPhoNy or be a transient place to store information, as all of its contents are **lost** when you close the Python shell.

3.3 Wrappers

Wrappers are sessions that are connected to data sources or other software. When you use then, you see a "box" filled with ontology entitites, but behind the scenes, they do the necessary computations to store your data on said data sources or transfer it to the software.

In this example, the SQLite (included with SimPhoNy) and SimLAMMPS wrappers are used.

First, start by creating an SQLite session. In this session, create three atoms, with their respective positions and velocities, and then commit the changes.

[6]:
```
from simphony_osp.wrappers import SQLite

with SQLite('atoms.db', create=True) as session:

    atom = simlammps.Atom(); atom.label = 'Atom 0'
    position = simlammps.Position(vector=[1, 0, 0]); position.label = 'Position 0'
    velocity = simlammps.Velocity(vector=[0, 1, 0]); velocity.label = 'Velocity 0'
    atom[simlammps.hasPart] = {position, velocity}

    atom = simlammps.Atom(); atom.label = 'Atom 1'
    position = simlammps.Position(vector=[1, 0, 1]); position.label = 'Position 1'
    velocity = simlammps.Velocity(vector=[3, 1, 0]); velocity.label = 'Velocity 1'
    atom[simlammps.hasPart] = {position, velocity}

    atom = simlammps.Atom(); atom.label = 'Atom 2'
    position = simlammps.Position(vector=[1, 1, 0]); position.label = 'Position 2'
    velocity = simlammps.Velocity(vector=[0, 1, 2]); velocity.label = 'Velocity 2'
    atom[simlammps.hasPart] = {position, velocity}

    session.commit()
```

When the session is opened again, the atoms are still there!

[7]:
```
with SQLite('atoms.db', create=False) as session:
    print(list(session))
```

```
[<OntologyIndividual: Atom 0 330dafe8-0465-440f-b36c-3a8327a06b25>, <OntologyIndividual:␣
↪Position 0 bdae4880-670e-4c0d-8ade-8a2ff840fa60>, <OntologyIndividual: Velocity 0␣
↪2a630f63-e786-49e1-9ddf-125de1ef914b>, <OntologyIndividual: Atom 1 bceb1b3a-5c10-4ba5-
↪aa0a-10499349527e>, <OntologyIndividual: Position 1 b9e812c7-8fb8-4d45-a064-
↪fce80a6ac986>, <OntologyIndividual: Velocity 1 2cd6be8a-0b03-48f2-bb1f-3a019b665ce6>,
↪<OntologyIndividual: Atom 2 988e2652-4c22-41fd-8471-3896553b4180>, <OntologyIndividual:
↪ Position 2 962aa585-4121-48b1-ba47-26f78092469c>, <OntologyIndividual: Velocity 2␣
↪536a16e3-4240-4414-8be2-a3a18f446e66>]
```

(continues on next page)

(continued from previous page)

The next step is to copy these atoms to a SimLAMMPS session and run a simulation.

[8]:
```
from simphony_osp.wrappers import SimLAMMPS

# open the SQLite database and create a new SimLAMMPS session
sqlite_session = SQLite('atoms.db', create=False)
simlammps_session = SimLAMMPS()

# prevent closing the sessions when leaving the contexts after
# using the `with` statement
sqlite_session.locked = True
simlammps_session.locked = True

# copy the individuals from the SQLite session to the SimLAMMPS session
ontology_individuals_from_database = list(sqlite_session)
simlammps_session.add(ontology_individuals_from_database);
```

```
LAMMPS output is captured by PyLammps wrapper
LAMMPS (23 Jun 2022 - Update 1)
OMP_NUM_THREADS environment is not set. Defaulting to 1 thread. (src/comm.cpp:98)
  using 1 OpenMP thread(s) per MPI task
```

```
--------------------------------------------------------------------------
The library attempted to open the following supporting CUDA libraries,
but each of them failed.  CUDA-aware support is disabled.
libcuda.so.1: cannot open shared object file: No such file or directory
libcuda.dylib: cannot open shared object file: No such file or directory
/usr/lib64/libcuda.so.1: cannot open shared object file: No such file or directory
/usr/lib64/libcuda.dylib: cannot open shared object file: No such file or directory
If you are not interested in CUDA-aware support, then run with
--mca opal_warn_on_missing_libcuda 0 to suppress this message.  If you are interested
in CUDA-aware support, then try setting LD_LIBRARY_PATH to the location
of libcuda.so.1 to get passed this issue.
--------------------------------------------------------------------------
```

[9]:
```
with simlammps_session:
    # a few additional entities that were omitted before for brevity
    # are needed to configure the simulation and are created now

    # material
    mass = simlammps.Mass(value=0.2)
    material = simlammps.Material()
    material[simlammps.hasPart] += mass
    for atom in simlammps_session.get(oclass=simlammps.Atom):
        atom[simlammps.hasPart] += material

    # simulation box
    box = simlammps.SimulationBox()
    face_x = simlammps.FaceX(vector=(10, 0, 0))
    face_x[simlammps.hasPart] += simlammps.Periodic()
    face_y = simlammps.FaceY(vector=(0, 10, 0))
```

(continues on next page)

(continued from previous page)

```
    face_y[simlammps.hasPart] += simlammps.Periodic()
    face_z = simlammps.FaceZ(vector=(0, 0, 10))
    face_z[simlammps.hasPart] += simlammps.Periodic()
    box[simlammps.hasPart] += {face_x, face_y, face_z}

    # molecular dynamics model
    md_nve = simlammps.MolecularDynamics()

    # solver component
    sp = simlammps.SolverParameter()

    # integration time
    steps = 100
    itime = simlammps.IntegrationTime(steps=steps)
    sp[simlammps.hasPart] += itime

    # verlet
    verlet = simlammps.Verlet()
    sp[simlammps.hasPart] += verlet

    # define the interatomic force as material relation
    lj = simlammps.LennardJones612(
        cutoffDistance=2.5, energyWellDepth=1.0, vanDerWaalsRadius=1.0
    )
    lj[simlammps.hasPart] += material
```

[10]:
```
# the simulation is now ready to be run
simlammps_session.compute()
```

After running the simulation, the data in the SQLite database can be overwritten with the new data.

[11]:
```
# finally, the data in the sqlite database can be overwritten
# with the new data
sqlite_session.add(
    simlammps_session,
    exists_ok = True,
    merge = False,  # overwrite an entity if it already exists
)
sqlite_session.commit()

# and both sessions can be closed
sqlite_session.close(), simlammps_session.close();
```

As expected, if the saved atoms are now examined, their positions and velicities have changed.

[12]:
```
with SQLite('atoms.db', create=False) as session:
    for i, atom in enumerate(session.get(oclass=simlammps.Atom)):
        velocity = atom.get(oclass=simlammps.Velocity).one()
        position = atom.get(oclass=simlammps.Position).one()
        print(
            f"{atom.label}:\n"
            f"  - Position {position.vector}\n"
```

(continues on next page)

(continued from previous page)

```
            f"  - Velocity {velocity.vector}"
        )
```

```
Atom 0:
  - Position [1.09604332 9.98007699 9.56039398]
  - Velocity [ 0.36107337  0.40331728 -0.30807669]
Atom 1:
  - Position [2.23711536 0.99949841 1.09814967]
  - Velocity [ 2.68929927  1.9921991  -0.5872734 ]
Atom 2:
  - Position [1.16684132 1.5204246  1.34145635]
  - Velocity [-0.05037264  0.60448362  2.89535009]
```

That's all! This was just a quick overview of the usage and purpose of SimPhoNy. Keep reading the documentation to learn more.

CHAPTER

FOUR

ONTOLOGY MANAGEMENT

4.1 Installing ontologies (pico)

SimPhoNy works with data that is based on ontologies. In particular, all information is represented in terms of ontology individuals that belong to specific ontology classes, have specific attributes and can be connected to other individuals through relationships. Classes, attributes and relationships are defined in the ontologies. Therefore, in order for SimPhoNy to be able to properly interpret the data, such ontologies need to be made available to it. For that purpose, SimPhoNy includes an ontology management tool called `pico`.

Ontologies can be added to SimPhoNy by installing ontology packages, which are YAML configuration files that, in addition to pointing to the actual ontology files, also define extra metadata. A list of supported ontology languages is available here.

We *bundle a few of these files with SimPhoNy* to enable rapid installation of common, well-known ontologies. If the ontology you wish to install is not among these, and the ontology author does not provide such a file, then you may just simply *create one yourself*.

There are three 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*.

4.1.1 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 install

Usage:

- `pico install <path/to/ontology_package.yml>`
- `pico install <path/to/ontology_package1.yml> <path/to/ontology_package2.yml> ...`
- `pico install city foaf emmo dcat` (ontology packages that are *bundled with SimPhoNy* can be installed using 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 [simphony_osp.utils.pico]: Will install the following packages: city.
INFO [simphony_osp.utils.pico]: Will install the following namespaces: city.
INFO [simphony_osp.utils.pico]: Installation successful
```

pico list

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 uninstall

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`.
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 simphony_osp.tools.pico <command>
```

For example:

```
python -m simphony_osp.tools.pico install city
```

4.1.2 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 `simphony_osp.tools.pico` module,

```
from simphony_osp.tools.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_package.yml')`, `install('foaf', 'dcat2')`
- `uninstall('city', 'foaf')`
- `print(list(packages()))`
- `print(list(namespaces()))`

4.1.3 Ontology installation folder

The installed ontologies are stored in the directory `~/.simphony-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 `SIMPHONY_ONTOLOGIES_DIR`.

4.2 Included ontologies

To use an ontology, you first have to add it to SimPhoNy by *installing* an *ontology package*. Ontology packages are YAML configuration files that, in addition to pointing to the actual ontology file, also define extra metadata.

We bundle a few of these files with SimPhoNy to enable rapid installation of common, well-known ontologies. 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 (DCAT)* - Data catalogue information
- *Friend of a Friend (FOAF)* - People and information on the web
- *The PROV Ontology (PROV-O)* - Provenance information
- *Simple Knowledge Organization System (SKOS)* - Knowledge organization systems
- *The City ontology* - Example ontology aimed at demonstrating the usage of SimPhoNy

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

Do not hesitate to *contact us* if you want your ontology to be shipped with SimPhoNy.

4.2.1 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

To install the EMMO ontology, use

```
pico install emmo
```

A few *EMMO domain ontologies* are also included, and may be installed passing the adequate package identifier to `pico install`.

- Datamodel ontology (version 0.0.2) - `emmo-datamodel`

4.2.2 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`, `dcelements`, `dcam` and `dcterms` RDFS vocabularies from the Dublin Core Metadata Initiative (DCMI), use

```
pico install dcmitype dcelements dcam dcterms
```

4.2.3 Data Catalog Vocabulary (DCAT)

> 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 DCAT ontology (version 2), use

```
pico install dcat
```

4.2.4 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
```

4.2.5 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
```

4.2.6 Simple Knowledge Organization System (SKOS)

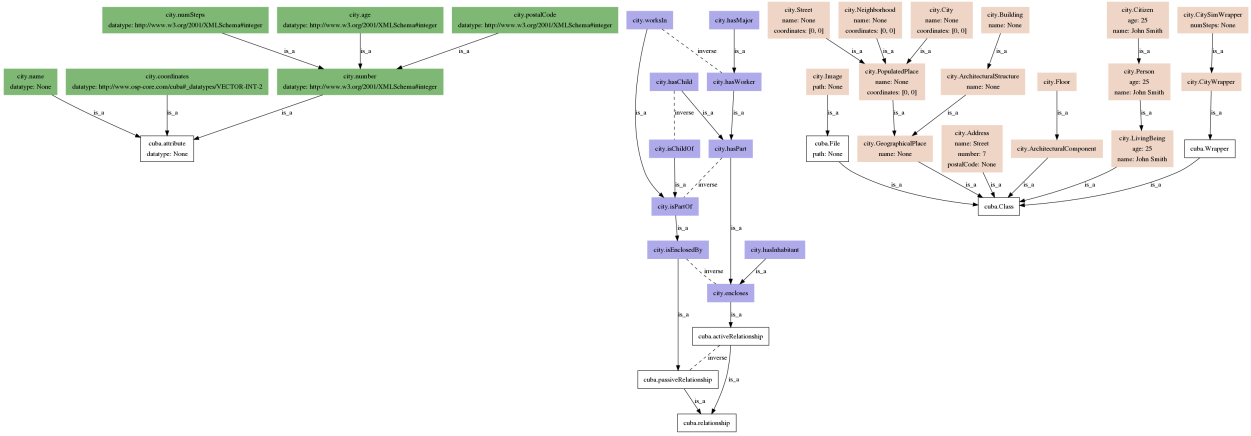
> SKOS is an area of work developing specifications and standards to support the use of knowledge organization systems (KOS) such as thesauri, classification schemes, subject heading systems and taxonomies within the framework of the Semantic Web.
>
> —Introduction to SKOS

To install SKOS, use

```
pico install skos
```

4.2.7 The City ontology

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



To install the City ontology, use

```
pico install city
```

4.3 Ontology packages

Ontologies can be added to SimPhoNy by *installing* ontology packages, which are YAML configuration files that, in addition to pointing to the actual ontology files, also define extra metadata.

An example of an ontology package file is shown below. A *description of each keyword* is provided right after the example.

```yaml
identifier: dcat
ontology_file: https://www.w3.org/ns/dcat2.rdf
format: "application/rdf+xml"
requirements:
  - dcterms
  - prov
  - foaf
namespaces:
  dcat: http://www.w3.org/ns/dcat#
  # another_namespace: http://www.w3.org/ns/example_namespace#
```

4.3.1 Keywords

identifier: Can be any alphanumerical string. It is the name of the ontology package. It is used for uninstallation (e.g. `pico uninstall dcat`) and dependency verification.

ontology_file: Path to the (inferred if applicable) ontology file. 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. When not provided, it will be guessed from the file extension, although such guess may not always be correct. The supported ontology languages and serialization formats are listed *here*.

requirements: A list of identifiers of other ontology packages that this one depends on. An ontology depends on another ontology if it references classes, relationships or attributes from it (e.g. by defining a subclass relationship).

namespaces: Mappings that give each IRI prefix in the ontology file a namespace name, used to import the namespace within SimPhoNy. If any of the provided IRIs do not end with "/" or "#", "#" will be automatically added.

4.4 Supported formats

SimPhoNy supports ontologies in any of the following ontology languages

- Web Ontology Language 2 (OWL 2) (see *limitations*)
- Resource Description Framework Schema (RDFS)

that are serialized in any of the formats that the RDFLib library 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`).

4.4.1 Limitations

At the moment, there are significant limitations on the supported features of OWL 2 ontologies.

OWL 2 ontologies

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

- `RDF.type` to determine the type of the entities.
- `RDFS.label` and `SKOS.prefLabel` to get the entities by label.
- `RDFS.subClassOf` / `RDFS.subPropertyOf` for subclasses and subproperties.
- `OWL.inverseOf` for inverse relationships.
- `RDFS.range` to determine the datatype of `DataProperties`.
- 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.
- Restrictions and compositions are also supported. They can be consulted using the `axioms` attribute of ontology classes.

We try to enlarge this list over time and support more of the OWL DL specification.

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

CHAPTER

FIVE

TERMINOLOGICAL KNOWLEDGE

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, attributes and annotations (terminological knowledge). This page **focuses on** how to access and navigate the **terminological knowledge** of an ontology using SimPhoNy.

Such functionality is presented in the form of a tutorial, in which both the `city` namespace from SimPhoNy's example *City ontology*, and the `emmo` namespace from the *Elementary Multiperspective Material Ontology (EMMO)* are used as examples.

If you want to follow the tutorial along, please *make sure that such ontologies are installed*. If you have not installed them yet, you can do so running the commands below.

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

Note

SimPhoNy does **not** feature the ability to edit the classes, relationships, attributes and annotations of ontologies, only to read them. This is the reason why ontologies need to be *installed using pico*.

5.1 Namespace objects: accessing entities

To access ontology entities, it is first needed to know the aliases of the installed ontology namespaces. The *pico ontology management tool* to can list the installed namespaces. Note that each namespace is provided by a specific *ontology package*.

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

```
Packages:
        - simlammps
        - city
        - emmo
Namespaces:
        - simphony
        - owl
        - rdfs
        - simlammps
        - city
        - emmo
```

Once the names of the namespaces to be used are known, they can be imported in Python. In this tutorial, the namespaces `city` and `emmo` are imported. Through those imported namespace Python objects, the entities within the namespaces can be accessed:

```
[3]: from simphony_osp.namespaces import city, emmo
```

The namespace objects are Python iterables. This implies that it is possible to get a list of all the entities available within a namespace running `list(namespace)`.

To get the IRI of a namespace object, use the `iri` property. This will yield an URIRef object from the RDFLib library, that SimPhoNy makes use of.

```
[4]: city.iri
[4]: rdflib.term.URIRef('https://www.simphony-project.eu/city#')
```

There are several ways to reference an ontology entity to be retrieved, which are summarized in the following list.

- By **suffix**. For example, for the namespace `city`, whose IRI is `http://www.simphony-osp.eu/city#`, requesting the suffix `Citizen` would return the ontology entity with IRI `http://www.simphony-osp.eu/city#Citizen`.
- By **label**. Retrieves the entity by the **main label** that has been assigned to it in the ontology using either the `rdfs:label` or `skos:prefLabel` predicates.
- By **IRI**. The IRI of an ontology entity is provided in order to retrieve it.

Although it is not the only possible approach, the **most convenient way** to access an ontology entity is to use the **dot notation** on the imported namespace objects (e.g. `city.Citizen`) to reference it either by suffix or label.

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 or label

Let's retrieve the *Living Being* class from the `city` namespace, whose IRI is `https://www.simphony-project.eu/city#LivingBeing`.

To retrieve the class using its suffix or its label, just use the dot notation

```
[5]: city.LivingBeing  # by suffix
[5]: <OntologyClass: Living Being https://www.simphony-project.eu/city#LivingBeing>
```

However, sometimes the suffix or label contains characters that Python does not accept as attribute names, such as dashes or spaces. In such cases, it is not possible to use the dot notation. An alternative way to retrieve entitites using the Python's index operator `[]` exists

```
[6]: city['Living Being']  # by label (contains a space)
[6]: <OntologyClass: Living Being https://www.simphony-project.eu/city#LivingBeing>
```

Tip

The index notation also supports IPython autocompletion.

Note that both operations are case-sensitive, and therefore the following would produce an error.

```
[7]: # city.Livingbeing  # -> Fails.
```

In addition, the namespace object has some advanced methods, *from_suffix*, *from_label* and *get* that can also be used to retrieve entities by suffix or label.

Accessing an ontology entity by IRI

Let's now retrieve the Integer class from the `emmo` namespace using its IRI `http://emmo.info/emmo#EMMO_f8bd64d5_5d3e_4ad4_a46e_c30714fecb7f`.

To do so, use the method *from_iri* of the corresponding namespace object

```
[8]: emmo.from_iri('http://emmo.info/emmo#EMMO_f8bd64d5_5d3e_4ad4_a46e_c30714fecb7f')
[8]: <OntologyClass: Integer http://emmo.info/emmo#EMMO_f8bd64d5_5d3e_4ad4_a46e_c30714fecb7f>
```

5.2 Ontology entity objects

5.2.1 Accessing an entity's label

Labels are tipically assigned to ontology entities through the `rdfs:label` or `skos:prefLabel` predicates. In particular, it is possible for a single entity to have several labels.

In SimPhoNy, all such labels can be seen, but one label is considered to be the *main* label. When SimPhoNy has to select exclusively one label among all the available ones, it will first retrieve all the available labels for an entity an then sort them based on the following criteria:

- The predicate used to assign the label. Both `rdfs:label` and `skos:prefLabel` are considered, but labels assigned using `rdfs:label` are preferred to labels assigned to `skos:prefLabel`.
- The language of the label. English labels are preferred to labels with no language assigned to them, and labels with no language to labels in any other language.

So for example, assume that the ontology class

```
[9]: city.City
[9]: <OntologyClass: City https://www.simphony-project.eu/city#City>
```

has as labels (which is actually not the case): *Municipality (language: "en", predicate: ``skos:prefLabel``)*, *City (language: None, predicate: ``rdfs:label``)*, *Città (language: 'it', predicate: ``rdfs:label``)*. Then the second one, *City*, would be the main label.

The main label of an ontology entity can be accessed using the *label* attribute.

```
[10]: city.City.label
[10]: 'City'
```

It is also possible to access the language of the main label,

```
[11]: city.City.label_lang
[11]: 'en'
```

and even the main label in the form of an RDFLib Literal, which contains both the label itself and its language information.

[12]:
```
city.City.label_literal
```
[12]:
```
rdflib.term.Literal('City', lang='en')
```

To retrieve all labels as RDFLib Literals, use the *iter_labels* method.

[13]:
```
list(city.City.iter_labels())
```
[13]:
```
[rdflib.term.Literal('City', lang='en')]
```

As you can see, actually the City class has only one label!

5.2.2 Accessing an entity's identifier and namespace

The identifier (IRI or blank node identifier) of an entity may be accessed using the *identifier* property.

[14]:
```
emmo.Real.identifier
```
[14]:
```
rdflib.term.URIRef('http://emmo.info/emmo#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.

[15]:
```
emmo.Equation.namespace
```
[15]:
```
<emmo: http://emmo.info/emmo#>
```

5.2.3 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:

[16]:
```
from IPython.display import display

print(f"Superclasses of \"Living Being\": {city.LivingBeing.superclasses}", end="\n"*2)
print(f"Subclasses of \"Living Being\": {city.LivingBeing.subclasses}", end="\n"*2)

print(f"Direct superclasses of \"Living Being\": {city.LivingBeing.direct_superclasses}",
↪ end="\n"*2)
print(f"Direct subclasses of \"Living Being\": {city.LivingBeing.direct_subclasses}",␣
↪end="\n"*2)

print("Is \"Person\" a subclass of \"Living Being\"?", city.Person.is_subclass_of(city.
↪LivingBeing))
print("Is \"Living Being\" a superclass of \"Person\"?", city.LivingBeing.is_superclass_
↪of(city.Person))
```
```
Superclasses of "Living Being": frozenset({<OntologyClass: Living Being https://www.
↪simphony-project.eu/city#LivingBeing>, <OntologyClass: http://www.w3.org/2002/07/owl
↪#Thing>})

Subclasses of "Living Being": frozenset({<OntologyClass: Person https://www.simphony-
↪project.eu/city#Person>, <OntologyClass: Living Being https://www.simphony-project.eu/
↪city#LivingBeing>, <OntologyClass: Citizen https://www.simphony-project.eu/city
↪#Citizen>})
```

(continues on next page)

(continued from previous page)

```
Direct superclasses of "Living Being": frozenset()

Direct subclasses of "Living Being": frozenset({<OntologyClass: Person https://www.
↪simphony-project.eu/city#Person>})

Is "Person" a subclass of "Living Being"? True
Is "Living Being" a superclass of "Person"? True
```

5.2.4 Checking the type of entity

In the terminological knowledge side of the ontology, four types of entities can be defined: *classes*, *relationships*, *attributes* and *annotations*. There are different Python objects for the different entity types. Both can be used to check to which type an entity belongs:

[17]:
```
# These are the Python classes for the different types of ontology entities.
# They all inherit from `simphony_osp.ontology.OntologyEntity`.
from simphony_osp.namespaces import owl, rdfs
from simphony_osp.ontology import (
    OntologyAnnotation,
    OntologyAttribute,
    OntologyClass,
    OntologyRelationship,
)

print("\nIs the entity is a class?")
print(isinstance(city.LivingBeing, OntologyClass))
print(city.LivingBeing.is_subclass_of(owl.Thing))
print(not city.LivingBeing.is_subclass_of(owl.topObjectProperty)
      and not city.LivingBeing.is_subclass_of(owl.topDataProperty))

print("\nIs the entity is a relationship?")
print(isinstance(city.hasInhabitant, OntologyRelationship))
print(city.hasInhabitant.is_subclass_of(owl.topObjectProperty))

print("\nIs the entity an attribute?")
print(isinstance(city['name'], OntologyAttribute))
print(city['name'].is_subclass_of(owl.topDataProperty))

print("\nIs the entity an annotation?")
print(isinstance(rdfs.label, OntologyAnnotation))
```

```
Is the entity is a class?
True
True
True

Is the entity is a relationship?
True
True
```

(continues on next page)

(continued from previous page)

```
Is the entity an attribute?
True
True

Is the entity an annotation?
True
```

All such objects **inherit from the ontology entity object**, and thus, share its common functionality, but each differs in the extra operations they offer.

5.2.5 Ontology class objects

One of the extra functionalities offered by ontology class objects is the access to their attributes. The attributes of an ontology class are those that all instances of a class are expected to share. For example, in OWL ontologies they are defined using the `owl:hasValue`, `owl:someValuesFrom`, `owl:cardinality` or `owl:minCardinality` predicates.

[18]:
```
city.Citizen.attributes  # returns a dictionary whose keys are attributes and whose␣
↪values are their default values (if defined)
```

[18]:
```
mappingproxy({<OntologyAttribute: name https://www.simphony-project.eu/city#name>: None,
              <OntologyAttribute: age https://www.simphony-project.eu/city#age>: None})
```

In addition, SimPhoNy 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.

[19]:
```
tuple(str(x) for x in city.Citizen.axioms)
```

[19]:
```
('is child of QUANTIFIER.MAX 2',
 'name QUANTIFIER.EXACTLY 1',
 'works in QUANTIFIER.MAX 1',
 'age QUANTIFIER.EXACTLY 1')
```

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.

[20]:
```
from simphony_osp.ontology import RESTRICTION_QUANTIFIER, RESTRICTION_TYPE

restriction = next(iter(city.Citizen.axioms))
print(restriction)
print(restriction.quantifier)
print(restriction.target)
print(restriction.rtype)
print(
```

(continues on next page)

(continued from previous page)

```
    restriction.attribute
    if restriction.rtype == RESTRICTION_TYPE.ATTRIBUTE_RESTRICTION else
    restriction.relationship
)
```

```
is child of QUANTIFIER.MAX 2
QUANTIFIER.MAX
2
RTYPE.RELATIONSHIP_RESTRICTION
is child of
```

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

[21]:
```
from simphony_osp.ontology import COMPOSITION_OPERATOR, Composition

composition = tuple(x for x in emmo.Integer.axioms if isinstance(x, Composition))[0]
print(composition)
print(composition.operator)
print(composition.operands)
```

```
(SymbolicConstruct OPERATOR.OR Symbol)
OPERATOR.OR
(<OntologyClass: SymbolicConstruct http://emmo.info/emmo#EMMO_89a0c87c_0804_4013_937a_
↪6fe234d9499c>, <OntologyClass: Symbol http://emmo.info/emmo#EMMO_a1083d0a_c1fb_471f_
↪8e20_a98f881ad527>)
```

5.2.6 Ontology relationship objects

You can access the inverse of relationships.

[22]:
```
print("\nYou can get the inverse of a relationship")
print(city.hasInhabitant.inverse)
```

```

You can get the inverse of a relationship
None
```

5.2.7 Ontology attribute objects

The datatype of attributes can be accessed.

[23]:
```
city.age.datatype
```

[23]:
```
rdflib.term.URIRef('http://www.w3.org/2001/XMLSchema#integer')
```

5.2.8 Ontology annotation objects

There is **no specific extra functionality** offered by ontology annotation objects.

CHAPTER

SIX

ASSERTIONAL KNOWLEDGE

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, attributes and annotations (terminological knowledge). This page **focuses on** how to access, edit and navigate the **assertional knowledge** of an ontology using SimPhoNy.

Such functionality is presented in the form of a tutorial, in which the city namespace from SimPhoNy's example City ontology, the emmo namespace from the Elementary Multiperspective Material Ontology (EMMO) are used as examples. If you want to follow the tutorial along, please make sure that both ontologies are installed. If it is not the case, you can install them by running the command below.

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

Moreover, this tutorial concentrates on how to interact with *ontology individual objects*. Each ontology individual object represents a single individual in the ontology.

6.1 Instantiating ontology individuals

On this page, examples are based **exclusively on newly created ontology individuals**. You can learn how to retrieve existing ontology individuals from a data source in the next sections.

To instantiate a new ontology individual, just call an ontology class object as shown below. If the words "ontology class object" sound new to you, please read the *previous section*.

Certain attributes of the ontology individual can already be set at creation time by passing their values as keyword arguments, where the keyword is any of the attribute labels or its namespace suffix. Such attributes are, specifically, the ones returned by the *attributes property* and the *optional attributes property* of the ontology class being called.

[2]:
```
from simphony_osp.namespaces import city, emmo, owl, rdfs, simphony

print(
    f'The following attributes of a new {city.Citizen} '
    f'individual can be set using keyword arguments:'
)
for attribute in set(city.Citizen.attributes) | city.Citizen.optional_attributes:
    print(f'  - {attribute}')

city.Citizen(name="Test Person", age=42)
```

```
The following attributes of a new Citizen individual can be set using keyword arguments:
  - name
  - age
```

[2]:
```
<OntologyIndividual: ea9cfe74-65e4-4ebc-ac27-8a66de82866b>
```

In fact, if any of the attributes is defined in the ontology as *mandatory* using ontology axioms, you will be forced to provide them in the function call (otherwise an exception will be raised).

Tip

In Python, you can pass keyword arguments with spaces or other characters not typically allowed in keyword arguments by unpacking a dictionary in the function call: `city.Citizen(name="Test Person", **{"age": 42})`.

Note

At the moment, it is not possible to instantiate multi-class individuals. We are aware of this issue, and planning to include this functionality in a future minor release.

Until this is fixed, the suggested workaround is to instantiate an ontology individual of any class and change the classes *a posteriori*, just as shown below.

```
person = owl.Thing()

person.classes = city.Citizen, emmo.Cogniser
```

By default, new ontology individuals are assigned a random IRI from the *simphony-osp.eu* domain.

[3]:
```
city.Citizen(name="Test Person", age=42).identifier
```

[3]:
```
rdflib.term.URIRef('https://www.simphony-osp.eu/entity#6b7cf472-9dbe-4d9e-93e2-
↪0f56ee308d27')
```

However, it is possible to fix the identifier using the `iri` keyword argument.

[4]:
```
city.Citizen(
    name="Test Person", age=42,
    iri='http://example.org/entity#test_person'
).identifier
```

[4]:
```
rdflib.term.URIRef('http://example.org/entity#test_person')
```

An individual can also be instantiated in a session different from the default one using the `session` keyword argument (see the sessions section).

6.2 Ontology individual objects

Ontology individuals are a special type of ontology entities, and thus, the ontology individual objects inherit from *ontology entity objects*, meaning that they share their functionality.

In SimPhoNy, an ontology individual is characterized by

- the information about the ontology individual itself such as the classes it belongs to, its label and its attributes;
- the connections to other ontology individuals.

Moreover, such information is stored on a so-called *session* (see next section).

As said, ontology individual objects inherit from *ontology entity objects*. Therefore, it is also possible to access their label, identifier, namespace and super- or subclasses. Below you can find an example. Head to the *terminological knowledge* section for more details.

Note

Even though ontology individual objects share the functionality of *ontology entity objects*, there are some slight differences to consider:

- The *namespace property* tipically returns `None`, regardless of the IRI of the ontology individual. This happens because in order to belong to a namespace, an ontology entity needs not only to have an IRI that contains the namespace IRI, but also to belong to the same session. Ontologies installed with pico live in their own, separate session.
- The *superclasses*, *direct_superclasses*, *subclasses* and *direct_subclasses* properties, as well as the *is_subclass_of* method refer to the superclasses and subclasses of all the classes the ontology individual belongs to, as illustrated in the example.
- The properties *label*, *label_lang*, *label_literal* and *session* are **writable**. This means that both the main label of ontology individuals can be changed and the individuals themselves may be moved from one session to another by changing the value of such properties.

[5]:
```
person = emmo.Cogniser()
# Instantiate an ontology individual of class Cogniser. According to the EMMO's
# documentation, a Cogniser is defined as:
# > An interpreter who establish the connection between an icon an an object
# > recognizing their resemblance (e.g. logical, pictorial)
# The following example for a Cogniser is provided:
# > The scientist that connects an equation to a physical phenomenon.

person.label, person.label_lang = "My neighbor", "en"

print("Label:", f"{person.label} ({person.label_lang})", end='\n'*2)
print("Label literal:", person.label_literal.__repr__(), end='\n'*2)
print("List of labels:", list(person.iter_labels()).__repr__(), end='\n'*2)
print("Identifier:", person.identifier.__repr__(), end='\n'*2)
print("Namespace:", person.namespace.__repr__(), end='\n'*2)

print('Superclasses:', person.superclasses, end='\n'*2)
print('Subclasses:', person.subclasses, end='\n'*2)
print('Direct superclasses:', person.direct_superclasses, end='\n'*2)
print('Direct subclasses:', person.direct_subclasses, end='\n'*2)

print("Does any of the classes of the individual belong the \"Semiotics\" branch of EMMO?
↪", person.is_subclass_of(emmo.Semiotics))

from simphony_osp.ontology import OntologyIndividual
print("\nIs the entity an individual?", isinstance(person, OntologyIndividual))
```

```
Label: My neighbor (en)

Label literal: rdflib.term.Literal('My neighbor', lang='en')

List of labels: [rdflib.term.Literal('My neighbor', lang='en')]

Identifier: rdflib.term.URIRef('https://www.simphony-osp.eu/entity#03659fa1-5c91-44a0-
↪a73c-f475d3b328fe')
```

(continues on next page)

(continued from previous page)

```
Namespace: None

Superclasses: frozenset({<OntologyClass: CausalObject http://emmo.info/emmo#EMMO_
↪c5ddfdba_c074_4aa4_ad6b_1ac4942d300d>, <OntologyClass: Semiotics http://emmo.info/emmo
↪#EMMO_8bb6b688_812a_4cb9_b76c_d5a058928719>, <OntologyClass: EMMO http://emmo.info/emmo
↪#EMMO_802d3e92_8770_4f98_a289_ccaaab7fdddf>, <OntologyClass: Item http://emmo.info/emmo
↪#EMMO_eb3a768e_d53e_4be9_a23b_0714833c36de>, <OntologyClass: CausalSystem http://emmo.
↪info/emmo#EMMO_e7aac247_31d6_4b2e_9fd2_e842b1b7ccac>, <OntologyClass: SemioticEntity␣
↪http://emmo.info/emmo#EMMO_b803f122_4acb_4064_9d71_c1e5fd091fc9>, <OntologyClass:␣
↪Interpreter http://emmo.info/emmo#EMMO_0527413c_b286_4e9c_b2d0_03fb2a038dee>,
↪<OntologyClass: http://www.w3.org/2002/07/owl#Thing>, <OntologyClass: Perspective http:
↪//emmo.info/emmo#EMMO_49267eba_5548_4163_8f36_518d65b583f9>, <OntologyClass: Cogniser␣
↪http://emmo.info/emmo#EMMO_19608340_178c_4bfd_bd4d_0d3b935c6fec>})

Subclasses: frozenset({<OntologyClass: Cogniser http://emmo.info/emmo#EMMO_19608340_178c_
↪4bfd_bd4d_0d3b935c6fec>})

Direct superclasses: frozenset({<OntologyClass: Interpreter http://emmo.info/emmo#EMMO_
↪0527413c_b286_4e9c_b2d0_03fb2a038dee>})

Direct subclasses: frozenset()

Does any of the classes of the individual belong the "Semiotics" branch of EMMO? True

Is the entity an individual? True
```

In addition, ontology individuals have extra functionality that is specific to them.

For example, there is an extra method to verify whether they are an instance of a specific ontology class (which is just an alias for is_subclass_of).

[6]:
```
person.is_a(emmo.Semiotics)
```

[6]:
```
True
```

It is also is possible not only to verify the classes that the individual belongs to,

[7]:
```
person.classes
```

[7]:
```
frozenset({<OntologyClass: Cogniser http://emmo.info/emmo#EMMO_19608340_178c_4bfd_bd4d_
↪0d3b935c6fec>})
```

but also to **change** them.

[8]:
```
person.classes = city.Citizen, emmo.Cogniser

person.classes
```

[8]:
```
frozenset({<OntologyClass: Citizen https://www.simphony-osp.eu/city#Citizen>,
           <OntologyClass: Cogniser http://emmo.info/emmo#EMMO_19608340_178c_4bfd_bd4d_
↪0d3b935c6fec>})
```

To get the session an individual belongs to, use the *session property*. Remember that this property can be also changed in order to transfer the individual from one session to another.

```
[9]: person.session
```

```
[9]: <simphony_osp.session.session.Session at 0x7f7855549fa0>
```

6.2.1 Managing attributes, relationships and annotations

Using the index operator []

SimPhoNy features a single, unified syntax based on the Python index `[]` operator to manage the relationships between ontology individuals, the values of the attributes of an individual, and the values of ontology annotations.

For example, assume one wants to create a city with several neighborhoods and inhabitants. The first step is to instantiate the ontology individuals that represent such elements.

```
[10]: freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])

neighborhoods = {
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Altstadt', [47.99525, 7.84726]),
        ('Stühlinger', [47.99888, 7.83774]),
        ('Neuburg', [48.00021, 7.86084]),
        ('Herdern', [48.00779, 7.86268]),
        ('Brühl', [48.01684, 7.843]),
    ]
}

citizen_1 = city.Citizen(name='Nikola', age=35)
citizen_2 = city.Citizen(name='Lena', age=70)
```

The next step is connecting them, modifying the values of their attributes and adding annotations.

Let's start trying to declare that the neighborhoods are part of the city and that the citizens are inhabitants of the city using the `city.hasPart` and `city.hasInhabitant` relationships.

The individuals that are already connected to the city through this relationship can be consulted as follows.

```
[11]: freiburg[city.hasPart]
```

```
[11]: set() <has part of ontology individual 1fa8d37a-d5de-42f9-ace1-8009c506bd8c>
```

The above statement yields a *relationship set object*. Relationship sets are set-like objects that manage the ontology individuals that are linked to the given individual and relationship (in this example, `freiburg` and `city.hasPart`). You will notice in the following examples, that relationship set objects have a few extra capabilities that Python sets do not have that make the interaction with them more natural.

Note

Set-like objects are objects compatible with the standard Python sets, meaning that all the methods and functionality from Python sets are available for set-like objects.

In order to attach items through the given relationship, all that is needed is an **in-place** set union.

```
[12]: freiburg[city.hasPart] | neighborhoods  # does not attach the neighborhoods
print(freiburg[city.hasPart])
freiburg[city.hasPart] |= neighborhoods  # attaches the neighborhoods (in-place union)
```

(continues on next page)

(continued from previous page)

```
print(freiburg[city.hasPart])

freiburg[city.hasInhabitant] += citizen_1, citizen_2  # attaches the citizens
# the '+=` operator is not available in standard Python sets and is a shorthand for
# the following operations:
# - `+= citizen_1, citizen_2` is equivalent to `|= {citizen_1, citizen_2}`
# - `+= {citizen_1, citizen_2}` is equivalent to `|= {citizen_1, citizen_2}`
# - `+= [citizen_1, citizen_2]` raises a TypeError (this shortcut only works for tuples␣
↪and set-like objects)
# -  `+= citizen_3` is equivalent to `|= {citizen_3}
# the `-=` operator is avilable in standard Python sets, but has been extended
# to work like in the above examples when used together with non set-like objects.
```

```
set()
{<OntologyIndividual: 5b326b26-0919-46dd-a07f-9d76f2839835>, <OntologyIndividual:␣
↪83b21a8e-060c-4530-a1cf-18bf84c7511a>, <OntologyIndividual: f7c7e3dd-bdd9-4d17-a365-
↪b2d07ac3202f>, <OntologyIndividual: 05b864f1-cec9-486f-98b4-1c802d51261e>,
↪<OntologyIndividual: 2604c106-45f3-4534-9ce3-2bc313469918>}
```

Exactly in the same way, when ontology attributes or ontology annotations are passed to the index operator `[]`, *attribute sets* and *annotation sets* are spawned, which behave similarly to relationship sets.

[13]:
```
# ATTRIBUTES
# - assign one more name to Lena
citizen_2[city['name']] += 'Helena'
print(citizen_2[city['name']].__repr__(), end='\n'*2)

# - change the age of Lena (`=` replaces all the values of the attribute)
print(citizen_2[city.age].__repr__())
citizen_2[city.age] = 55
print(citizen_2[city.age].__repr__())

# ANNOTATIONS
citizen_1[rdfs.comment] = (
    'Lena was born in Berlin, but moved to Freiburg when she was 28 years old.',
    'She likes to go into the woods and get lost in her thoughts.'
)
print(citizen_1[rdfs.comment].__repr__())
```

```
{'Lena', 'Helena'} <name of ontology individual 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>

{70} <age of ontology individual 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>
{55} <age of ontology individual 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>
{'Lena was born in Berlin, but moved to Freiburg when she was 28 years old.', 'She likes␣
↪to go into the woods and get lost in her thoughts.'} <http://www.w3.org/2000/01/rdf-
↪schema#comment of ontology individual cbc45216-955a-4eee-993e-d9169e627128>
```

Note

In SimPhoNy, relationships, attributes and annotations are treated in an ontological sense. This means that when using the corresponding Python object to access or modify them, one is referring not only to such ontology entity, but also to all of its subclasses. You can verify this fact noting that `freiburg[owl.topObjectProperty]` returns all individuals attached to `freiburg`, as all relationships are a subclass of `owl:topObjectProperty`.

Strings can also be used with the index notation `[]` as a shorthand in certain cases

- to access the *attributes of any of the classes* that the individual belongs to, or other attributes that have already been assigned to the individual,
- to access relationships that have already been used to link the inidividual to others,
- to access annotations whose value has been already assigned.

[14]:
```
freiburg["hasInhabitant"]
```

[14]:
```
{<OntologyIndividual: 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>, <OntologyIndividual:␣
↪cbc45216-955a-4eee-993e-d9169e627128>} <has inhabitant of ontology individual 1fa8d37a-
↪d5de-42f9-ace1-8009c506bd8c>
```

[15]:
```
citizen_1["age"]
```

[15]:
```
{35} <age of ontology individual cbc45216-955a-4eee-993e-d9169e627128>
```

Therefore the most relevant use-case of passing strings is accessing information from existing individuals, rather than constructing new ones.

Tip

The index notation `[]` supports IPython autocompletion for strings. When working on a Jupyter notebook, it is possible to get suggestions for the strings that will work for that specific individual by writing `individual["` and pressing TAB.

Even though in this example only a few possibilities of the relationship-, attribute- and annotation sets have been covered, remember that they are compatible with standard Python sets. So hopefully, this introduction should be enough to consider the remaining possibilities on your own: remove elements with `-=`, check if a certain relationship is being used `if freiburg[city.hasInhabitant]:`, loop over elements `for connected_individual in freiburg[city.hasInhabitant]:`, etc.

`del freiburg[city.hasInhabitant]` and `freiburg[city.hasInhabitant] = None` can also be used and are equivalent to `freiburg[city.hasInhabitant] = set()`.

When it comes to **accessing single values** from a relationship-, attribute- or annotation set, there are three built-in shortcuts to make it easier than iterating over them:

- `any()` returns an element from the set in a non-deterministic way. Returns `None` if the set is empty.
- `one()` returns the single element in the set. If the set is empty or has multiple elements, thein the exceptions *ResultEmptyError* or *MultipleResultsError* are respectively raised.
- `all()` returns the set itself, and is therefore redundant. Can be used to improve code readability if needed.

[16]:
```
# print(citizen_2['name'].one())  # Raises `MultipleResultsError`, as Lena has multiple␣
↪names.
print(citizen_1['name'].any())
print(citizen_1['name'].all())

# print(citizen_2[city.hasChild].one())  # Raises `ResultEmptyError`, as Nikola has not␣
↪been declared to have children.
print(citizen_2[city.hasChild].any())
print(citizen_2[city.hasChild].all().__repr__())
```

```
Nikola
{'Nikola'}
None
set() <has child of ontology individual 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>
```

Finally, if it is needed to find individuals that are connected through an *inverse relationship*, the `.inverse` attribute of the relationship sets can be used.

```
[17]: citizen_1[city.hasInhabitant].inverse.one()['name'].one()
```

```
[17]: 'Freiburg'
```

Using the Python dot notation (attributes only)

The Python dot notation can be used to access and set the attributes of individuals in all cases when both strings can be passed to the index notation `[]` and the string is compatible with the Python syntax (e.g. it contains no spaces). See the *previous section for more details*.

```
[18]: citizen_1.name, citizen_1.age
```

```
[18]: ('Nikola', 35)
```

```
[19]: citizen_1.age = 34
      citizen_1.age
```

```
[19]: 34
```

```
[20]: citizen_1.attributes
```

```
[20]: mappingproxy({<OntologyAttribute: name https://www.simphony-osp.eu/city#name>: frozenset(
      ↪{'Nikola'}),
                  <OntologyAttribute: age https://www.simphony-osp.eu/city#age>: frozenset(
      ↪{34})})
```

Tip

The dot notation also supports IPython autocompletion.

The dot notation is limited to attributes with a single value. When several values are assigned to the same attribute, a `RuntimeError` is raised.

It is possible to get a dictionary with all the attributes of an individual and its values using the `attributes` attribute.

```
[21]: citizen_2.attributes
```

```
[21]: mappingproxy({<OntologyAttribute: name https://www.simphony-osp.eu/city#name>: frozenset(
      ↪{'Helena',
                              'Lena'}),
                  <OntologyAttribute: age https://www.simphony-osp.eu/city#age>: frozenset(
      ↪{55})})
```

Using the `get`, `iter`, `connect`, and `disconnect` methods (relationships only)

The method *connect* connects individuals using the given relationship.

```
[22]: # remove the existing connections between Freiburg and its citizens
      del freiburg[city.hasInhabitant]
      print(freiburg[city.hasInhabitant].__repr__())

      # use the connect method to restore them
      freiburg.connect(citizen_1, citizen_2, rel=city.hasInhabitant)
      print(freiburg[city.hasInhabitant].__repr__())
```

```
set() <has inhabitant of ontology individual 1fa8d37a-d5de-42f9-ace1-8009c506bd8c>
{<OntologyIndividual: 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>, <OntologyIndividual:␣
↪cbc45216-955a-4eee-993e-d9169e627128>} <has inhabitant of ontology individual 1fa8d37a-
↪d5de-42f9-ace1-8009c506bd8c>
```

The method *disconnect* disconnects ontology individuals. Optionally a relationship and class filter can be given.

[23]:
```
citizen_3 = city.Citizen(name='Lukas', age=2)
citizen_1.connect(citizen_3, rel=city.hasChild)
print(citizen_1[city.hasChild])

citizen_1.disconnect(citizen_3)  # disconnects citizen_3
print(citizen_1[city.hasChild])

citizen_1.connect(citizen_3, rel=city.hasChild)

citizen_1.disconnect(rel=city.worksIn)  # does not disconnect citizen_3, as the␣
↪relationship does not match the filter
print(citizen_1[city.hasChild])
citizen_1.disconnect(rel=city.hasChild, oclass=city.Building)  # does not disconnect␣
↪citizen_3, as the its class does not match the filter
print(citizen_1[city.hasChild])
citizen_1.disconnect(citizen_3, oclass=city.Citizen)  # disconnect works, as the filters␣
↪match now
print(citizen_1[city.hasChild])
```

```
{<OntologyIndividual: f57f53e1-95b4-4fd2-9272-c5c9a44c42f2>}
set()
{<OntologyIndividual: f57f53e1-95b4-4fd2-9272-c5c9a44c42f2>}
{<OntologyIndividual: f57f53e1-95b4-4fd2-9272-c5c9a44c42f2>}
set()
```

The method *get* is used to obtain the individuals linked through a given relationship. Filters to restrict the results only to specific individuals, relationships and classes, as well as any combination of them can optinally be provided. The *iter* method behaves similarly, but returns an interator instead.

[24]:
```
print(freiburg.get().__repr__(), end='\n'*2)  # returns everything attached to Freiburg␣
↪(a relationship set)

print(freiburg.get(rel=city.hasInhabitant).__repr__(), end='\n'*2)  # returns only the␣
↪citizens (a relationship set)

print(freiburg.get(oclass=city.Citizen).__repr__(), end='\n'*2)  # also returns only the␣
↪citizens (a relationship set)

# filtering specific individuals (can be combined with class and relationship filters)
print(freiburg.get(citizen_1).__repr__())
print(freiburg.get(citizen_1, citizen_2).__repr__())
print(freiburg.get(citizen_1.identifier).__repr__())
print(freiburg.get('https://example.org/city#unknown_citizen').__repr__())
print(freiburg.get(citizen_1, rel=city.hasChild).__repr__())
print(freiburg.get(citizen_1, rel=city.hasInhabitant).__repr__())
print(freiburg.get(citizen_1, rel=city.hasInhabitant, oclass=city.Building).__repr__())
```

```
{<OntologyIndividual: 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>, <OntologyIndividual:␣
↪5b326b26-0919-46dd-a07f-9d76f2839835>, <OntologyIndividual: cbc45216-955a-4eee-993e-
↪d9169e627128>, <OntologyIndividual: 83b21a8e-060c-4530-a1cf-18bf84c7511a>,
↪<OntologyIndividual: f7c7e3dd-bdd9-4d17-a365-b2d07ac3202f>, <OntologyIndividual:␣
↪05b864f1-cec9-486f-98b4-1c802d51261e>, <OntologyIndividual: 2604c106-45f3-4534-9ce3-
↪2bc313469918>} <http://www.w3.org/2002/07/owl#topObjectProperty of ontology individual␣
↪1fa8d37a-d5de-42f9-ace1-8009c506bd8c>

{<OntologyIndividual: 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>, <OntologyIndividual:␣
↪cbc45216-955a-4eee-993e-d9169e627128>} <has inhabitant of ontology individual 1fa8d37a-
↪d5de-42f9-ace1-8009c506bd8c>

{<OntologyIndividual: 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>, <OntologyIndividual:␣
↪cbc45216-955a-4eee-993e-d9169e627128>} <http://www.w3.org/2002/07/owl
↪#topObjectProperty of ontology individual 1fa8d37a-d5de-42f9-ace1-8009c506bd8c>

<OntologyIndividual: cbc45216-955a-4eee-993e-d9169e627128>
(<OntologyIndividual: cbc45216-955a-4eee-993e-d9169e627128>, <OntologyIndividual:␣
↪605663a6-0642-4a6a-82b6-cc7f5e04ab9b>)
<OntologyIndividual: cbc45216-955a-4eee-993e-d9169e627128>
None
None
<OntologyIndividual: cbc45216-955a-4eee-993e-d9169e627128>
None
```

Using the `get` and `iter` methods, it is also possible to discover the specific relationships that connect two individuals when a superclass of them is given.

[25]:
```
freiburg.get(rel=owl.topObjectProperty, return_rel=True)
```

[25]:
```
((<OntologyIndividual: 5b326b26-0919-46dd-a07f-9d76f2839835>,
  <OntologyRelationship: has part https://www.simphony-osp.eu/city#hasPart>),
 (<OntologyIndividual: 83b21a8e-060c-4530-a1cf-18bf84c7511a>,
  <OntologyRelationship: has part https://www.simphony-osp.eu/city#hasPart>),
 (<OntologyIndividual: f7c7e3dd-bdd9-4d17-a365-b2d07ac3202f>,
  <OntologyRelationship: has part https://www.simphony-osp.eu/city#hasPart>),
 (<OntologyIndividual: 05b864f1-cec9-486f-98b4-1c802d51261e>,
  <OntologyRelationship: has part https://www.simphony-osp.eu/city#hasPart>),
 (<OntologyIndividual: 2604c106-45f3-4534-9ce3-2bc313469918>,
  <OntologyRelationship: has part https://www.simphony-osp.eu/city#hasPart>),
 (<OntologyIndividual: 605663a6-0642-4a6a-82b6-cc7f5e04ab9b>,
  <OntologyRelationship: has inhabitant https://www.simphony-osp.eu/city#hasInhabitant>),
 (<OntologyIndividual: cbc45216-955a-4eee-993e-d9169e627128>,
  <OntologyRelationship: has inhabitant https://www.simphony-osp.eu/city#hasInhabitant>))
```

6.2.2 Operations

Operations are actions (written in Python) that can be executed on instances of specific ontology classes that they are defined for.

A great example of the applications of operations is the interaction with file objects in SimPhoNy wrappers that support it, for example, the included dataspace wrapper.

[26]:
```
from pathlib import Path
from tempfile import TemporaryDirectory
from urllib import request

from IPython.display import Image

from simphony_osp.wrappers import Dataspace

dataspace_directory = TemporaryDirectory()
example_directory = TemporaryDirectory()

# Download a picture of Freiburg using urllib
# from _Visit Freiburg_ - https://visit.freiburg.de
url = (
    "https://visit.freiburg.de/extension/portal-freiburg"
    "/var/storage/images/media/bibliothek/teaser-bilder-startseite"
    "/freiburg-kunst-kultur-copyright-fwtm-polkowski/225780-1-ger-DE"
    "/freiburg-kunst-kultur-copyright-fwtm-polkowski_grid_medium.jpg"
)
file, response = request.urlretrieve(url)

# Open a dataspace session in a temporary directory
with Dataspace(dataspace_directory.name, True) as session:
    # Create an individual belonging to SimPhoNy's file class
    picture = simphony.File(
        iri='http://example.org/freiburg#my_picture'
    )

    # Use the `upload` operation to assign data to the file object
    picture.operations.upload(file)

    # Commit the changes
    session.commit()

# Access the saved data and retrieve the Picture using the `download` operation
with Dataspace(dataspace_directory.name, True) as session:
    picture = session.from_identifier('http://example.org/freiburg#my_picture')
    download_path = Path(example_directory.name) / 'my_picture.jpg'
    picture.operations.download(download_path)

# Uncomment this line to show the downloaded picture
# (you can do so by running the tutorial yourself using Binder)
# Image(download_path)
```

CHAPTER

SEVEN

SESSIONS

7.1 Introduction

In SimPhoNy, *assertional knowledge* is stored in *sessions*. You may think of a session as a "box" were *ontology individuals* can be placed. But sessions go beyond just storing assertional knowledge. Sessions can be connected to *SimPhoNy Wrappers*. Each wrapper is a piece of software that seamlessly translates the assertional knowledge to a form that is compatible with a specific simulation engine, database, data repository or file format.

In order to keep things simple, this section focuses on sessions that are not connected to any wrapper. All the information stored in such sessions is stored in the computer's volatile memory and lost when the Python shell is closed. After having read this section, you can head to the *next one* to learn more about SimPhoNy Wrappers.

All ontology individuals in SimPhoNy are stored in a session. The session where an individual is stored is always accesible through the *session* attribute, which is **writable**. In fact, changing this attribute is one of the several ways to transfer an ontology individual between sessions.

[1]:
```
from simphony_osp.namespaces import city

freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])
freiburg.session
```

[1]:
```
<simphony_osp.session.session.Session at 0x7f0c2c11f940>
```

But where are newly created individuals stored? Indeed, when a new individual is created, it has to be stored in a session. It is possible to pass a *session object* to the call using the `session` keyword argument to choose where the individual will be stored.

[2]:
```
from simphony_osp.session import Session

session_A = Session()  # create a new session
paris = city.City(
    name="Paris", coordinates=[48.85333, 2.34885],
    session=session_A
)
session_A, paris in session_A, freiburg in session_A
```

[2]:
```
(<simphony_osp.session.session.Session at 0x7f0bee95ee80>, True, False)
```

But such argument is optional. When it is not specified, the individual is stored on the so-called *default session*. Every time you work with SimPhoNy in a Python shell, it creates a new session, called *Core Session* and sets it as the default session. The default session can be changed later to any of your choice. The Core Session can be accessed at any time by importing it from the `simphony_osp.session` module.

[3]:
```
from simphony_osp.session import core_session

core_session, freiburg in core_session, paris in core_session
```

[3]:
```
(<simphony_osp.session.session.Session at 0x7f0c2c11f940>, True, False)
```

Note

The interface of the session object has been designed to interact exclusively with ontology individuals, and therefore, this page only shows you how to use sessions to deal with assertional knowledge.

However, there is no technical reason for such limitation. Sessions can actually store any ontology entity (including terminological knowledge). As a curiosity, the entities from the ontologies that you install using pico are stored in a hidden session that is not meant to be directly accessed.

The default session can be temporarily changed using the `with` statement.

[4]:
```
session_B = Session()

# this will be explained later
session_A.locked = True
session_B.locked = True

with session_B:
    london = city.City(name="London", coordinates=[51.50737, -0.12767])

print(paris in session_A, london in session_B)

# Be careful when using the with statement with several session objects:
# keep in mind that the second will be the one set as default.
with session_A, session_B:
    default_session = Session.get_default_session()
    print(default_session is session_A, default_session is session_B)
```

```
True True
False True
```

Sessions actually work in a way similar to databases. To start using them, one first has to "open" or "connect" to them. After that, changes can be performed on the data they contain, but such changes are not made permanent until a "commit" is performed. When one finishes working with them, the connection should be "closed". Unconfirmed changes are lost when the connection is "closed".

In SimPhoNy, all sessions are automatically "opened" when they are created. The "commit" and "close" operations are controlled manually.

In spite of that general rule, for sessions that are not connected to a wrapper, which are the ones being illustrated in this page, the "commit" command actually does nothing, as confirmed changes have nowhere else to go and be made permanent. You can think of commits being automatic in this case. These sessions also do not implement the "close" command.

Therefore, this general rule has just been introduced in order to present a useful mental model for working with *all* sessions, which includes *sessions connected to a wrapper*.

Having said that, it is now simpler to understand the purpose of the *locked attribute of session objects* that appears in the last example. The `with` statement not only sets a session as the default, but also *closes* it when leaving its context. *Locking a session* with the *locked* attribute prevents the session from being closed if one intents to continue using it. To restore the original behavior, set it to `False`.

7.2 Managing the session contents

As said, sessions can be though of as a "box" that stores ontology individuals. Consequently, adding and removing individuals are among the operations that can be performed on them.

As described in *one of the previous sections*, in SimPhoNy, an ontology individual is characterized by

- the information about the ontology individual itself such as the classes it belongs to, its label and its attributes;
- the connections to other ontology individuals.

The methods *add* and delete try to accommodate this definition. To see how they work, consider a city example again.

[1]:
```
from simphony_osp.namespaces import city

freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])

neighborhoods = {
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Altstadt', [47.99525, 7.84726]),
        ('Stühlinger', [47.99888, 7.83774]),
        ('Neuburg', [48.00021, 7.86084]),
        ('Herdern', [48.00779, 7.86268]),
        ('Brühl', [48.01684, 7.843]),
    ]
}

citizen_1 = city.Citizen(name='Nikola', age=35)
citizen_2 = city.Citizen(name='Lena', age=70)

freiburg[city.hasPart] |= neighborhoods
freiburg[city.hasInhabitant] += citizen_1, citizen_2
```

[2]:
```
from simphony_osp.tools import semantic2dot  # explained in a later section

semantic2dot(freiburg, *neighborhoods, citizen_1, citizen_2)
```

[2]:

Tip

On most web browsers (including mobile ones), you can right-click the picture above and then click "open in new tab" to see the picture in its full size.

Then create a new session to transfer individuals to.

[3]:
```
from simphony_osp.session import Session

session = Session()
```

Passing any individual to the method *add* of the new session creates a copy of it. The copy contained in the new session is assigned the same identifier as the original.

[4]:
```
freiburg_copy = session.add(freiburg)
list(session), freiburg_copy.name, freiburg_copy.coordinates, freiburg_copy.get()
```

[4]:
```
([<OntologyIndividual: cc02ad20-4c05-4257-8f40-47bb0e243934>],
 'Freiburg',
 <simphony_osp.utils.datatypes.Vector: array([47.997791, 7.842609])>,
 set() <http://www.w3.org/2002/07/owl#topObjectProperty of ontology individual cc02ad20-
↪4c05-4257-8f40-47bb0e243934>)
```

In the example above, the variable `freiburg` refers now to the individual in the Core Session, while `freiburg_copy` refers to the copy of the individual that has been created in the new session.

As it can be inferred from the fact that the new session contains only one individual, only the *intrinsic* information about the individual has been transferred, but not the connections to other ontology individuals nor any intrinsic information of the other ontology individuals themselves.

To keep the connection to another ontology individual, it is necessary to transfer **both of them together**.

[5]:
```
session.clear()  # this removes all individuals from the session

copies = session.add(freiburg, neighborhoods)  # note that Python iterables of␣
↪individuals can also be passed to `add` (such as `neighborhoods`)
freiburg_copy = copies[0]  # the order in which individuals were added is respected
freiburg_copy.get()
```

[5]:
```
{<OntologyIndividual: 7824a475-52e0-48b3-ac15-95cbe1d11298>, <OntologyIndividual:␣
↪ec1484ec-94df-4bfc-bf37-ece246888089>, <OntologyIndividual: d136fbd0-3eb8-482b-908f-
↪b5755cd94459>, <OntologyIndividual: 4070b09c-ab77-439b-8404-5e4fd1e0662e>,
↪<OntologyIndividual: 18bb022d-3c87-4d0d-8bde-527ffe5b877c>} <http://www.w3.org/2002/07/
↪owl#topObjectProperty of ontology individual cc02ad20-4c05-4257-8f40-47bb0e243934>
```

As shown in the example, the *clear* method removes all ontology individuals from a session. The *delete* method allows to exactly select the individuals to remove. Both ontology individuals and identifiers can be passed to *delete*.

[6]:
```
session.delete(freiburg.identifier, copies[1:])
list(session)
```

[6]:
```
[]
```

A special situation arises when one tries to add an ontology individual to a session that already contains an individual with the same identifier.

[7]:
```
session_A = Session(); session_A.locked = True
with session_A:
    lena_A = city.Citizen(name='Lena', age=70,
                          iri='https://example.org/entities#Lena')

session_B = Session(); session_B.locked = True
with session_B:
    lena_B = city.Citizen(name='Helena', age=31,
                          iri='https://example.org/entities#Lena')
```

[8]:
```
with Session() as session:
    session.add(lena_A)
    session.add(lena_B)  # -> Fails, there is already an individual with the same␣
↪identifier.
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-8-3c658123bfbf> in <module>
      1 with Session() as session:
      2     session.add(lena_A)
----> 3     session.add(lena_B)  # -> Fails, there is already an individual with the␣
↪same identifier.

~/Dokumente/SimPhoNy/simphony-osp/simphony_osp/session/session.py in add(self, merge,␣
↪exists_ok, all_triples, *individuals)
    808             and exists_ok is False
    809         ):
--> 810             raise RuntimeError(
    811                 "Some of the added entities already exist on the session."
    812             )

RuntimeError: Some of the added entities already exist on the session.
```

When this happens, an exception is raised. It is still possible to copy the individual, but is necessary to decide whether:

- the existing individual should be overwritten (default) or;
- the new individual should be merged with the existing one.

To apply default action (overwrite), use the keyword argument `exists_ok=True`.

[9]:
```
with Session() as session:
    copy = session.add(lena_A)
    print(copy.age, copy.name)
    copy = session.add(lena_B, exists_ok=True)
    print(copy.age, copy.name)
```

```
70 Lena
31 Helena
```

Overwriting replaces the classes, attributes, attribute values and connections to other ontology individuals.

To merge the new individual with the previous one, use `exists_ok=True, merge=True` instead.

[10]:
```
with Session() as session:
    copy = session.add(lena_A)
    print(copy.age, copy.name)
    copy = session.add(lena_B, exists_ok=True, merge=True)
    print(copy['age'], copy['name'])
```

```
70 Lena
{70, 31} {'Helena', 'Lena'}
```

Merging combines the classes, atrributes, attribute values and connections of the two individuals.

7.3 Queries: using the session object

SimPhoNy's *session object* implements methods that enable the retrieval of the ontology individuals it contains. The queries that can be run using such methods are extremely basic. If you need to run more advanced queries, head to the sections dealing with the *search module* and *SPARQL*.

Consider a similar example from the city ontology again. Remember that by default, newly created individuals created are stored in the *Core Session*.

[1]:
```
from simphony_osp.namespaces import city, rdfs
from simphony_osp.session import core_session

freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])

neighborhoods = {
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Altstadt', [47.99525, 7.84726]),
        ('Stühlinger', [47.99888, 7.83774]),
        ('Neuburg', [48.00021, 7.86084]),
        ('Herdern', [48.00779, 7.86268]),
        ('Brühl', [48.01684, 7.843]),
    ]
}

citizen_1 = city.Citizen(name='Nikola', age=35,
                         iri='https://example.org/entities#Nikola')
citizen_2 = city.Citizen(name='Lena', age=70,
                         iri='https://example.org/entities#Lena')
citizen_2[rdfs.label] = 'Helena'
citizen_3 = city.Citizen(name='Marco', age=36,
                         iri='https://example.org/entities#Marco')
citizen_3[rdfs.label] = 'Marco'

freiburg[city.hasPart] |= neighborhoods
freiburg[city.hasInhabitant] += citizen_1, citizen_2
```

As shown in a *previous section*, the session object is iterable. This means that a simple method (althought very inefficient) to retrieve the individuals in the session is just iterating over the session object. It is also possible to know how many individuals are contained in a session using the Python built-in `len`.

[8]:
```
list(core_session)  # iterates over the session, very slow with big data
```

[8]:
```
[<OntologyIndividual: bd967f58-048b-45c2-bfba-d0ad7a1d4dc2>,
 <OntologyIndividual: f2f1dfb2-4fb9-4255-8059-9d9a56029721>,
 <OntologyIndividual: b706791a-93be-4192-9b2a-bf9e027ba47b>,
 <OntologyIndividual: 795c9172-cd38-4998-9f52-d6244e39c699>,
 <OntologyIndividual: 6087bd19-c53d-4b47-ad24-ec29b5585902>,
 <OntologyIndividual: a7700db4-5d6f-4eea-9a4c-488aab13b4be>,
 <OntologyIndividual: https://example.org/entities#Nikola>,
 <OntologyIndividual: Helena https://example.org/entities#Lena>]
```

[3]:
```
len(core_session)
```

[3]:
```
9
```

The session object also features a *get* method that is similar to the *get method from ontology individuals*. It returns, in fact, a set-like object that manages the individuals contained in the session. Consequently, it is even possible to add and remove individuals from the session using in-place modifcations (although not very practical). The most common use case of the method is just retrieving ontology individuals.

[4]:
```
core_session.get()
```

Just like for the get method from ontology individuals, it is possible to filter the results to specific individuals (given their identifier is known or they are already available as objects) and classes, as well as any combination of both.

[5]:
```
core_session.get('https://example.org/entities#Lena')
```

[5]:
```
<OntologyIndividual: Helena https://example.org/entities#Lena>
```

[6]:
```
core_session.get(oclass=city.Citizen)
```

[6]:
```
{<OntologyIndividual: https://example.org/entities#Nikola>, <OntologyIndividual: Helena␣
↪https://example.org/entities#Lena>} <class Citizen of session default session>
```

Finally, if a label has been given to the individual, then it is also possible to use it as retrieval method.

[7]:
```
core_session.from_label('Helena')
```

[7]:
```
frozenset({<OntologyIndividual: Helena https://example.org/entities#Lena>})
```

7.4 Queries: using the search module

The search module enables more advanced queries of session's contents when compared to the *methods from the session object*. It is intended to offer a pythonic approach to querying the session, but still, it has significant efficiency and expresivity limitations. An additional constraint is the topology of the graph that the relationships (and annotations) between the individuals form, as the functions from the module work **traversing the graph from a given initial individual**. Therefore, whenever you require more advanced or more performant queries, use *SPARQL*.

SimPhoNy's search module is located under `simphony_osp.tools.search` and can be imported as follows.

[1]:
```
from simphony_osp.tools import search
```

The following functions are available:

- *find*
- *find_by_identifier*
- *find_by_class*
- *find_by_attribute*
- *find_relationships*

A `sparql` function is also provided by the module. However, it is not covered in this section, *but the next one*.

To see how these functions work, let's set-up an example with two cities, Freiburg and Paris.

[2]:
```
from simphony_osp.namespaces import city, owl, rdfs
from simphony_osp.session import core_session

# Create a city called "Freiburg"
freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])
freiburg_neighborhoods = [
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Altstadt', [47.99525, 7.84726]),
        ('Stühlinger', [47.99888, 7.83774]),
        ('Neuburg', [48.00021, 7.86084]),
        ('Herdern', [48.00779, 7.86268]),
        ('Brühl', [48.01684, 7.843]),
    ]
]
freiburg_citizens = {
    city.Citizen(name='Nikola', age=35,
                 iri="http://example.org/entities#Nikola"),
    city.Citizen(name='Lena', age=70,
                 iri="http://example.org/entities#Lena"),
}
freiburg[city.hasPart] |= freiburg_neighborhoods
freiburg[city.hasInhabitant] |= freiburg_citizens

# Create a city called "Paris"
paris = city.City(name="Paris", coordinates=[48.85333, 2.34885])
paris_neighborhoods = {
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Louvre', [48.86466, 2.33487]),
        ('Bourse', [48.86864, 2.34146]),
        ('Temple', [48.86101, 2.36037]),
        ('H^otel-de-Ville', [48.85447, 2.35902]),
        ('Panthéon', [48.84466, 2.3471]),
    ]
}
paris_citizens = {
    city.Citizen(name='François', age=32)
}
paris[city.hasPart] |= paris_neighborhoods
paris[city.hasInhabitant] = paris_citizens
```

[3]:
```
from simphony_osp.tools import semantic2dot  # explained in a later section

semantic2dot(freiburg, *freiburg_neighborhoods, *freiburg_citizens,
             paris, *paris_neighborhoods, *paris_citizens)
```

[3]:

Tip

On most web browsers (including mobile ones), you can right-click the picture above and then click “open in new tab” to see the picture in its full size.

The most important detail to keep in mind is that the functions above do not act on a session object, but on an ontology individual object that has to be provided beforehand. They perform the search by **traversing the relationships and**

annotations in the graph starting from the given individual.

This is the precisely the reason why a directed graph with two weakly connected components is provided as example: it illustrates that it is impossible to find individuals from one of the components if the search function is called on an individual belonging to another component. Head to the first subsection, dealing with the function *find*, to see this behavior in action.

7.4.1 find

The function `find` traverses the graph starting from a given *root* individual, following only relationships that are a subclass of the *given relationships*, and annotations that are a subclass of the *given annotations*. Along the path, it verifies a given *criterion* on each ontology individual it finds, and then returns the matching individuals.

To clarify how it works, several calls of increasing difficulty to this function, applied to the graph constructed at the top of this page, are shown as examples.

The simplest call just needs the positional argument `root`. As no relationships nor annotations are provided, all relationships and annotations are followed. Similarly, as no criterion is provided, it is always assumed to be satisfied.

[4]:
```
found = list(
    search.find(freiburg)
)

semantic2dot(*found)
```

[4]:

As it can be seen above, when using `freiburg` as the root object, the result is the subgraph of all ontology individuals reachable from `freiburg`. None of the citizens or neighborhoods of `paris` are returned, as they cannot be reached from the `freiburg` individual.

To increase the complexity of the query, the relationships to be followed can be limited to the subclasses of various selected relationships using the `rel` keyword arugment. For example, let's restrict the search to inhabitants and workers of Freiburg (as well as Freiburg itself).

[5]:
```
found = list(
    search.find(
        freiburg,
        rel=(city.hasInhabitant, city.hasWorker)  # if only one relationship has to be␣
↪considered, just pass it directly (i.e. `rel=city.hasInhabitant`)
    )
)

semantic2dot(*found)
```

[5]:

Note

There is a similar keyword argument `annotation` that can be used to constrain the ontology annotations to follow. Just like for the `rel` argument, it can take as values either a single annotation or several ones. However, in addition, it also accepts boolean values. Setting it to `True` means following all annotations (the default value), while setting it to `False` means following no annotations at all.

The next complexity step is passing a function as search criterion. For example, one may want to restrict the search to persons that are more than 40 years old.

[6]:
```
found = list(
    search.find(
```

(continues on next page)

(continued from previous page)

```
        freiburg,
        criterion=lambda individual: individual.is_a(city.Person) and individual.age >␣
↪40,
        rel=(city.hasInhabitant, city.hasWorker)
    )
)

semantic2dot(*found)
```

[6]:

Note that `freiburg` itself does not satisfy the given criterion, and thus it is excluded from the results.

That is all! You have mastered the `find` function, although there are still two little details that allow to control the algorithm and the output.

Relaxing the criterion to just requiring the results to be persons and re-running the search with the keyword argument `find_all=False` causes the search to return only the first individual it finds.

[7]:
```
found = search.find(
    freiburg,
    criterion=lambda individual: individual.is_a(city.Person),
    rel=(city.hasInhabitant, city.hasWorker),
    find_all=False,
)

semantic2dot(found)
```

[7]:

Lastly, the maximum depth of the search can be adjusted using the `max_depth` parameter. Try setting it to `0`, it should yield no results!

[8]:
```
found = list(
    search.find(
        freiburg,
        criterion=lambda individual: individual.is_a(city.Person),
        rel=(city.hasInhabitant, city.hasWorker),
        max_depth=0,
    )
)

semantic2dot(*found)
```

[8]:

7.4.2 find_by_identifier

This function is constructed as a call to *find*, but with the criterion fixed to matching the given identifier.

The positional arguments `root` and keyword arguments `rel` and `annotation` behave exactly like in *find*. An additional positional argument, `identifier` is required. The keyword arguments `criterion`, `find_all` and `max_depth` are not available, as they are fixed to: matching the given identifier, `False` (as identifiers are unique) and unlimited.

Therefore, this function yields either `None` or a single ontology individual whose identifier matches the given one.

[9]:
```
found = search.find_by_identifier(
    freiburg,
```

(continues on next page)

(continued from previous page)

```
    "http://example.org/entities#Lena",
    rel=owl.topObjectProperty,
)

semantic2dot(found)
```

[9]:

7.4.3 find_by_class

This function is also constructed as a call to *find*, but with the criterion fixed to the individual belonging to a specific ontology class.

Just like before, the positional arguments `root` and keyword arguments `rel` and `annotation` behave exactly like in *find*. An additional positional argument, `oclass` is required. The keyword arguments `criterion`, `find_all` and `max_depth` are not available, as they are fixed to: the individual belonging to the given class, `True` (as identifiers are unique) and unlimited.

[10]:
```
found = list(search.find_by_class(
    freiburg,
    city.Person,
    rel=owl.topObjectProperty,
))

semantic2dot(*found)
```

[10]:

7.4.4 find_by_attribute

Finds individuals that have a matching attribute value. `find_all` is set to `True` and `max_depth` to unlimited.

For example, looking for individuals that are 35 years old yields Nikola.

[11]:
```
found = list(search.find_by_attribute(
    freiburg,
    city.age,
    35,
    rel=owl.topObjectProperty,
))

semantic2dot(*found)
```

[11]:

7.4.5 find_relationships

In this case, individuals that are attached to any other individual through the given relationship (the *source* individuals of the relationship) are returned. `find_all` is set to `True` and `max_depth` to unlimited.

[12]:
```
found = list(search.find_relationships(
    freiburg,
    city.hasInhabitant,  # the relationship used as criterion
    rel=owl.topObjectProperty,  # the usual `rel` argument, only this and sub-
↪relationships will be traversed during the search
```

(continues on next page)

(continued from previous page)

```
))

semantic2dot(*found)
```

[12]:

Note that however, by default sub-relationships of the given relationship are not considered. This can be demonstrated using `owl.topObjectProperty` as relationship, as no results are yielded when doing so

[13]:
```
found = list(search.find_relationships(
    freiburg,
    owl.topObjectProperty,  # the relationship used as criterion
    rel=owl.topObjectProperty,  # the usual `rel` argument, only this and sub-
↪relationships will be traversed during the search
))

semantic2dot(*found)
```

[13]:

unless the keyword argument `find_sub_relationships` is set to `True`.

[14]:
```
found = list(search.find_relationships(
    freiburg,
    owl.topObjectProperty,  # the relationship used as criterion
    find_sub_relationships=True,
    rel=owl.topObjectProperty,  # the usual `rel` argument, only this and sub-
↪relationships will be traversed during the search
))

semantic2dot(*found)
```

[14]:

7.5 Queries: using SPARQL

SimPhoNy sessions store the ontology individual information using the RDF standard in an RDF graph object from the RDFLib library. This means that they are naturally compatible with the SPARQL 1.1 Query Language for RDF graphs.

SPARQL queries can be invoked both from a function in the search module or from the *sparql method of the session object*. Both are equivalent, except for the fact that a target session can be passed to the function from the search module, whereas for the sparql method of the session object, the target session is fixed.

[1]:
```
from simphony_osp.tools.search import sparql
```

Freiburg and Paris will serve as an example again to showcase this functionality.

[2]:
```
from simphony_osp.namespaces import city, owl, rdfs
from simphony_osp.session import core_session

# Create a city called "Freiburg"
freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])
freiburg_neighborhoods = [
```

(continues on next page)

(continued from previous page)

```
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Altstadt', [47.99525, 7.84726]),
        ('Stühlinger', [47.99888, 7.83774]),
        ('Neuburg', [48.00021, 7.86084]),
        ('Herdern', [48.00779, 7.86268]),
        ('Brühl', [48.01684, 7.843]),
    ]
]
freiburg_citizens = {
    city.Citizen(name='Nikola', age=35,
                 iri="http://example.org/entities#Nikola"),
    city.Citizen(name='Lena', age=70,
                 iri="http://example.org/entities#Lena"),
}
freiburg[city.hasPart] |= freiburg_neighborhoods
freiburg[city.hasInhabitant] |= freiburg_citizens

# Create a city called "Paris"
paris = city.City(name="Paris", coordinates=[48.85333, 2.34885])
paris_neighborhoods = {
    city.Neighborhood(name=name, coordinates=coordinates)
    for name, coordinates in [
        ('Louvre', [48.86466, 2.33487]),
        ('Bourse', [48.86864, 2.34146]),
        ('Temple', [48.86101, 2.36037]),
        ('H^otel-de-Ville', [48.85447, 2.35902]),
        ('Panthéon', [48.84466, 2.3471]),
    ]
}
paris_citizens = {
    city.Citizen(name='François', age=32)
}
paris[city.hasPart] |= paris_neighborhoods
paris[city.hasInhabitant] = paris_citizens
```

Start by getting all objects connected to Freiburg. This will return a query result object. Such object inherits from RDFLib's SPARQLResult object. The example below illustrates the its basic functionality. Check RDFLib's documentation to learn all the capabilities of the SPARQLResult object.

[3]:
```
result = sparql(  # no session specified, uses the default session (Core Session in this␣
↪example)
    f"""SELECT ?o WHERE {{
        <{freiburg.identifier}> ?p ?o .
    }}
    """
)

print(
    len(result),  # number of rows in the result
    bool(result)  # True when at least one match has been found
)
```

(continues on next page)

(continued from previous page)

```
for row in result:  # iterating the result yields ResultRow objects
    print(row.__repr__())
    # ResultRows inherint from tuples
    # the order of the variables passed to the query is respected

    print(row[0].__repr__())  # a specific variable can be accessed using either its␣
↪position,
    print(row['o'].__repr__())  # or name

    print(row.get('unknown_variable', None))  # a dict-like `get` method is available

    print(row.asdict())  # transforms the row into a dictionary

    break  # only one result is shown in order not to flood this page
```

```
10 True
(rdflib.term.Literal('13YFp0RR93AD@t&xBo{#)k4YS)LtJz', datatype=rdflib.term.URIRef(
↪'https://www.simphony-osp.eu/types#Vector')),)
rdflib.term.Literal('13YFp0RR93AD@t&xBo{#)k4YS)LtJz', datatype=rdflib.term.URIRef('https:
↪//www.simphony-osp.eu/types#Vector'))
rdflib.term.Literal('13YFp0RR93AD@t&xBo{#)k4YS)LtJz', datatype=rdflib.term.URIRef('https:
↪//www.simphony-osp.eu/types#Vector'))
None
{'o': rdflib.term.Literal('13YFp0RR93AD@t&xBo{#)k4YS)LtJz', datatype=rdflib.term.URIRef(
↪'https://www.simphony-osp.eu/types#Vector'))}
```

All results from the query are by default RDFLib objects (e.g. `URIRef`, `Literal`, ...). However, query results from SimPhoNy feature the capability to easily convert the results to other data types using keyword arguments.

For example, to query all the citizens in the session, as well as their name name and age; and obtain the results as ontology individual objects, Python strings and Python integers; use the following.

[4]:
```
from simphony_osp.ontology import OntologyIndividual

result = sparql(
    f"""SELECT ?person ?name ?age WHERE {{
        ?person rdf:type <{city.Citizen.identifier}> .
        ?person <{city['name'].identifier}> ?name .
        ?person <{city.age.identifier}> ?age .
    }}
    """
)

for row in result(person=OntologyIndividual, name=str, age=int):
    print(row)
```

```
(<OntologyIndividual: http://example.org/entities#Nikola>, 'Nikola', 35)
(<OntologyIndividual: http://example.org/entities#Lena>, 'Lena', 70)
(<OntologyIndividual: d78308dc-6db8-4216-be15-76fa0072c1c7>, 'François', 32)
```

By default, the ontologies installed with *pico* are not included in the search. If you wish to make use of the terminological knowledge, pass the keyword argument `ontology=True`. The example below looks for persons instead of citizens, therefore including the terminological knowledge is necessary to obtain the desired results.

[5]:
```
result = sparql(
    f"""SELECT ?person ?name ?age WHERE {{
        ?person rdf:type/rdfs:subClassOf <{city.Person.identifier}> .
        ?person <{city['name'].identifier}> ?name .
        ?person <{city.age.identifier}> ?age .
    }}
    """,
    ontology=False
)

print("Query without ontology:", len(result), "results")

result = sparql(
    f"""SELECT ?person ?name ?age WHERE {{
        ?person rdf:type <{city.Citizen.identifier}> .
        ?person <{city['name'].identifier}> ?name .
        ?person <{city.age.identifier}> ?age .
    }}
    """,
    ontology=True
)

print("Query with ontology:", len(result), "results")

for row in result(person=OntologyIndividual, name=str, age=int):
    print(row)
```

```
Query without ontology: 0 results
Query with ontology: 3 results
(<OntologyIndividual: http://example.org/entities#Nikola>, 'Nikola', 35)
(<OntologyIndividual: http://example.org/entities#Lena>, 'Lena', 70)
(<OntologyIndividual: d78308dc-6db8-4216-be15-76fa0072c1c7>, 'François', 32)
```

Note

When using `ontology=True`, the current version of SimPhoNy assumes that there is virtually no latency between your computer and the session that is being queried. If `ontology=True` is used, for example, with a session connected to a triplestore located on a remote server, the query will be extremely slow.

7.6 RDF Import and export

SimPhoNy sessions store the ontology individual information using the RDF standard in an RDF graph object from the RDFLib library. Exporting such RDF graph is possible using the functions *import_file* and *export_file*.

[1]:
```
from simphony_osp.tools import import_file, export_file
```

Tip

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

In the examples on this page, the *city ontology* is used. Make sure the city ontology is installed. If not, run the following command:

[2]:
```
!pico install city
```

Then create a few ontology individuals

[3]:
```
from simphony_osp.namespaces import city

freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])
peter = city.Citizen(name="Peter", age=30)
anne = city.Citizen(name="Anne", age=20)
freiburg[city.hasInhabitant] += peter, anne
```

7.6.1 Exporting individuals

The `export_file` function allows to export either all the contents of a session, or select a few ontology individuals to be exported.

For example, exporting Freiburg and Peter

[4]:
```
export_file({freiburg, peter}, file='./data.ttl', format='turtle')
```

creates the file `data.ttl` with the following content.

[5]:
```
from sys import platform

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

```
@prefix ns1: <https://www.simphony-osp.eu/city#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<https://www.simphony-osp.eu/entity#e0eb6516-b2f8-4323-a407-f4a98bf46a61> a ns1:City ;
    ns1:coordinates "13YFp0RR93AD@t&xBo{#)k4YS)LtJz"^^<https://www.simphony-osp.eu/types
↪#Vector> ;
    ns1:hasInhabitant <https://www.simphony-osp.eu/entity#3153c78e-a5ee-4065-913a-
↪776c33c30c9e> ;
    ns1:name "Freiburg"^^xsd:string .

<https://www.simphony-osp.eu/entity#3153c78e-a5ee-4065-913a-776c33c30c9e> a ns1:Citizen ;
    ns1:age 30 ;
    ns1:name "Peter"^^xsd:string .
```

Note that when individuals that are connected are **exported together**, their connections are kept in the exported file.

If instead, you wish to export all the individuals in a session, then pass the session object to be exported.

[6]:
```
from simphony_osp.session import core_session

export_file(core_session, file='./data.ttl', format='turtle')

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

(continues on next page)

(continued from previous page)

```
else:
    !cat data.ttl
```

```
@prefix city: <https://www.simphony-osp.eu/city#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<https://www.simphony-osp.eu/entity#e0eb6516-b2f8-4323-a407-f4a98bf46a61> a city:City ;
    city:coordinates "13YFp0RR93AD@t&xBo{#)k4YS)LtJz"^^<https://www.simphony-osp.eu/types
↪#Vector> ;
    city:hasInhabitant <https://www.simphony-osp.eu/entity#3153c78e-a5ee-4065-913a-
↪776c33c30c9e>,
        <https://www.simphony-osp.eu/entity#70d74233-989f-4cb2-9b67-3635801b2037> ;
    city:name "Freiburg"^^xsd:string .

<https://www.simphony-osp.eu/entity#3153c78e-a5ee-4065-913a-776c33c30c9e> a city:Citizen␣
↪;
    city:age 30 ;
    city:name "Peter"^^xsd:string .

<https://www.simphony-osp.eu/entity#70d74233-989f-4cb2-9b67-3635801b2037> a city:Citizen␣
↪;
    city:age 20 ;
    city:name "Anne"^^xsd:string .
```

You can change the output format by entering a different value for the parameter `format`. A list of supported formats is available on this page.

7.6.2 Importing individuals

To import data, use the `import_file` method. Let's assume we wish to import the data from the previous example in a new session. The following code will help us achieve our aim:

[7]:
```
from simphony_osp.session import Session
from simphony_osp.tools import import_file

session = Session(); session.locked = True

with session:
    import_file('./data.ttl')
```

Note

The format is automatically inferred from the file extension. To specify it explicitly, you can add the `format` parameter, like so: `import_file('./data.ttl', format='turtle')`.

Note

A `session` keyword argument can be optionally provided. When not specified, data is imported to the default sesion. You can specify the session explicitly like so: `import_file('./data.ttl', session=session)`.

Now we can verify the data was indeed imported:

[8]:
```
from simphony_osp.tools import semantic2dot

semantic2dot(session)
```
[8]:

7.6.3 Interpretation of RDF files

The *ontology languages supported by SimPhoNy* can be serialized as RDF files, but the RDF standard can store data that does not necessarily have anything to do with an ontology. Moreover, as described in the *introduction to sessions*, SimPhoNy sessions have been designed to exlusively store assertional knowledge (ontology individuals).

Due to these factors, SimPhoNy enforces a few constraints when importing and exporting individuals from/to RDF files, that can, however, **also be disabled if desired**.

1. No terminological knowledge can be present in the file. Any RDF triple with predicate `rdf:type` and one of `owl:Class`, `rdfs:Class`, `owl:AnnotationProperty`, `rdf:Property`, `owl:DatatypeProperty`, `owl:ObjectProperty`, `owl:Restriction` as object raises an exception.
2. The subjects of any RDF triple with predicate `rdf:type` and an IRI not listed above as object are considered to be the identifiers of ontology individuals. Therefore, SymPhoNy will look into its installed ontologies for a class matching the object of the triple. If this lookup fails, an exception is raised. This behavior is meant to prevent you from making the mistake of importing data for which you have not installed the corresponding ontology.
3. If an RDF triple where the subject has been succesfully identified as an ontology individual has a predicate different from `rdf:type` that cannot be recognized as an annotation, relationship nor an attribute, an exception will be raised.
4. If an RDF triple where the subject has been succesfully identified as an ontology individual has a predicate that can be recognized as a relationship, but has an object that cannot be recognized as an ontology individual (for example, because no `rdf:type` has been defined for it), then an exception will be raised.
5. If an RDF triple where the subject has been succesfully identified as an ontology individual has a predicate that can be recognized as a relationship, but has a literal as an object, an exception will be raised.
6. If an RDF triple where the subject has been succesfully identified as an ontology individual has a predicate that can be recognized as an attribute, but has a IRI as an object, an exception will be raised.
7. Any triples whose subject cannot be identified as terminological knowledge, as ontology individuals, or for which (2) does not apply are **not imported**. No warning nor exception of any kind is raised for such triples.

Warning

Point (7) implies that using the default options, you can lose data that was originally in the source, **without** warnings nor errors to notify you about it. Keep reading to learn how to prevent it.

There are two keyword arguments that can be passed to the import and export functions to bypass these checks.

- `all_triples`: When set to `True`, no exceptions will be raised for points (2)*, (3), (4), (5), (6). Warnings will still be emitted.
- `all_statements`: When set to `True`, none of the points above apply. All RDF triples are imported, and **no information is lost**. No warnings are emitted.

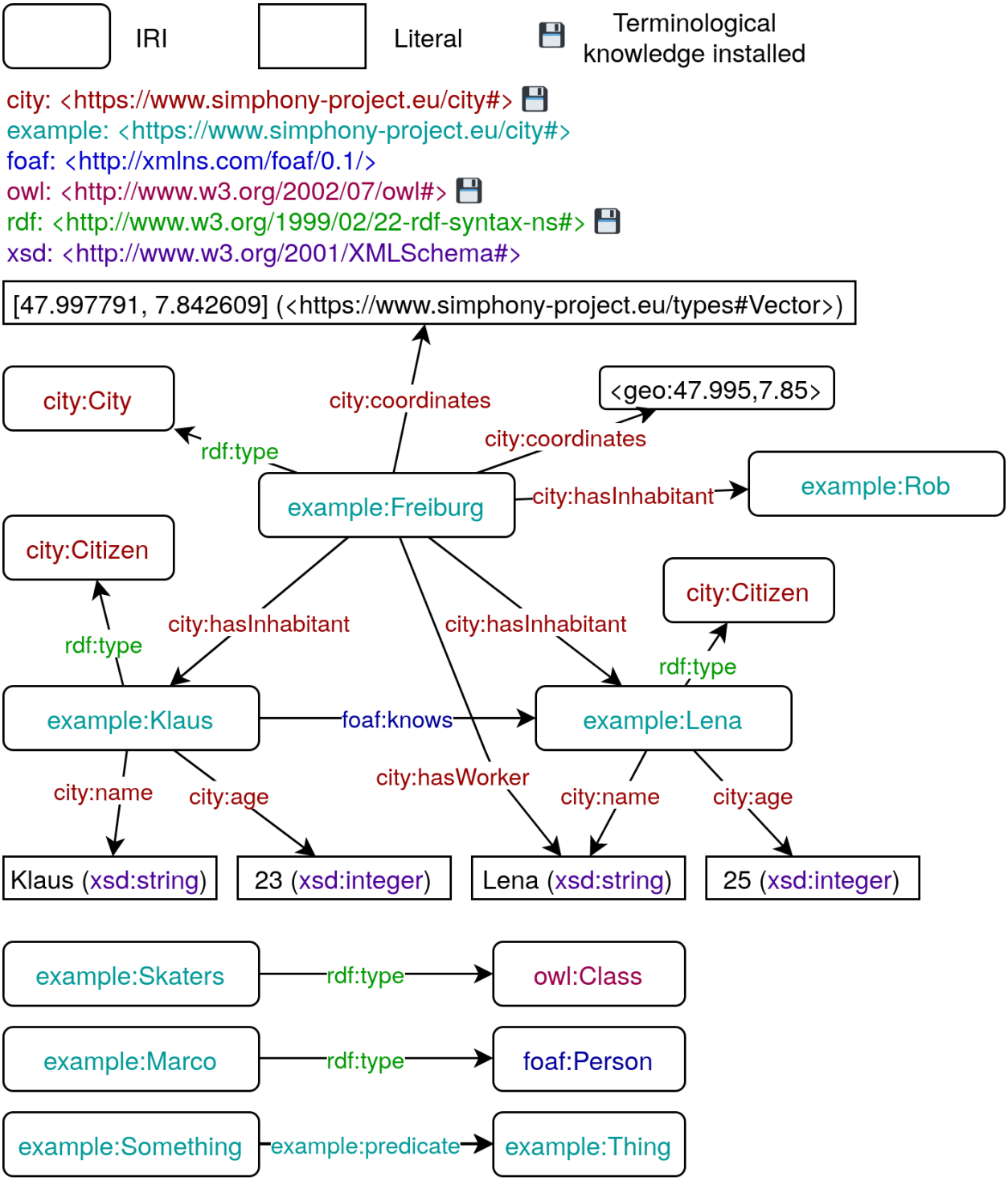
Note

* To be recognized as an ontology individual, at least one of the types of the subject need to be defined in the *installed ontologies*. If you use `all_triples=True` when none of the types are defined in the installed ontologies, then (7) applies to such subject (its information is lost).

Warning

Be careful when using `all_triples` and especially, when using `all_statements`. SimPhoNy's sessions have been designed to work only with ontology individuals. If you use `all_statements=True`, then also classes, relationships, annotations and attributes will be imported to the session, but as SimPhoNy is not ready to deal with this situation, this may lead to errors.

Below there is sample RDF graph that helps understanding all the cases.



1. The triple (`example:Skaters`, `rdf:type`, `owl:Class`) would trigger this case, because it defines terminolog-

ical knowledge.

2. The triple (`example:Marco`, `rdf:type`, `foaf:Person`) matches this case, because the namespace `foaf` is not installed.
3. The triple (`example:Klaus`, `foaf:knows`, `example:Lena`) raises an exception, because the namespace `foaf` is not installed.
4. The triple (`example:Freiburg`, `city:hasInhabitant`, `example:Rob`) triggers this case, because `example:Rob` does not have any type assigned and therefore, it is not identified as an ontology individual.
5. The triple (`example:Freiburg`, `city:hasWorker`, `Lena (xsd:string)`) matches this case, as "has worker" is a relationship, but the object is a literal.
6. The triple (`example:Freiburg`, `city:coordinates`, <`geo:47.995,7.85`>) raises an exception, because "coordinates" is an attribute, but the object is an IRI.
7. The triple (`example:Something`, `example:predicate`, `example:Thing`) fits into this case.

CHAPTER

EIGHT

WRAPPERS

8.1 Introduction

SimPhoNy Wrappers are pieces of software that seamlessly translate *assertional knowledge* to a form that is compatible with a specific simulation engine, database, data repository or file format. The way to interact with wrappers is through a *session object* that is connected to them. Therefore, before continuing, make sure that you have read the *previous section on sessions*.

A few wrappers are included with SimPhoNy, but generally, they must be installed separately. The included wrappers are:

- *SQLite wrapper*
- *SQLAlchemy wrapper*
- *Dataspace wrapper*
- *Remote wrapper*

After installation, wrappers are available under the `simphony_osp.wrappers` module.

Tip

The `simphony_osp.wrappers` module supports IPython autocompletion. When working on a Jupyter notebook, it is possible to get the installed wrappers as suggestions writing `from simphony_osp.wrappers import` and pressing TAB.

8.1.1 Wraper capabilities

Even though from the user's perspective all wrappers are used in the same way, their capabilities may vary with respect to the following points, and thus, have an influence on how the wrapper is used.

Persistence

Not all wrapper sessions persist the data that you pass to them. Wrappers that interact with databases, such as the included *SQLite wrapper* and *SQLAlchemy wrapper* will persist it in the database, but for example, most simulation wrappers do not persist the information anywhere. When using the latter, you are typically expected to load ontology individuals to the session in order to configure the simulation, run it, and then copy ontology individuals back from the simulation wrapper to somewhere else. The contents of the wrapper session are discarded after the session is closed.

Files

SimPhoNy allows to mix assertional knowledge with files in the same session. Files are represented by ontology individuals that belong to the class "File" from the `simphony` namespace (included with SimPhoNy). Head to the *assertional knowledge* section for an example on how to work with files.

However, this functionality only works in some wrapper sessions that support it. When transferring file individuals, raw files will **not** be transferred together with them to wrapper sessions that do not support files. Therefore, always be mindful when transferring file individuals to prevent data loss.

Tip

Think twice before deleting a "File" individual that has raw files attached to it. Do you have a copy in another session **that supports files**? If not, then by doing so you are deleting your raw file.

If you delete a file individual by mistake, remember that changes will not be applied until you commit them. Consider closing the session without committing the changes.

Simulation

SimPhoNy wrappers interact with different kinds of software. Wrapper sessions can have a method `compute` (more on it *later*) that serves to fire up the transformation of the data contained in them (e.g. run a simulation). However, such a method makes no sense, for example, for wrappers that interact with databases. Thus, not all wrapper sessions have this method defined.

Cache

Wrapper developers can choose to opt-in for a caching feature that is built-in in SimPhoNy. Imagine that you are operating a database wrapper that connects to a remote server. Every time you perform an action in SimPhoNy, data needs to be transferred to the database and back, introducing latency. The caching feature keeps a Least Recently Used (LRU) cache of ontology individuals so that in many situations this transfer can be omitted.

8.1.2 Operating wrapper sessions

As described on the *introduction to sessions*, sessions, and as a consequence also wrappers, work in a way similar to databases. To start using them, one first has to "open" or "connect" to them. After that, changes can be performed on the data they contain, but such changes are not made permanent until a "commit" is performed. When one finishes working with them, the connection should be "closed". Unconfirmed changes are lost when the connection is "closed".

Let's see how to manage a wrapper session using the SQLite wrapper as example.

[1]:
```
from simphony_osp.wrappers import SQLite
```

To initialize a session linked to a wrapper, call the imported object. Wrappers take two positional arguments: the `configuration_string` and the `create` argument.

The configuration string lets the wrapper know which resource to "open" or to "connect to". For the SQLite wrapper, it is the path of an SQLite database file. The `create` arguments can be set to `True` to ask the wrapper to create the resource specified by the configuration string if it does not already exist.

[2]:
```
sqlite = SQLite('database.db', True)
sqlite.clear()  # just in case you already ran this notebook
sqlite.locked = True  # was explained in the "introduction to sessions" section
sqlite
```

[2]:
```
<simphony_osp.session.session.Session at 0x7f25580ac370>
```

[3]:
```
len(sqlite)
```

[3]:
```
0
```

Note

Some wrappers may accept additional keyword arguments.

As you can see, the `sqlite` object that has been created is just a normal session that can store ontology individuals. The wrapper session is **automatically "opened" when it is created**.

[4]:
```
from simphony_osp.namespaces import city
from simphony_osp.tools import semantic2dot

with sqlite:
    freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])
    peter = city.Citizen(name="Peter", age=30)
    anne = city.Citizen(name="Anne", age=20)
    freiburg[city.hasInhabitant] += peter, anne

semantic2dot(sqlite)
```

[4]:

To confirm the set of changes that have been performed, it is necessary to "commit" them. Remember that **uncommitted changes are lost** after closing the session or the Python shell.

[5]:
```
sqlite.commit()
```

After the job is done, the session should be "closed" to free the resource that is being used. In the SQLite case, to close the database file.

[6]:
```
sqlite.close()
```

The city and its citizens have been successfully saved to the database file. If the wrapper is used to reopen the database, the saved individuals will be available in the resulting session object.

[7]:
```
from IPython.display import display

with SQLite('database.db', True) as sqlite:
    display(semantic2dot(sqlite))
```

Wrappers sessions that interact with simulation engines have an additional method, `compute`, that commits the uncommitted data, executes the simulation and updates the session to reflect the new status of the individuals involved. Depending on the wrapper, the `compute` method may accept keyword arguments. The *quickstart* tutorial demonstrates the use of the `compute` method with the SimLAMMPS wrapper, as well as how transfer of ontology individuals between different wrappers, which is done in exactly the same way as *between sessions*.

8.2 SQLite

Capability	Support
Persistence	✓
Files	
Simulation	
Cache	

The SQLite wrapper can store ontology individuals using several tables in an SQLite database file. It is actually just a special case of the more feature-rich *SQLAlchemy wrapper*. The SQLite wrapper is included with SimPhoNy and available under `simphony_osp.wrappers.SQLite`.

```
[1]: from simphony_osp.wrappers import SQLite
```

Configuration

The *configuration string* for the SQLite wrapper is the path to the database file to be used. To use the wrapper, just call it providing the *two arguments that SimPhoNy wrappers require*: the configuration string and the `create` argument.

```
[2]: sqlite = SQLite('database.db', True)
```

Tip

Keep in mind that the SQLite wrapper really just calls the *SQLAlchemy wrapper* adding the suffix `sqlite:///` to the given configuration string.

8.3 SQLAlchemy

Capability	Support
Persistence	✓
Files	
Simulation	
Cache	

The SQLAlchemy wrapper can store ontology individuals using several tables on any of the database backends suported by SQLAlchemy. It is based on the rdflib-sqlalchemy package. The wrapper is included with SimPhoNy and available under `simphony_osp.wrappers.SQLAlchemy`.

```
[1]: from simphony_osp.wrappers import SQLAlchemy
```

Configuration

The *configuration string* for the SQLAlchemy wrapper is an SQLAlchemy Database URL. To use the wrapper, just provide such an URL and the `create` argument.

```
[2]: sqlalchemy = SQLAlchemy('sqlite:///database.db', True)
```

Tip

Check the SQLAlchemy documentation to learn how to construct database URLs.

8.4 Dataspace

Capability	Support
Persistence	✓
Files	✓
Simulation	
Cache	

The Dataspace wrapper combines the functionality of the *SQLite* wrapper with support for files. Given a path on your computer, it creates an SQLite database file where the ontology individual data is stored, as well as a `files` subfolder where raw files associated with "File" individuals are stored.

Tip

Head to the *assertional knowledge* section for an example on how to work with files.

The Dataspace wrapper is included with SimPhoNy and located under `simphony_osp.wrappers.Dataspace`.

[1]:
```
from simphony_osp.wrappers import Dataspace
```

Configuration

The *configuration string* for the Dataspace wrapper is a path to a folder in your computer. To use the wrapper, just call it providing the *two arguments that SimPhoNy wrappers require*: the configuration string and the `create` argument.

[4]:
```
dataspace = Dataspace('dataspace', True)
```

8.5 Remote

Capability	Support
Persistence	✓*
Files	✓*
Simulation	✓*
Cache	✓

Note

* Capabilities are supported on the client-side, but the wrapper session running on the server needs to support them as well.

The Remote Wrapper is a special type of wrapper used to interact with a wrapper session that lives in a remote server. For example, a *Dataspace* wrapper session could be started on a remote server to let a single user store his/her data there. The Remote wrapper is included with SimPhoNy and available under `simphony_osp.wrappers.Remote`.

[1]:
```
from simphony_osp.wrappers import Remote
```

Configuration

The *configuration string* for the Remote wrapper is a URI pointing to the server that hosts the remote wrapper session.

[3]:
```
# needed to use the wrapper on a Jupyter notebook
import nest_asyncio
nest_asyncio.apply()

remote = Remote('ws://username:password@127.0.0.1:4008', True)
# If you are running this page on Binder, do not run this cell until the server is␣
↪running.
# Wait a few seconds after launching the server so that it has enough time to initialize.
```

When the remote wrapper session is closed, only the connection is closed. The server will keep its wrapper session open and wait for a user to connect again.

[4]:
```
remote.close()
```

8.5.1 Server setup

To quickly fire up a server, use the function `simphony_osp.tools.host`.

[2]:
```
import multiprocess as multiprocessing  # patched version of multiprocessing to work in␣
↪Binder, use normal multiprocessing in your machine

def launch_server():
    from simphony_osp.tools import host
    from simphony_osp.wrappers import Dataspace

    host(
        Dataspace,
        "dataspace",
        True,
        hostname="127.0.0.1",  # hostname to listen on
        port=4008,  # port to listen on
        username="username",
        password="password",
    )

multiprocessing.set_start_method("spawn")  # needed for the example to work in Binder
server = multiprocessing.Process(target=launch_server)
server.start()
```

Note

The server on this example works for just one user. SimPhoNy does not include a method to spawn a server that can be used by multiple users.

However, if you take a look at SimPhoNy's source code, you will realize that most of the pieces of the puzzle are already there, so it would be relatively simple to modify the `host` function to achieve such goal.

CHAPTER

NINE

VISUALIZATION

SimPhoNy includes two visualization tools:

- *semantic2dot* to visualize ontology entities, including both assertional and terminological knowledge,
- *pretty_print*, used to "visualize" ontology individuals as text.

9.1 semantic2dot

`semantic2dot` makes use of Graphviz and is located under `simphony_osp.tools.semantic2dot`. It can be used to quickly draw ontologies, as well as ontology individuals. In the representations created by the tool, each ontology entity is represented by a graph node. The relationships between ontology entities are the edges connecting them. The attributes values and classes that individuals belong to are written inside the nodes.

To **draw a namespace** from an installed ontology, import the namespace and pass the namespace object to `semantic2dot`. Several namespace objects can be drawn together in the same picture.

[3]:
```
!pico install city foaf
```

[2]:
```
from simphony_osp.namespaces import city, foaf
from simphony_osp.tools import semantic2dot
```

[4]:
```
semantic2dot(city, foaf)
```

[4]:

Tip

On most web browsers (including mobile ones), you can right-click the picture above and then click "open in new tab" to see the picture in its full size.

It is also possible to **draw the contents of a session**. To do so, pass the session object to `semantic2dot`.

[5]:
```
from simphony_osp.session import core_session

freiburg = city.City(name="Freiburg", coordinates=[47.997791, 7.842609])
peter = city.Citizen(name="Peter", age=30)
anne = city.Citizen(name="Anne", age=20)
freiburg[city.hasInhabitant] += peter, anne

semantic2dot(core_session)
```

[5]:

Another option is to **draw only specific ontology individuals**.

[6]:
```
semantic2dot(freiburg, anne)
```

[6]:

Any combination of such three types of object, any number of times can be provided.

[7]:
```
semantic2dot(foaf, city, core_session, freiburg, anne)
```

[7]:

A relationship can optionally be passed to `semantic2dot` so that it automatically calls the *find* method from the search module on all ontology individuals and includes the additional individuals in the picture.

[8]:
```
from simphony_osp.namespaces import owl

semantic2dot(freiburg, rel=owl.topObjectProperty)
```

[8]:

`semantic2dot` automatically renders its output as `svg` image files on Jupyter notebooks. However, it is possible to save the generated figures in `gv` (Graphviz) and `png` formats using the `render` method.

[10]:
```
semantic2dot(freiburg, rel=owl.topObjectProperty).render("./picture.gv")
```

After running the code above, two files, `picture.gv` and `picture.gv.png`, are generated in the current directory.

9.2 pretty_print

`pretty_print` is located under `simphony_osp.tools.pretty_print`. It generates a tree-like, text-based representation stemming from a given ontology individual, that includes the IRI, ontology classes and attributes of the involved individuals, as well as the relationships connecting them.

[11]:
```
from simphony_osp.tools import pretty_print

pretty_print(freiburg)
```

```
- Ontology individual:
  identifier: 48927171-fd87-42d7-9f40-be7a4f9b7334
  type: City (city)
  superclasses: City (city), Populated Place (city), Thing (owl), Geographical Place␣
↪(city)
  values: coordinates: [47.997791  7.842609]
          name: Freiburg
   |_Relationship has inhabitant (city):
     - Ontology individual of class Citizen
      . identifier: 7047ae55-f831-44bc-aa25-6f654c17cdcf
      . age: 20
      . name: Anne
     - Ontology individual of class Citizen
        identifier: c1816c6e-c441-408f-b0cc-d99641d4f4fc
        age: 30
        name: Peter
```

`pretty_print` recursively finds the ontology individuals that are connected to the given one using the *find* method from the search module. By default, all relationships are followed, but the relationships to follow can be restricted using the keyword argument `rel`. In the example below, the search has been restricted to several relationships that do not exist between `freiburg` and its citizens. Therefore, only `freiburg` is displayed as output.

[13]:
```
pretty_print(freiburg, rel=(city.hasWorker, city.hasMajor))
```

```
- Ontology individual:
  identifier: 48927171-fd87-42d7-9f40-be7a4f9b7334
  type: City (city)
  superclasses: City (city), Populated Place (city), Thing (owl), Geographical Place␣
↪(city)
  values: coordinates: [47.997791  7.842609]
          name: Freiburg
```

CHAPTER

TEN

WRAPPER DEVELOPMENT

SimPhoNy Wrappers are software components that transform data from the assertional-knowledge form to the data structures of other software and back. Wrappers are abstracted to users as *sessions*, which may be viewed as "boxes" where ontology individuals can be stored.

Sessions work in a way similar to databases. To start using them, one first has to "open" or "connect" to them. After that, changes can be made on the data they contain, but such changes are not permanent until a "commit" is performed. When one finishes working with them, the connection should be "closed". Unconfirmed changes are lost when the connection is "closed".

Therefore, developing a wrapper involves crafting:

- An abstraction of the concepts handled by the software as terminological knowledge, that can then be used to represent the information as assertional knowledge.
- A database-like interface that is used by SimPhoNy to communicate with the software.

For the latter, SimPhoNy defines the *Wrapper API*, that must be implemented by the developer.

10.1 `Wrapper` abstract class

The database-like interface used by SimPhoNy to communicate with the software, called *Wrapper API*, is defined by the `simphony_osp.development.Wrapper` abstract class. Objects belonging to the `Wrapper` class are indirectly controlled by the interactions between the user and session objects, as the diagram below shows.

UML object diagram showing the objects involved in the SimPhoNy wrapper mechanism that are relevant from a developer's perspective.

SimPhoNy makes use of the RDFLib library to handle RDF data. Thus, the session is in fact an interface to an RDFLib Graph. As the user interacts with the session, triples from the underlying graph are queried, added or removed. The library also provides a further abstraction, the store. Stores abstract triplestores, a kind of database that can store collections of RDF graphs in the form of triples. SimPhoNy implements a special kind of store that is designed to communicate with SimPhoNy wrapper objects, the final pieces of the chain.

The **wrapper object's interface is what the developer must implement** in order to make the software interoperable with SimPhoNy. As pointed out in the diagram, there are several RDF Graph objects, session objects and lists (of ontology individual objects) that are accessible from the wrapper object. Those offer several ways for the developer to retrieve the inputs from the user and translate them into inputs for the software, or conversely, reflect the outputs from the software into the user's session.

Perhaps the most important of all is the **base graph**. The base graph is an RDFLib Graph that is accessible from the wrapper object, and must be kept in sync with the software's data structures at all times, as it constitutes their **RDF representation**. The goal of the SimPhoNy Wrapper API is to facilitate this task to the developer.

Note: If an RDFLib store plug-in already exists for a specific software, then it can be trivially reused to implement a SimPhoNy wrapper. Just create a graph using the plug-in, and set it as the base graph for the SimPhoNy wrapper object.

10.2 API Overview

The flowchart below illustrates the **lifecycle** of a session connected to a wrapper object: from its creation to the moment it is closed.

A sequence of method calls is executed as a consequence of each possible action the user can take. Each sequence is represented using a different color, and the action that triggers it is written next to its accompanying arrow, that points to the first method call in the sequence.

Flowchart showing the catalogue of possible user actions and how they translate to calls to the methods of the wrapper class, that the wrapper developer must implement.

The `Wrapper` object is spawned when the user opens the session. The `open` and `populate` methods are then subsequently called in order to gain access to the resources needed by the software and pre-populate the session with ontology individuals if necessary. After that, the session is ready to be used. The user may then access or modify the assertional knowledge (triggering the optional, low-level RDF manipulation methods), access files associated to ontology individuals belonging to the *File* class (triggering the `load` method), add or change files, commit the changes or request the software to compute new results. When the user is done, the session is closed.

10.3 API Specification

This section describes the expected behavior of all methods and features from the Wrapper API.

10.3.1 Main methods

open

Secure access to the resources used by your software. For example:

- spawn simulation engine objects
- open a connection to a database
- open files that need to be accessed

`simphony_osp.development.Wrapper.`**`open`**(*self*, *configuration: str*, *create: bool = False*) → None

Open the data source that the wrapper interacts with.

You can expect calls to this method even when the data source is already accesible, therefore, an implementation similar to the one below is recommended.

```
>>> def open(self, configuration: str, create: bool = False):
>>>     if your_data_source_is_already_open:
>>>         return
>>>         # To improve the user experience you can check if the
```

(continues on next page)

(continued from previous page)

```
>>>         # configuration string leads to a resource different from
>>>         # the current one and raise an error informing the user.
>>>
>>>     # Connect to your data source...
>>>     # your_data_source_is_already_open is for now True.
```

If you are using a custom base graph, please set *self.base* = *your_graph* within this method. Otherwise, an empty base graph will be created instead.

Parameters

- **configuration** – Used to locate or configure the data source to be opened.
- **create** – Whether the data source should be created if it does not exist. When false, if the data source does not exist, you should raise an exception. When true, create an empty data source.

populate

Populate the base session so that it represents the initial state of the software. For example:

- given a path to a simulation settings file, populate the session with entities descibing the contents of the file
- populate the session with dataset individuals after connecting to a data repository

simphony_osp.development.Wrapper.**populate**(*self*) → None

Populate the base session so that it represents the data source.

This command is run after the data source is opened. Here you are expected to populate the base graph so that its information mimics the information on the data source, unless you are generating triples on the fly using the *triples* method. The default session inside this method is a session based on the base graph.

The base graph is available on *self.base*, and a session based on the base graph is available on *self.session* and *self.session_base*.

commit

Reflect user's changes on the session in the software's data structure. For example:

- configure a simulation's settings
- update an SQL table
- modify a file

simphony_osp.development.Wrapper.**commit**(*self*) → None

This method commits the changes made by the user.

Within this method, you have access to the following resources:

- *self.base*: The base graph (rw). You are not expected to modify it.
- *self.old_graph*: The old graph (ro).
- *self.new_graph*: The new graph (ro).
- *self.buffer*: The buffer of triples caught by *add* and *remove* (rw) that you now have to reflect on the data structures of your software.
- *self.session_base*: A session based on the base graph (rw). You are not expected to modify it.

- *self.session_old*: A session based on the old graph (ro).
- *self.session_new*: A session based on the new graph (ro).
- *self.session*: same as *self.session_new*.
- *self.added*: A list of added individuals (rw). You are not expected to modify the entities.
- *self.updated*: A list of updated individuals (rw). You are not expected to modify the entities.
- *self.deleted*: A list of deleted individuals (rw). You are not expected to modify the entities.

Before updating the data structures, check that the changes provided by the user do not leave them in a consistent state. This necessary because SimPhoNy cannot revert the changes you make to your data structures. Raise an AssertionError if the check fails.

Raises `AssertionError` – When the data provided by the user would produce an inconsistent or unpredictable state of the data structures.

The difference with respect to the `compute` method is that this method should not update the content's of the session itself.

`close`

Release the resources used by your software. For example:

- terminate the running process
- close the connection to a database
- close files that are not needed

`simphony_osp.development.Wrapper.close`(*self*) → None

Close the data source that the interface interacts with.

This method should NOT commit uncommitted changes.

This method should close the connection that was obtained in *open*, and free any locked up resources.

You can expect calls to this method even when the data source is already closed. Therefore, an implementation like the following is recommended.

```
>>> def close(self):
>>>     if your_data_source_is_already_closed:
>>>         return
>>>
>>>     # Close the connection to your data source.
>>>     # your_data_source_is_already_closed is for now True
```

`compute` *(optional)*

Perform a computation using the data currently present on the software's data structures and update the base session to reflect the changes afterwards. For example:

- run a simulation

`simphony_osp.development.Wrapper.compute`(*self*, ***kwargs: Union[str, int, float, bool, None, Iterable[Optional[Union[str, int, float, bool]]]]*) → None

Compute new information (e.g. run a simulation).

Compute the new information on the backend and reflect the changes on the base graph. The default session is the base session.

The base graph is available on *self.base*, and a session based on the base graph is available on *self.session* and *self.session_base*.

`__init__` *(optional)*

The `__init__` method allows the user to provide extra parameters in the form of JSON-serializable keyword arguments.

Note: This method does not appear on the flowchart for simplicity, but is executed before the `open` method.

`simphony_osp.development.Wrapper.__init__`(*self*, ***kwargs: Union[str, int, float, bool, None, Iterable[Optional[Union[str, int, float, bool]]]]*)

Initialize the wrapper.

The *__init__* method accepts JSON-serializable keyword arguments in order to let the user configure parameters of the wrapper that are not configurable via the ontology. For example, the type of solver used by a simulation engine.

Save such parameters to private attribute to use them later (e.g. in the *open* method).

Parameters **`kwargs`** – JSON-serializable keyword arguments that contain no nested JSON objects (check the type hint for this argument).

10.3.2 File manipulation methods

`load` *(optional)*

Receive an identifier of a file individual and retrieve the corresponding file.

`simphony_osp.development.Wrapper.load`(*self*, *key: str*) → BinaryIO

Retrieve a file.

Provide a file handle associated with the provided key.

Parameters **`key`** – Identifier of the individual associated with the file.

Returns File handle associated with the provided key.

`save` *(optional)*

Receive an identifier of a file individual and a file handle and save the contents somewhere.

`simphony_osp.development.Wrapper.save`(*self*, *key: str*, *file: BinaryIO*) → None

Save a file.

Read the bytestream offered as a file handle and save the contents somewhere, associating them with the provided key for later retrieval.

Parameters

- **`key`** – Identifier of the individual associated with the file.

- **file** – File (as a file-like object) to be saved.

delete *(optional)*

Receive the identifier of a file individual and delete the stored file.

simphony_osp.development.Wrapper.**delete**(*self*, *key: str*) → None

Delete a file.

Delete the file associated with the provided key.

Parameters **key** – Identifier of the individual associated with the file.

10.3.3 RDF manipulation methods

These methods operate at the RDF triple level. When a triple (or pattern) is added or removed, they can intercept the operation and decide whether the triple should go to the base graph or not when a commit operation is executed. The intercepted triples get stored in a buffer that is accessible during the commit operation.

They are useful when one wants to prevent storing the same information twice (in the base graph and in the software's data structure).

add *(optional)*

simphony_osp.development.Wrapper.**add**(*self*, *triple: Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]*) → bool

Inspect and control the addition of triples to the base graph.

Parameters **triple** – The triple being added.

Returns True when the triple should be added to the base graph. False when the triple should be caught, and therefore not added to the base graph. This triple will be latter available during commit on the buffer so that the changes that it introduces can be translated to the data structure.

remove *(optional)*

simphony_osp.development.Wrapper.**remove**(*self*, *pattern: Tuple[Optional[rdflib.term.Node], Optional[rdflib.term.Node], Optional[rdflib.term.Node]]*) → Iterator[Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]]

Inspect and control the removal of triples from the base graph.

Parameters **pattern** – The pattern being removed.

Returns An iterator with the triples that should be removed from the base graph. Any triples not included will not be removed, and will be available on the buffer during commit.

`triples` *(optional)*

simphony_osp.development.Wrapper.**triples**(*self*, *pattern: Tuple[Optional[rdflib.term.Node], Optional[rdflib.term.Node], Optional[rdflib.term.Node]]*) → Iterator[Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]]

Intercept a triple pattern query.

Can be used to produce triples that do not exist on the base graph on the fly.

Parameters `pattern` – The pattern being queried.

Returns An iterator yielding triples

10.3.4 Options

SimPhoNy includes several configurable features that facilitate wrapper development. Some of them are enabled by default, while others are disabled. Enabling or disabling them consists of configuring the value of an attribute in the implementation of the *`Wrapper` abstract class*. This subsection describes the attributes that can be configured.

`entity_tracking` *(default `True`)*

Entity tracking is actually what makes it possible to use the `added`, `updated` and `deleted` attributes of the *`Wrapper` abstract class* from within the *`commit` method*. When set to `True` (the default), whenever triples describing a new subject are added, SimPhoNy will add an ontology individual object to the `added` list. If the individual already existed and still exists, but just some triples for which it is the subject have been added or removed, then it is put in the `updated` list. When all triples describing the individual are deleted, an ontology individual object based on the session's status before the commit is put in the `deleted` list.

Disabling this feature speeds up *commits*.

`cache` *(default `False`)*

When set to `True`, the *`InterfaceDriver`* tries to retrieve triples from a cache managed by SimPhoNy instead of from the implementation of the *`Wrapper` abstract class.* or its *base graph*. Whenever there is a cache miss, SimPhoNy queries the `Wrapper` object for that specific triple pattern. In addition, if the pattern contains a subject, SimPhoNy will also query all information about such ontology individual as well as all "parent", "children" and "neighboring" individuals. For an individual, its *parents* are those other individuals who are connected to it through a relationship in which the parent is the subject and the aforementioned individual is the object. Similarly, the children are those individuals connected to it through a relationship where the aforementioned individual is the subject, and the children are the objects. The neighbors are the children of the parent individual.

This feature is useful whenever there is communication with a remote server involved, as it replaces large amounts of small exchanges of information with the server with small amounts of large exchanges of information.

10.4 Packaging template

This page is meant to offer a mid-level view on what SimPhoNy Wrappers are and how do they work. If you are interested in developing one, you may find a template for building and packaging a wrapper in the wrapper development repository.

CHAPTER

ELEVEN

OPERATIONS DEVELOPMENT

SimPhoNy operations are actions (written in Python) that can be executed on demand on any ontology individual belonging to the ontology class they are defined for.

Tip

File uploads and downloads in SimPhoNy are an example of SimPhoNy operations. Head to the assertional knowledge section to see them in action.

SimPhoNy operations are distributed as (or as part of) Python packages. Developing operations for an ontology class is fairly simple. To do so, import the *Operations abstract class* from the `simphony_osp.development` module, and create an implementation by subclassing it.

Then, define an `iri` attribute having as value the IRI of the ontology class that the operation should be associated to. Every public method (not starting with an underscore) defined on the implementation is automatically detected and assigned as an operation to said ontology class. The *ontology individual object* on which the operation is called is available as the private attribute `_individual` on every instance of the implementation. For a specific ontology individual, the implementation gets instantiated the first time that any operation defined on it is called by the user.

Finally, define an entry point (or many if implementing operations for several ontology classes) under the `simphony_osp.ontology.operations` group that points to your implementation of the `Operations` abstract class.

An example implementation of an operation that takes advantage of geopy to compute the distance between two points on Earth defined using the WGS84 Geo Positioning vocabulary is shown below.

```
"""Operations for classes from the WGS84 Geo Positioning vocabulary."""

from geopy import distance
from simphony_osp.namespaces import wgs84_pos
from simphony_osp.ontology import OntologyIndividual
from simphony_osp.ontology.operations import Operations


class Wgs84Pos(Operations):
    """Operations for the Point ontology class."""

    iri = wgs84_pos.Point.iri

    def distance(self, other: OntologyIndividual) -> float:
        """Compute the distance between two points on Earth.

        Args:
            other: Another point with respect to which the distance will be computed.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            The distance between this point and the given one in km.
        """
        lat_self = float(self._individual[wgs84_pos.latitude].one())
        long_self = float(self._individual[wgs84_pos.longitude].one())
        lat_other = float(other[wgs84_pos.latitude].one())
        long_other = float(other[wgs84_pos.longitude].one())
        return distance.geodesic(
            (lat_self, long_self),
            (lat_other, long_other),
            ellipsoid="WGS-84"
        ).km
```

Note

Remember that the implementation above is still not enough for the operation to work: the corresponding entry point for `Wgs84Pos` must have been defined and the `wgs84_pos` vocabulary needs to be installed using pico.

The operation can be used as follows.

[1]:
```
from simphony_osp.namespaces import wgs84_pos

location_freiburg = wgs84_pos.Point()
location_freiburg[wgs84_pos.latitude] = 47.997791
location_freiburg[wgs84_pos.longitude] = 7.842609

location_paris = wgs84_pos.Point()
location_paris[wgs84_pos.latitude] = 48.85333
location_paris[wgs84_pos.longitude] = 2.34885

location_freiburg.operations.distance(location_paris)
```

[1]:
```
417.4695920611045
```

CHAPTER

TWELVE

API REFERENCE

This document is for advanced users of SimPhoNy and defines all its public API details. This means that only when there is a breaking change in any of the methods listed on this page, the major version number of SimPhoNy will change accordingly, as prescribed by the Semantic Versioning Specification.

Do not rely on the stability of methods not listed on this page.

If the `__init__` method is not listed for a class, you are **not expected to instantiate such class by yourself**, and thus doing so is unsupported.

12.1 Ontology management

`simphony_osp.tools.pico.`**`install`**(**files: Union[pathlib.Path, str]*, *overwrite: bool = False*) → None

Install ontology packages.

Parameters

- **`files`** – Paths of ontology packages to install. Alternatively, identifiers of ontology packages that are bundled with SimPhoNy.
- **`overwrite`** – Whether to overwrite already installed ontology packages.

`simphony_osp.tools.pico.`**`uninstall`**(**identifiers: str*) → None

Uninstall ontology packages.

Parameters **`identifiers`** – Identifiers of the ontology packages to uninstall.

`simphony_osp.tools.pico.`**`packages`**() → Tuple[str]

Returns the identifiers of all installed packages.

`simphony_osp.tools.pico.`**`namespaces`**() → Tuple[*simphony_osp.ontology.OntologyNamespace*]

Returns namespace objects for all the installed namespaces.

12.2 Terminological- and assertional knowledge

`class` `simphony_osp.ontology.`**`OntologyNamespace`**(*iri: Union[str, URIRef]*, *ontology: Optional[*Session*] = None*, *name: Optional[str] = None*)

Bases: `object`

An ontology namespace.

Ontology namespace objects allow access to the terminological knowledge from the installed ontologies.

__contains__(*item: Union[*simphony_osp.ontology.OntologyEntity, *rdflib.term.Identifier]*) → bool

Check whether the given ontology entity is part of the namespace.

An ontology entity is considered to be part of a namespace if its IRI starts with the namespace IRI and if it is part of the session that the namespace is bound to. Identifiers are only required to start with the namespace IRI to be considered part of the namespace object. Blank nodes are never part of a namespace.

Parameters **item** – An ontology entity or identifier.

Returns Whether the given entity or identifier is part of the namespace. Blank nodes are never part of a namespace.

__eq__(*other:* simphony_osp.ontology.OntologyNamespace) → bool

Check whether the two namespace objects are equal.

Two namespace objects are considered to be equal when both have the same IRI and are bound to the same session.

Parameters **other** – The namespace object to compare with.

Returns Whether the given namespace object is equal.

Return type bool

__getattr__(*name: str*) → *simphony_osp.ontology.OntologyEntity*

Retrieve an entity by suffix or label.

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

Raises

- **AttributeError** – Unknown label or suffix.
- **AttributeError** – Multiple entities for the given label or suffix.

Returns An ontology entity with matching label or suffix.

__getitem__(*name: str*) → *simphony_osp.ontology.OntologyEntity*

Retrieve an entity by suffix or label.

Useful for entities whose labels or suffixes contain characters which are not compatible with the Python syntax rules.

Parameters **name** – The suffix or label of the ontology entity.

Raises

- **KeyError** – Unknown label or suffix.
- **KeyError** – Multiple entities for the given label or suffix.

Returns An ontology entity with matching label or suffix.

__iter__() → Iterator[*simphony_osp.ontology.OntologyEntity*]

Iterate over the ontology entities in the namespace.

__len__() → int

Return the number of entities in the namespace.

from_iri(*iri: Union[str, rdflib.term.URIRef]*) → *simphony_osp.ontology.OntologyEntity*

Get an ontology entity directly from its IRI.

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

Parameters **iri** – The iri of the ontology entity.

Returns The ontology entity.

Raises

- **KeyError** – When the IRI does not belong to the namespace.
- **ValueError** – When an invalid IRI is received.

from_label(*label: str*, *lang: Optional[str] = None*, *case_sensitive: bool = False*) → *simphony_osp.ontology.OntologyEntity*

Get an ontology entity from its label.

Parameters

- **label** – The label to match.
- **lang** – Optionally filter labels by a specific language.
- **case_sensitive** – Whether the match should be case-sensitive or not. The default setting is a case-insensitive lookup.

Raises

- **KeyError** – No label matches the given one.
- **KeyError** – More than one label matches the given one.

from_suffix(*suffix: str*) → *simphony_osp.ontology.OntologyEntity*

Get an ontology entity from its namespace suffix.

Parameters **suffix** – Suffix of the ontology entity.

Raises

- **KeyError** – When no entity with such suffix exists in the namespace.
- **ValueError** – When an invalid suffix is received (e.g. it contains a space).

get(*name: str*, *default: Optional[Any] = None*) → *simphony_osp.ontology.OntologyEntity*

Get ontology entities from the bounded session by suffix or label.

Parameters

- **name** – The label or suffix of the ontology entity.
- **default** – The entity to return if no entity with such label or suffix is found.

Raises

- **KeyError** – Unknown label or suffix (and no default given).
- **KeyError** – Multiple entities for the given label or suffix.

Returns The ontology entity with given label or suffix, or the default value.

property iri: rdflib.term.URIRef

The IRI of the namespace.

property name: Optional[str]

The name of the namespace.

For namespaces that have been imported from the *simphony_osp.namespaces* module, this name matches the alias given to the namespace in its ontology package.

class simphony_osp.ontology.**OntologyEntity**(*uid: UID*, *session: Optional[Union[*Session*, Container,* Wrapper*]] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: Optional[bool] = False*)

Abstract superclass of any entity in ontology entity.

__bool__()

Returns the boolean value of the entity, always true.

__eq__(*other:* simphony_osp.ontology.OntologyEntity) → bool

Check whether two entities are the same.

Two entities are considered equal when they have the same identifier and are stored in the same session.

Parameters **other** – The other entity.

Returns Whether the two entities are the same.

property direct_subclasses: FrozenSet[simphony_osp.ontology.entity.ONTOLOGY_ENTITY]

Get the direct subclasses of the entity.

Returns The direct subclasses of the entity.

property direct_superclasses: FrozenSet[simphony_osp.ontology.entity.ONTOLOGY_ENTITY]

Get the direct superclasses of the entity.

Returns The direct superclasses of the entity.

property identifier: rdflib.term.Identifier

Semantic web resource identifying the entity.

Usually an URIRef or BNode.

property iri: rdflib.term.URIRef

IRI of the Entity.

Raises **TypeError** – When the identifier of the ontology entity is not an IRI.

is_subclass_of(*other:* simphony_osp.ontology.OntologyEntity) → bool

Perform a subclass check.

Parameters **other** – The other entity.

Returns Whether self is a subclass of the other entity.

Return type bool

is_superclass_of(*other:* simphony_osp.ontology.OntologyEntity) → bool

Perform a superclass check.

Parameters **other** – The other ontology entity.

Returns Whether self is a superclass of the other other entity.

iter_labels(*lang: Optional[str] = None*, *return_prop: bool = False*, *return_literal: bool = True*) → Iterator[Union[rdflib.term.Literal, str, Tuple[str, rdflib.term.URIRef], Tuple[rdflib.term.Literal, rdflib.term.URIRef]]]

Returns all the available labels for this ontology entity.

Parameters

- **lang** – retrieve labels only in a specific language.

- **return_prop** – Whether to return the property that designates the label. When active, it is the second argument.
- **return_literal** – Whether to return a literal or a string with the label (the former contains the language, the latter not).

Returns An iterator yielding strings or literals; or tuples whose first element is a string or literal, and second element the property defining this label.

property label: Optional[str]

Get the preferred label of this entity, if it exists.

See the docstring for *label_literal* for more information on the definition of preferred label.

property label_lang: Optional[str]

Get the language of the main label of this entity.

See the docstring for *label_literal* for more information on the definition of main label.

property label_literal: Optional[rdflib.term.Literal]

Get the main label for this entity.

The labels are first sorted by the property defining them, then by their language, and then by their length.

Returns The first label in the resulting ordering is returned. If the entity has no label, then None is returned.

property namespace: Optional[*OntologyNamespace*]

Return the ontology namespace to which this entity is associated.

property session: *Session*

The session where the entity is stored.

property subclasses: FrozenSet[simphony_osp.ontology.entity.ONTOLOGY_ENTITY]

Get the subclasses of the entity.

Returns The subclasses of the entity

property superclasses: FrozenSet[simphony_osp.ontology.entity.ONTOLOGY_ENTITY]

Get the superclass of the entity.

Returns The superclasses of the entity.

property triples: Set[Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]]

Get the all the triples where the entity is the subject.

Triples from the underlying RDFLib graph where the entity is stored in which the entity's identifier is the subject.

class simphony_osp.ontology.**OntologyClass**(*uid: UID*, *session: Optional[Session] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*)

Bases: *simphony_osp.ontology.OntologyEntity*

A class defined in the ontology.

__call__(*session=None*, *iri: Optional[Union[rdflib.term.URIRef, str]] = None*, *identifier: Optional[Union[uuid.UUID, str, rdflib.term.Node, int, bytes]] = None*, *_force: bool = False*, ***kwargs*)

Create an OntologyIndividual object from this ontology class.

Parameters

- **identifier** – The identifier of the ontology individual. When set to a string, has the same effect as the keyword argument *iri*. When set to`None`, a new identifier with a random UUID is generated. When set to any of the other accepted types, the given value is used to generate the UUID of the identifier. Defaults to None.
- **iri** – The same as the identifier, but exclusively for IRI identifiers.
- **session** – The session that the ontology individual will be stored in. Defaults to *None* (the default session).

Raises **TypeError** – Error occurred during instantiation.

Returns The new ontology individual.

property attributes: Mapping[*simphony_osp.ontology.OntologyAttribute*, FrozenSet[Optional[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]]]

Get the attributes of this class.

The attributes that all instances of this class are expected to have. A class can have attributes because one of its superclasses (including itself) has a default value for an attribute, or because the axioms affecting the superclass explicitly state that the class has such an attribute.

property axioms: FrozenSet[Union[*simphony_osp.ontology.Restriction*, *simphony_osp.ontology.Composition*]]

Get all the axioms for the ontology class.

Axioms are OWL Restrictions and Compositions. Includes axioms inherited from its superclasses.

Returns Axioms for the ontology class.

property optional_attributes: FrozenSet[*simphony_osp.ontology.OntologyAttribute*]

Get the optional attributes of this class.

The optional attributes are the non-mandatory attributes (those not returned by the *attributes* property) that have the class defined as their domain, or any of its superclasses.

class simphony_osp.ontology.**Restriction**(*uid: UID*, *session: Optional[*Session*] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*)

Bases: *simphony_osp.ontology.OntologyEntity*

Restrictions on ontology classes.

property attribute: *simphony_osp.ontology.OntologyAttribute*

The attribute that the ATTRIBUTE_RESTRICTION acts on.

Raises **AttributeError** – Called on a RELATIONSHIP_RESTRICTION.

Returns The attribute.

property quantifier: *simphony_osp.ontology.QUANTIFIER*

Get the quantifier of the restriction.

Returns The quantifier of the restriction.

property relationship: *simphony_osp.ontology.OntologyRelationship*

The relationship that the RELATIONSHIP_RESTRICTION acts on.

Raises **AttributeError** – Called on an ATTRIBUTE_RESTRICTION.

Returns The relationship the restriction acts on.

property rtype: *simphony_osp.ontology.RTYPE*

Type of restriction.

Whether the restriction acts on attributes or relationships.

Returns The type of restriction.

Return type *RTYPE*

property target: Union[*OntologyClass*, URIRef]

The target ontology class or datatype.

Returns The target class or datatype.

class simphony_osp.ontology.**RESTRICTION_QUANTIFIER**(*value*)

Bases: enum.Enum

Quantifiers for restrictions.

EXACTLY: int = 3

MAX: int = 5

MIN: int = 4

ONLY: int = 2

SOME: int = 1

VALUE: int = 6

class simphony_osp.ontology.**RESTRICTION_TYPE**(*value*)

Bases: enum.Enum

Types of restrictions.

ATTRIBUTE_RESTRICTION = 1

RELATIONSHIP_RESTRICTION = 2

class simphony_osp.ontology.**Composition**(*uid: UID*, *session: Optional[*Session*] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*)

Bases: *simphony_osp.ontology.OntologyEntity*

Combinations of multiple classes using logical formulae.

property operands: Tuple[Union[*OntologyClass*, *Composition*, *Restriction*]]

The individual classes the formula is composed of.

Returns The operands.

property operator: *simphony_osp.ontology.OPERATOR*

The operator that connects the different classes in the formula.

Returns The operator Enum.

class simphony_osp.ontology.**COMPOSITION_OPERATOR**(*value*)

Bases: enum.Enum

Operations applicable to class definitions.

AND = 1

NOT = 3

OR = 2

class simphony_osp.ontology.**OntologyRelationship**(*uid: UID*, *session: Optional[Session] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*)

Bases: *simphony_osp.ontology.OntologyEntity*

A relationship defined in the ontology.

property **inverse: Optional[*simphony_osp.ontology.OntologyRelationship*]**

Get the inverse relationship if it exists.

class simphony_osp.ontology.**OntologyAttribute**(*uid: UID*, *session: Optional[Session] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*)

Bases: *simphony_osp.ontology.OntologyEntity*

An attribute defined in the ontology.

property **datatype: Optional[rdflib.term.URIRef]**

Get the data type of the attribute.

Returns IRI of the datatype.

Raises **NotImplementedError** – More than one data type associated with the attribute.

class simphony_osp.ontology.**OntologyAnnotation**(*uid: UID*, *session: Optional[Session] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*)

Bases: *simphony_osp.ontology.OntologyEntity*

An annotation property defined in the ontology.

class simphony_osp.ontology.**OntologyIndividual**(*uid: Optional[UID] = None*, *session: Optional[Session] = None*, *triples: Optional[Iterable[Triple]] = None*, *merge: bool = False*, *class_: Optional[OntologyClass] = None*, *attributes: Optional[Mapping[OntologyAttribute, Iterable[AttributeValue]]] = None*)

Bases: *simphony_osp.ontology.OntologyEntity*

An ontology individual.

__delitem__(*rel: Union[OntologyAnnotation, OntologyAttribute, OntologyRelationship]*)

Delete all objects attached through the given predicate.

Parameters **rel** – Either an ontology attribute, an ontology relationship or an ontology annotation (OWL datatype property, OWL object property, OWL annotation property). Alternatively a string, which will be resolved, using labels, to one of the classes described above.

__getattr__(*name: str*) → *simphony_osp.ontology.AttributeSet*

Retrieve the value of an attribute of the individual.

Parameters **name** – The label or suffix of the attribute.

Raises

- **AttributeError** – Unknown attribute label or suffix.

- **AttributeError** – Multiple attributes for the given label or suffix.

Returns The value of the attribute (a python object).

__getitem__(*rel: Union[*OntologyAnnotation, OntologyAttribute, OntologyRelationship*, str]*) → Union[*simphony_osp.ontology.AttributeSet*, *simphony_osp.ontology.RelationshipSet*, *simphony_osp.ontology.AnnotationSet*]

Retrieve linked individuals, attribute values or annotation values.

The subscripting syntax *individual[rel]* allows: - When *rel* is an OntologyAttribute, to obtain a set containing all

the values assigned to the specified attribute. Such set can be modified in-place to change the assigned values.

- When *rel* is an OntologyRelationship, to obtain a set containing all ontology individuals objects that are connected to *individual* through rel. Such set can be modified in-place to modify the existing connections.
- When *rel* is an OntologyAnnotation, to obtain a set containing all the annotation values assigned to the specified annotation property. Such set can be modified in-place to modify the existing connections.
- When *rel* is a string, the string is resolved to an OntologyAttribute, OntologyRelationship or OntologyAnnotation with a matching label, and then one of the cases above applies.

The reason why a set is returned and not a list, or any other container allowing repeated elements, is that the underlying RDF graph does not accept duplicate statements.

Parameters **rel** – An ontology attribute, an ontology relationship or an ontology annotation (OWL datatype property, OWL object property, OWL annotation property). Alternatively a string, which will be resolved, using labels, to one of the classes described above.

Raises

- **KeyError** – Unknown attribute, relationship or annotation label or suffix.
- **KeyError** – Multiple attributes, relationships or annotations found for the given label or suffix.
- **TypeError** – Trying to use something that is neither an OntologyAttribute, an OntologyRelationship, an OntologyAnnotation or a string as index.

__setattr__(*name: str*, *value: Optional[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, Set[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]]]*) → None

Set the value(s) of an attribute.

Parameters

- **name** – The label or suffix of the attribute.
- **value** – The new value(s).

Raises

- **AttributeError** – Unknown attribute label or suffix.
- **AttributeError** – Multiple attributes for the given label or suffix.

__setitem__(*rel: Union[OntologyAnnotation, OntologyAttribute, OntologyRelationship, str]*, *values: Optional[Union[OntologyIndividual, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, OntologyAnnotation, OntologyAttribute, OntologyClass, OntologyRelationship, rdflib.term.URIRef, rdflib.term.Literal, Iterable[Union[OntologyIndividual, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, OntologyAnnotation, OntologyAttribute, OntologyClass, OntologyRelationship, rdflib.term.URIRef, rdflib.term.Literal]]]]*) → None

Manages object, data and annotation properties.

The subscripting syntax `individual[rel] = ` allows,

- When *rel* is an OntologyRelationship, to replace the list of ontology individuals that are connected to *individual* through rel.
- When *rel* is an OntologyAttribute, to replace the values of such attribute.
- When *rel* is an OntologyAnnotation, to replace the annotation values of such annotation property.
- When *rel* is a string, the string is resolved to an OntologyAttribute, OntologyRelationship or OntologyAnnotation with a matching label, and then one of the cases above applies.

This function only accepts hashable objects as input, as the underlying RDF graph does not accept duplicate statements.

Parameters

- **rel** – Either an ontology attribute, an ontology relationship or an ontology annotation (OWL datatype property, OWL object property, OWL annotation property). Alternatively a string, which will be resolved, using labels, to one of the classes described above.
- **values** – Either a single element compatible with the OWL standard (this includes ontology individuals) or a set of such elements.

Raises

- **KeyError** – Unknown attribute, relationship or annotation label or suffix.
- **KeyError** – Multiple attributes, relationships or annotations found for the given label or suffix.
- **TypeError** – Trying to assign attributes using an object property, trying to assign ontology individuals using a data property, trying to use something that is neither an OntologyAttribute, an OntologyRelationship, an OntologyAnnotation nor a string as index.

property attributes: Mapping[*simphony_osp.ontology.OntologyAttribute*, FrozenSet[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]]

Get the attributes of this individual as a dictionary.

property classes: FrozenSet[*simphony_osp.ontology.OntologyClass*]

Get the ontology classes of this ontology individual.

This property is writable. The classes that an ontology individual belongs to can be changed writing the desired values to this property.

Returns A set with all the ontology classes of the ontology individual. When the individual has no classes, the set is empty.

connect(**individuals: Union[simphony_osp.ontology.OntologyIndividual, rdflib.term.Identifier, str], rel: Union[simphony_osp.ontology.OntologyRelationship, rdflib.term.Identifier]*) → None

Connect ontology individuals to other ontology individuals.

Parameters

- **individuals** – The individuals to be connected. Their identifiers may also be used.
- **rel** – The relationship between the objects.

Raises

- **TypeError** – Objects that are not ontology individuals, identifiers or strings provided as positional arguments.
- **TypeError** – Object that is not an ontology relationship or the identifier of an ontology relationship passed as keyword argument *rel*.
- **RuntimeError** – Ontology individuals that belong to a different session provided.

disconnect(**individuals: Union[simphony_osp.ontology.OntologyIndividual, rdflib.term.Identifier, str], rel: Union[simphony_osp.ontology.OntologyRelationship, rdflib.term.Identifier] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty'), oclass: Optional[simphony_osp.ontology.OntologyClass] = None*) → None

Disconnect ontology individuals from this one.

Parameters

- **individuals** – Specify the individuals to disconnect. When no individuals are specified, all connected individuals are considered.
- **rel** – Only remove individuals which are connected by subclass of the given relationship. Defaults to OWL.topObjectProperty (any relationship).
- **oclass** – Only remove elements which are a subclass of the given ontology class. Defaults to None (no filter).

Raises

- **TypeError** – Objects that are not ontology individuals, identifiers or strings provided as positional arguments.
- **TypeError** – Object that is not an ontology relationship or the identifier of an ontology relationship passed as keyword argument *rel*.
- **TypeError** – Object that is not an ontology class passed as keyword argument *oclass*.
- **RuntimeError** – Ontology individuals that belong to a different session provided.

get(**individuals: Union[simphony_osp.ontology.OntologyIndividual, rdflib.term.Identifier, str], rel: Union[simphony_osp.ontology.OntologyRelationship, rdflib.term.Identifier] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty'), oclass: Optional[simphony_osp.ontology.OntologyClass] = None, return_rel: bool = False*) → Union[*simphony_osp.ontology.RelationshipSet*, *simphony_osp.ontology.OntologyIndividual*, None, Tuple[Optional[*simphony_osp.ontology.OntologyIndividual*], ...], Tuple[Tuple[*simphony_osp.ontology.OntologyIndividual*, *simphony_osp.ontology.OntologyRelationship*]]]

Return the connected individuals.

The structure of the output can vary depending on the form used for the call. See the "Returns:" section of this docstring for more details on this.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Parameters

- **individuals** – Restrict the elements to be returned to a certain subset of the connected elements.
- **rel** – Only return individuals which are connected by a subclass of the given relationship. Defaults to OWL.topObjectProperty (any relationship).
- **oclass** – Only yield individuals which are a subclass of the given ontology class. Defaults to None (no filter).
- **return_rel** – Whether to return the connecting relationship. Defaults to False.

Returns

The result of the call is a set-like object. This corresponds to the calls *get()*, *get(rel=___)*, *get(oclass=___)*, *get(rel=___, oclass=___)*, with the parameter *return_rel* unset or set to False.

Calls with **individuals* (Optional[OntologyIndividual],

Tuple[Optional["OntologyIndividual"], ...]):

The position of each element in the result is determined by the position of the corresponding identifier/individual in the given list of identifiers/individuals. In this case, the result can contain *None* values if a given identifier/individual is not connected to this individual, or if it does not satisfy the class filter. When only one individual or identifier is specified, a single object is returned instead of a Tuple. This description corresponds to the calls *get(*individuals)*, *get(*individuals, rel=___)*, *get(*individuals, rel=___, oclass=___)*, with the parameter *return_rel* unset or set to False.

Calls with *return_rel=True* (Tuple[

Tuple[OntologyIndividual, OntologyRelationship]]):

The dependence of the order of the elements is maintained for the calls with **individuals*, a non-deterministic order is used for the calls without **individuals*. No *None* values are contained in the result (such identifiers or individuals are simply skipped). Moreover, the elements returned are now pairs of individuals and the relationship connecting this individual to such ones. This description corresponds to any call of the form *get(..., return_rel=True)*.

Return type Calls without **individuals* (RelationshipSet)

Raises

- **TypeError** – Objects that are not ontology individuals, identifiers or strings provided as positional arguments.
- **TypeError** – Object that is not an ontology relationship or the identifier of an ontology relationship passed as keyword argument *rel*.
- **TypeError** – Object that is not an ontology class passed as keyword argument *oclass*.
- **RuntimeError** – Ontology individuals that belong to a different session provided.

is_a(*ontology_class:* simphony_osp.ontology.OntologyClass) → bool

Check if the individual is an instance of the given ontology class.

Parameters **ontology_class** – The ontology class to test against.

Returns Whether the ontology individual is an instance of such ontology class.

iter(**individuals: Union[simphony_osp.ontology.OntologyIndividual, rdflib.term.Identifier, str], rel: Union[simphony_osp.ontology.OntologyRelationship, rdflib.term.Identifier] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty'), oclass: Optional[simphony_osp.ontology.OntologyClass] = None, return_rel: bool = False*) → Union[Iterator[*simphony_osp.ontology.OntologyIndividual*], Iterator[Optional[*simphony_osp.ontology.OntologyIndividual*]], Iterator[Tuple[*simphony_osp.ontology.OntologyIndividual*, *simphony_osp.ontology.OntologyRelationship*]]]

Iterate over the connected individuals.

The structure of the output can vary depending on the form used for the call. See the "Returns:" section of this docstring for more details on this.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Parameters

- **individuals** – Restrict the elements to be returned to a certain subset of the connected elements.
- **rel** – Only yield individuals which are connected by a subclass of the given relationship. Defaults to OWL.topObjectProperty (any relationship).
- **oclass** – Only yield individuals which are a subclass of the given ontology class. Defaults to None (no filter).
- **return_rel** – Whether to yield the connecting relationship. Defaults to False.

Returns

The position of each element in the result is non-deterministic. This corresponds to the calls *iter()*, *iter(rel=___)*, *iter(oclass=___)*, *iter(rel=___, oclass=___)*, with the parameter *return_rel* unset or set to False.

Calls with **individuals* (Iterator[Optional[

OntologyIndividual]]):

The position of each element in the result is determined by the position of the corresponding identifier/individual in the given list of identifiers/individuals. In this case, the result can contain *None* values if a given identifier/individual is not connected to this individual, or if it does not satisfy the class filter. This description corresponds to the calls *iter(*individuals)*, *iter(*individuals, rel=___)*, *iter(*individuals, rel=___, oclass=`___)`*.

Calls with *return_rel=True* (Iterator[

Tuple[OntologyIndividual, OntologyRelationship]]):

The dependence of the order of the elements is maintained for the calls with **individuals*. No *None* values are contained in the result (such identifiers or individuals are simply skipped). Moreover, the elements returned are now pairs of individualsand the relationship connecting this individual to such ones. This description corresponds to any call of the form *iter(..., return_rel=True)*.

Return type Calls without **individuals* (Iterator[OntologyIndividual])

Raises

- **TypeError** – Objects that are not ontology individuals, identifiers or strings provided as positional arguments.

- **TypeError** – Object that is not an ontology relationship or the identifier of an ontology relationship passed as keyword argument *rel*.
- **TypeError** – Object that is not an ontology class passed as keyword argument *oclass*.
- **RuntimeError** – Ontology individuals that belong to a different session provided.

property operations: simphony_osp.ontology.operations.operations.OperationsNamespace

Access operations specific this individual's class.

class simphony_osp.ontology.**RelationshipSet**(*relationship: Optional[simphony_osp.ontology.OntologyRelationship]*, *individual:* simphony_osp.ontology.OntologyIndividual, *oclass: Optional[simphony_osp.ontology.OntologyClass]* = *None*, *inverse: bool* = *False*, *uids: Optional[Iterable[simphony_osp.utils.datatypes.UID]]* = *None*)

A set interface to an ontology individual's relationships.

This class looks like and acts like the standard *set*, but it is an interface to the methods from *OntologyIndividual* that manage the relationships.

__and__(*other: set*) → set

Return self&other.

__contains__(*item:* simphony_osp.ontology.OntologyIndividual) → bool

Check if an individual is connected via set's relationship.

__ge__(*other: set*) → bool

Return self>=other.

__gt__(*other: set*) → bool

Return self>other.

__iadd__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self+=other (equivalent to self|=other).

__iand__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self&=other.

Should perform the intersection on the underlying data structure.

__invert__() → *simphony_osp.ontology.RelationshipSet*

Get the inverse RelationshipSet.

__ior__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self|=other.

Should perform the union on the underlying data structure.

__isub__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self-=other.

Based on *difference_update*.

__iter__() → Iterator[*simphony_osp.ontology.OntologyIndividual*]

Iterate over individuals assigned to *self._predicates*.

Note: no class filter.

Returns The mentioned underlying set.

__ixor__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self^=other.

Should perform the XOR operation on the underlying data structure.

__le__(*other: set*) → bool

Return self<=other.

__len__() → int

Return len(self).

__lt__(*other: set*) → bool

Return self<other.

__ne__(*other: set*) → bool

Return self!=other.

__or__(*other: set*) → set

Return self|other.

__radd__(*other: set*) → set

Return other&self.

__ror__(*other: set*) → set

Return other|self.

__rsub__(*other: set*) → set

Return value-self.

__rxor__(*other: set*) → set

Return value^self.

__xor__(*other: set*) → set

Return self^other.

add(*other: Any*) → None

Add an element to a set.

This has no effect if the element is already present.

all() → simphony_osp.ontology.individual.ObjectSet

Return all elements from the set.

Returns All elements from the set, namely the set itself.

any() → Optional[Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]]

Return any element of the set.

Returns Any element from the set if the set is not empty, else None.

clear() → None

Remove all elements from this set.

copy() → set

Return a shallow copy of a set.

difference(*other: Iterable*) → set

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

difference_update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Remove all elements of another set from this set.

discard(*other: Any*) → None

Remove an element from a set if it is a member.

If the element is not a member, do nothing.

property individual: ***simphony_osp.ontology.OntologyIndividual***

Ontology individual that this set refers to.

intersection(*other: set*) → set

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

intersection_update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Update the set with the intersection of itself and another.

property inverse: ***simphony_osp.ontology.RelationshipSet***

Get the inverse RelationshipSet.

Returns a RelationshipSet that works in the inverse direction: the ontology individuals displayed are the ones which are the subject of the relationship.

isdisjoint(*other: set*) → bool

Return True if two sets have a null intersection.

issubset(*other: set*) → bool

Report whether another set contains this set.

issuperset(*other: set*) → bool

Report whether this set contains another set.

one() → Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]

Return one element.

Return one element if the set contains one element, else raise an exception.

Returns The only element contained in the set.

Raises

- ***ResultEmptyError*** – No elements in the set.
- ***MultipleResultsError*** – More than one element in the set.

pop() → Any

Remove and return an arbitrary set element.

Raises KeyError if the set is empty.

property predicate: Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyRelationship*]

Predicate that this set refers to.

remove(*other: Any*) → None

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference(*other: set*) → set

Return the symmetric difference of two sets as a new set.

symmetric_difference_update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Update with the symmetric difference of it and another.

union(*other: set*) → set

Return the union of sets as a new set.

update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Update the set with the union of itself and other.

class simphony_osp.ontology.**AttributeSet**(*attribute: Optional[*simphony_osp.ontology.OntologyAttribute*]*, *individual:* simphony_osp.ontology.OntologyIndividual)

A set interface to an ontology individual's attributes.

This class looks like and acts like the standard *set*, but it is an interface to the methods from *OntologyIndividual* that manage the attributes.

__and__(*other: set*) → set

Return self&other.

__contains__(*item: Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]*) → bool

Check whether a value is assigned to the set's attribute.

__ge__(*other: set*) → bool

Return self>=other.

__gt__(*other: set*) → bool

Return self>other.

__iadd__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self+=other (equivalent to self|=other).

__iand__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self&=other.

Should perform the intersection on the underlying data structure.

__ior__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self|=other.

Should perform the union on the underlying data structure.

__isub__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self-=other.

Based on *difference_update*.

__iter__() → Iterator[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]

The values assigned to the referred predicates.

Such predicates are the main attribute and its subclasses.

Returns The mentioned values.

__ixor__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self^=other.

Should perform the XOR operation on the underlying data structure.

__le__(*other: set*) → bool

Return self<=other.

__len__() → int

Return len(self).

__lt__(*other: set*) → bool

Return self<other.

__ne__(*other: set*) → bool

Return self!=other.

__or__(*other: set*) → set

Return self|other.

__radd__(*other: set*) → set

Return other&self.

__ror__(*other: set*) → set

Return other|self.

__rsub__(*other: set*) → set

Return value-self.

__rxor__(*other: set*) → set

Return value^self.

__xor__(*other: set*) → set

Return self^other.

add(*other: Any*) → None

Add an element to a set.

This has no effect if the element is already present.

all() → simphony_osp.ontology.individual.ObjectSet

Return all elements from the set.

Returns All elements from the set, namely the set itself.

any() → Optional[Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]]

Return any element of the set.

Returns Any element from the set if the set is not empty, else None.

clear() → None

Remove all elements from this set.

copy() → set

Return a shallow copy of a set.

difference(*other: Iterable*) → set

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

difference_update(*other: Iterable[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]*) → None

Remove all elements of another set from this set.

discard(*other: Any*) → None

Remove an element from a set if it is a member.

If the element is not a member, do nothing.

property individual: *simphony_osp.ontology.OntologyIndividual*

Ontology individual that this set refers to.

intersection(*other: set*) → set

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

intersection_update(*other: Iterable[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]*) → None

Update the set with the intersection of itself and another.

isdisjoint(*other: set*) → bool

Return True if two sets have a null intersection.

issubset(*other: set*) → bool

Report whether another set contains this set.

issuperset(*other: set*) → bool

Report whether this set contains another set.

one() → Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]

Return one element.

Return one element if the set contains one element, else raise an exception.

Returns The only element contained in the set.

Raises

- *ResultEmptyError* – No elements in the set.
- *MultipleResultsError* – More than one element in the set.

pop() → Any

Remove and return an arbitrary set element.

Raises KeyError if the set is empty.

property predicate: Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyRelationship*]

Predicate that this set refers to.

remove(*other: Any*) → None

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference(*other: set*) → set

Return the symmetric difference of two sets as a new set.

symmetric_difference_update(*other: Set[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]*) → None

Update set with the symmetric difference of it and another.

union(*other: set*) → set

Return the union of sets as a new set.

update(*other: Iterable[Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]]*) → None

Update the set with the union of itself and others.

class simphony_osp.ontology.**AnnotationSet**(*annotation: Optional[*simphony_osp.ontology.OntologyAnnotation*]*, *individual:* simphony_osp.ontology.OntologyIndividual)

A set interface to an ontology individual's annotations.

This class looks like and acts like the standard *set*, but it is an interface to the methods from *OntologyIndividual* that manage the annotations.

__and__(*other: set*) → set

Return self&other.

__contains__(*item*) → bool

Determine whether the individual is annotated with an item.

__ge__(*other: set*) → bool

Return self>=other.

__gt__(*other: set*) → bool

Return self>other.

__iadd__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self+=other (equivalent to self|=other).

__iand__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self&=other.

Should perform the intersection on the underlying data structure.

__ior__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self|=other.

Should perform the union on the underlying data structure.

__isub__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self-=other.

Based on *difference_update*.

__iter__() → Iterator[Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]]

Iterate over annotations linked to the individual.

__ixor__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet

Return self^=other.

Should perform the XOR operation on the underlying data structure.

__le__(*other: set*) → bool

Return self<=other.

__len__() → int

Return len(self).

__lt__(*other: set*) → bool

Return self<other.

__ne__(*other: set*) → bool

Return self!=other.

__or__(*other: set*) → set

Return self|other.

__radd__(*other: set*) → set

Return other&self.

__ror__(*other: set*) → set

Return other|self.

__rsub__(*other: set*) → set

Return value-self.

__rxor__(*other: set*) → set

Return value^self.

__xor__(*other: set*) → set

Return self^other.

add(*other: Any*) → None

Add an element to a set.

This has no effect if the element is already present.

all() → simphony_osp.ontology.individual.ObjectSet

Return all elements from the set.

Returns All elements from the set, namely the set itself.

any() → Optional[Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]]

Return any element of the set.

Returns Any element from the set if the set is not empty, else None.

clear() → None

Remove all elements from this set.

copy() → set

Return a shallow copy of a set.

difference(*other: Iterable*) → set

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

difference_update(*other: Iterable[Any]*) → None

Return self-=other.

discard(*other: Any*) → None

Remove an element from a set if it is a member.

If the element is not a member, do nothing.

property individual: *simphony_osp.ontology.OntologyIndividual*

Ontology individual that this set refers to.

intersection(*other: set*) → set

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

intersection_update(*other: Iterable[Union[*OntologyIndividual*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID,* OntologyAnnotation*,* OntologyAttribute*,* OntologyClass*,* OntologyRelationship*, rdflib.term.URIRef, rdflib.term.Literal]]*) → None

Update the set with the intersection of itself and another.

isdisjoint(*other: set*) → bool

Return True if two sets have a null intersection.

issubset(*other: set*) → bool

Report whether another set contains this set.

issuperset(*other: set*) → bool

Report whether this set contains another set.

one() → Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]

Return one element.

Return one element if the set contains one element, else raise an exception.

Returns The only element contained in the set.

Raises

- *ResultEmptyError* – No elements in the set.
- *MultipleResultsError* – More than one element in the set.

pop() → Any

Remove and return an arbitrary set element.

Raises KeyError if the set is empty.

property predicate: Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyRelationship*]

Predicate that this set refers to.

remove(*other: Any*) → None

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference(*other: set*) → set

Return the symmetric difference of two sets as a new set.

symmetric_difference_update(*other: Iterable[Union[*OntologyIndividual*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID,* OntologyAnnotation*,* OntologyAttribute*,* OntologyClass*,* OntologyRelationship*, rdflib.term.URIRef, rdflib.term.Literal]]*) → None

Return self^=other.

union(*other: set*) → set

Return the union of sets as a new set.

update(*other: Iterable[Union[*OntologyIndividual*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID,* OntologyAnnotation*,* OntologyAttribute*,* OntologyClass*,* OntologyRelationship*, rdflib.term.URIRef, rdflib.term.Literal]]*) → None

Update the set with the union of itself and other.

class simphony_osp.ontology.**ResultEmptyError**

Bases: Exception

The result is unexpectedly empty.

class simphony_osp.ontology.**MultipleResultsError**

Bases: Exception

Only a single result is expected, but there were multiple.

12.3 Sessions and wrappers

class simphony_osp.session.**Session**(*base: Optional[Graph] = None*, *driver: Optional[InterfaceDriver] = None*, *ontology: Optional[Union[Session, bool]] = None*, *identifier: Optional[str] = None*, *namespaces: Dict[str, URIRef] = None*, *from_parser: Optional[OntologyParser] = None*)

'Box' that stores ontology individuals.

__contains__(*item:* simphony_osp.ontology.OntologyEntity)

Check whether an ontology entity is stored on the session.

__enter__()

Sets the session as the default session.

__exit__(*exc_type*, *exc_val*, *exc_tb*)

Restores the previous default session.

__init__(*base: Optional[Graph] = None*, *driver: Optional[InterfaceDriver] = None*, *ontology: Optional[Union[Session, bool]] = None*, *identifier: Optional[str] = None*, *namespaces: Dict[str, URIRef] = None*, *from_parser: Optional[OntologyParser] = None*)

Initializes the session.

The keyword arguments are used internally by SimPhoNy and are not meant to be set manually.

__iter__() → Iterator[*simphony_osp.ontology.OntologyEntity*]

Iterate over all the ontology entities in the session.

Be careful when using this operation, as it can be computationally very expensive.

__len__() → int

Return the number of ontology entities within the session.

add(**individuals: Union[*simphony_osp.ontology.OntologyIndividual, *Iterable[*simphony_osp.ontology.OntologyIndividual*]]*, *merge: bool = False*, *exists_ok: bool = False*, *all_triples: bool = False*) → Union[*simphony_osp.ontology.OntologyIndividual*, FrozenSet[*simphony_osp.ontology.OntologyIndividual*]]

Copies ontology individuals to the session.

Parameters

- **individuals** – Ontology individuals to add to this session.
- **merge** – Whether to merge individuals with existing ones if their identifiers match (read the SimPhoNy documentation for more details).
- **exists_ok** – Merge or overwrite individuals when they already exist in the session rather than raising an exception.
- **all_triples** – When an individual is added to the session, SimPhoNy only copies the details that are relevant from an ontological point of view: the individual's attributes, the classes it belongs to, and its connections to other ontology individuals that are also being copied at the same time.

  However, in some cases, it is necessary to keep all the information about the individual, even if it cannot be understood by SimPhoNy. Set this option to *True* to copy all RDF statements describing the individual, that is, all RDF statements where the individual is the subject.

One example of a situation where this option is useful is when the individual is attached through an object property to another one which is not properly defined (i.e. has no type assigned). This situation commonly arises when using the *dcat:accessURL* object property.

Returns The new copies of the individuals.

Raises

- **RuntimeError** – The individual being added has an identifier that
- **matches the identifier of an individual that already exists in the** –
- **session.** –

clear(*force: bool = False*)

Clear all the data stored in the session.

Parameters **force** – Try to clear read-only sessions too.

close() → None

Close the connection to the session's backend.

Sessions are an interface to a graph linked to an RDFLib store (a backend). If the session will not be used anymore, then it makes sense to close the connection to such backend to free resources.

commit() → None

Commit pending changes to the session's graph.

compute(***kwargs*) → None

Run simulations on supported graph stores.

delete(**entities: Union[*simphony_osp.ontology.OntologyEntity*, rdflib.term.Identifier, Iterable[Union[*simphony_osp.ontology.OntologyEntity*, rdflib.term.Identifier]]]*)

Remove ontology individuals from the session.

Parameters **entities** – Ontology individuals to remove from the session. It is also possible to just provide their identifiers.

Raises **ValueError** – When at least one of the given ontology individuals is not contained in the session.

from_label(*label: str*, *lang: Optional[str] = None*, *case_sensitive: bool = False*) → FrozenSet[*simphony_osp.ontology.OntologyEntity*]

Get an ontology entity from its label.

Parameters

- **label** – The label of the ontology entity.
- **lang** – The language of the label.
- **case_sensitive** – when false, look for similar labels with different capitalization.

Raises **KeyError** – Unknown label.

Returns The ontology entity.

get(**individuals: Union[*simphony_osp.ontology.OntologyIndividual*, rdflib.term.Identifier, str]*, *oclass: Optional[*simphony_osp.ontology.OntologyClass*] = None*) → Union[Set[*simphony_osp.ontology.OntologyIndividual*], *simphony_osp.ontology.OntologyIndividual*, None, Tuple[Optional[*simphony_osp.ontology.OntologyIndividual*]]]

Return the individuals in the session.

The structure of the output can vary depending on the form used for the call. See the "Returns:" section of this docstring for more details on this.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Parameters

- **individuals** – Restrict the individuals to be returned to a certain subset of the individuals in the session.
- **oclass** – Only yield ontology individuals which belong to a subclass of the given ontology class. Defaults to None (no filter).

Returns

The result of the call is a set-like object. This corresponds to the calls *get()*, *get(oclass=___)*.

Calls with **individuals* (Optional[OntologyIndividual],

Tuple[Optional["OntologyIndividual"], ...]):

The position of each element in the result is determined by the position of the corresponding identifier/individual in the given list of identifiers/individuals. In this case, the result can contain *None* values if a given identifier/individual is not in the session, or if it does not satisfy the class filter. This description corresponds to the calls *get(*individuals)*, *get(*individuals, oclass=`___)`*.

Return type Calls without **individuals* (SessionSet)

Raises

- **TypeError** – Objects that are not ontology individuals, identifiers or strings provided as positional arguments.
- **TypeError** – Object that is not an ontology class passed as keyword argument *oclass*.
- **RuntimeError** – Ontology individuals that belong to a different session provided.

iter(**individuals: Union[*simphony_osp.ontology.OntologyIndividual*, rdflib.term.Identifier, str]*, *oclass: Optional[*simphony_osp.ontology.OntologyClass*] = None*) → Union[Iterator[*simphony_osp.ontology.OntologyIndividual*], Iterator[Optional[*simphony_osp.ontology.OntologyIndividual*]]]

Iterate over the ontology individuals in the session.

The structure of the output can vary depending on the form used for the call. See the "Returns:" section of this docstring for more details on this.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Parameters

- **individuals** – Restrict the individuals to be returned to a certain subset of the individuals in the session.
- **oclass** – Only yield ontology individuals which belong to a subclass of the given ontology class. Defaults to None (no filter).

Returns

The position of each element in the result is non-deterministic. This corresponds to the calls *iter()*, *iter(oclass=___)*.

Calls with **individuals* (Iterator[Optional[

OntologyIndividual]]):

The position of each element in the result is determined by the position of the corresponding identifier/individual in the given list of identifiers/individuals. In this case, the result can contain *None* values if a given identifier/individual is not in the session, or if it does not satisfy the class filter. This description corresponds to the calls *iter(*individuals)*, *iter(*individuals, oclass=`___)`*.

Return type Calls without **individuals* (Iterator[OntologyIndividual])

Raises

- **TypeError** – Objects that are not ontology individuals, identifiers or strings provided as positional arguments.
- **TypeError** – Object that is not an ontology class passed as keyword argument *oclass*.
- **RuntimeError** – Ontology individuals that belong to a different session provided.

property locked: bool

Whether the environment is locked or not.

A locked environment will not be closed when using it as a context manager and leaving the context. Useful for setting it as the default environment when it is not intended to close it afterwards.

sparql(*query: str*, *ontology: bool = False*) → simphony_osp.session.session.QueryResult

Perform a SPARQL CONSTRUCT, DESCRIBE, SELECT or ASK query.

By default, the query is performed only on the session's data (the ontology is not included).

Parameters

- **query** – String to use as query.
- **ontology** – Whether to include the ontology in the query or not. When the ontology is included, only read-only queries are possible.

class simphony_osp.session.**SessionSet**(*session: Optional[*simphony_osp.session.Session*] = None*, *oclass: Optional[*simphony_osp.ontology.OntologyClass*] = None*, *uids: Optional[Iterable[simphony_osp.utils.datatypes.UID]] = None*)

A set interface to a session.

This class looks like and acts like the standard *set*, but it is an interface to the methods from *Session* that manage the addition and removal of individuals.

__and__(*other: set*) → set

Return self&other.

__contains__(*item:* simphony_osp.ontology.OntologyIndividual) → bool

Check whether an ontology entity belongs to the session.

__ge__(*other: set*) → bool

Return self>=other.

__gt__(*other: set*) → bool

Return self>other.

__iadd__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet

Return self+=other (equivalent to self|=other).

__iand__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet
Return self&=other.

Should perform the intersection on the underlying data structure.

__ior__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet
Return self|=other.

Should perform the union on the underlying data structure.

__isub__(*other: Any*) → simphony_osp.ontology.utils.DataStructureSet
Return self-=other.

Based on *difference_update*.

__iter__()
The entities contained in the session.

__ixor__(*other: set*) → simphony_osp.ontology.utils.DataStructureSet
Return self^=other.

Should perform the XOR operation on the underlying data structure.

__le__(*other: set*) → bool
Return self<=other.

__len__() → int
Return len(self).

__lt__(*other: set*) → bool
Return self<other.

__ne__(*other: set*) → bool
Return self!=other.

__or__(*other: set*) → set
Return self|other.

__radd__(*other: set*) → set
Return other&self.

__ror__(*other: set*) → set
Return other|self.

__rsub__(*other: set*) → set
Return value-self.

__rxor__(*other: set*) → set
Return value^self.

__xor__(*other: set*) → set
Return self^other.

add(*other: Any*) → None
Add an element to a set.

This has no effect if the element is already present.

all() → *simphony_osp.session.SessionSet*

Return all elements from the set.

Returns All elements from the set, namely the set itself.

any() → Optional[Union[*OntologyAnnotation*, *OntologyAttribute*, *OntologyClass*, *OntologyIndividual*, *OntologyRelationship*, str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID, rdflib.term.URIRef, rdflib.term.Literal]]

Return any element of the set.

Returns Any element from the set if the set is not empty, else None.

clear() → None

Remove all elements from this set.

copy() → set

Return a shallow copy of a set.

difference(*other: Iterable*) → set

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

difference_update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Remove all elements of another set from this set.

discard(*other: Any*) → None

Remove an element from a set if it is a member.

If the element is not a member, do nothing.

intersection(*other: set*) → set

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

intersection_update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Update the set with the intersection of itself and another.

isdisjoint(*other: set*) → bool

Return True if two sets have a null intersection.

issubset(*other: set*) → bool

Report whether another set contains this set.

issuperset(*other: set*) → bool

Report whether this set contains another set.

one() → *simphony_osp.ontology.OntologyIndividual*

Return one element.

Return one element if the set contains one element, else raise an exception.

Returns The only element contained in the set.

Raises

- ***ResultEmptyError*** – No elements in the set.
- ***MultipleResultsError*** – More than one element in the set.

pop() → Any

Remove and return an arbitrary set element.

Raises KeyError if the set is empty.

remove(*other: Any*) → None

Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference(*other: set*) → set

Return the symmetric difference of two sets as a new set.

symmetric_difference_update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Update set with the symmetric difference of it and another.

union(*other: set*) → set

Return the union of sets as a new set.

update(*other: Iterable[*simphony_osp.ontology.OntologyIndividual*]*) → None

Update the set with the union of itself and others.

simphony_osp.tools.**import_file**(*file: Union[str, TextIO, dict, List[dict]]*, *session: Optional[*simphony_osp.session.Session*] = None*, *format: Optional[str] = None*, *all_triples: bool = False*, *all_statements: bool = False*) → Union[*simphony_osp.ontology.OntologyIndividual*, Set[*simphony_osp.ontology.OntologyIndividual*]]

Imports ontology individuals from a file and load them into a session.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Parameters

- **file** – 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** – the session in which the imported data will be stored.

- **format** – the format of the content to import. The supported formats are 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.

- **all_triples** – By default, SimPhoNy imports only ontology individuals. Moreover, not all information about such individuals is imported, but only the details that are relevant from an ontological point of view: the individual's attributes, the classes it belongs to, and its connections to other ontology individuals that are also being copied at the same time.

  However, in some cases, it is necessary to keep all the information about an individual, even if it cannot be understood by SimPhoNy. Set this option to *True* to copy all RDF statements describing ontology individuals, that is, all RDF statements where the individuals are the subject.

One example of a situation where this option is useful is when an individual is attached through an object property to another one which is not properly defined (i.e. has no type assigned). This situation commonly arises when using the *dcat:accessURL* object property.

- **all_statements** – SimPhoNy imports only ontology individuals by default. Moreover, not all information about such individuals is imported, but only the details that are relevant from an ontological point of view.

  Set this option to *True* to import all RDF statements contained in the file, even if they cannot be understood by SimPhoNy. Note that this option differs from *all_triples* because it is more general: the latter imports all triples where an ontology individual is the subject. This one imports all RDF statements, regardless of whether the subjects of the statements are individuals or not.

Returns A set with the imported ontology individuals. If an individual was defined as the "main" one using the SimPhoNy ontology, then only the main individual is returned instead.

simphony_osp.tools.**export_file**(*individuals_or_session: Optional[Union[*simphony_osp.ontology.OntologyIndividual*, Iterable[*simphony_osp.ontology.OntologyIndividual*], *simphony_osp.session.Session*]] = None*, *file: Optional[Union[str, TextIO]] = None*, *main: Optional[Union[str, rdflib.term.Identifier, *simphony_osp.ontology.OntologyIndividual*]] = None*, *format: str = 'text/turtle'*, *all_triples: bool = False*, *all_statements: bool = False*) → Optional[str]

Exports ontology individuals to a variety of formats.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Parameters

- **individuals_or_session** – (OntologyIndividual) A single ontology individual to export, or (Iterable[OntologyIndividual]) an iterable of ontology individuals, (Session) a session to serialize all of its ontology individuals. If *None* is specified, then the current session is exported.
- **file** – either, (None) returns the exported file as a string, or (str) a path, to save the ontology individuals to, 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.
- **main** – the identifier of an ontology individual to be marked as the "main" individual using the SimPhoNy ontology.
- **format** – the target format. Defaults to triples in turtle syntax.
- **all_triples** – By default, SimPhoNy exports only ontology individuals. Moreover, not all information about such individuals is exported, but only the details that are relevant from an ontological point of view: the individual's attributes, the classes it belongs to, and its connections to other ontology individuals that are also being copied at the same time.

  However, in some cases, it is necessary to keep all the information about an individual, even if it cannot be understood by SimPhoNy. Set this option to *True* to export all RDF statements describing ontology individuals, that is, all RDF statements where the individuals are the subject.

  One example of a situation where this option is useful is when an individual is attached through an object property to another one which is not properly defined (i.e. has no type assigned). This situation commonly arises when using the *dcat:accessURL* object property.

- **all_statements** – SimPhoNy exports only ontology individuals by default. Moreover, not all information about such individuals is exported, but only the details that are relevant from an ontological point of view.

  Set this option to *True* to export all RDF statements contained in a session, even if they cannot be understood by SimPhoNy. Note that this option differs from *all_triples* because it is more general: the latter exports all triples where an ontology individual is the subject. This one exports all RDF statements, regardless of whether the subjects of the statements are individuals or not.

Returns The contents of the exported file as a string (if no *file* argument was provided), or nothing.

12.3.1 Search

simphony_osp.tools.search.**find**(*root: simphony_osp.ontology.OntologyIndividual, criterion: typing.Callable[[simphony_osp.ontology.OntologyIndividual], bool] = <function <lambda>>, rel: typing.Union[simphony_osp.ontology.OntologyRelationship, rdflib.term.Node, typing.Iterable[typing.Union[simphony_osp.ontology.OntologyRelationship, rdflib.term.Node]]] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty'), annotation: typing.Union[bool, simphony_osp.ontology.OntologyAnnotation, rdflib.term.Node, typing.Iterable[typing.Union[simphony_osp.ontology.OntologyAnnotation, rdflib.term.Node]]] = True, find_all: bool = True, max_depth: typing.Union[int, float] = inf*) → Union[*simphony_osp.ontology.OntologyIndividual*, None, Iterator[*simphony_osp.ontology.OntologyIndividual*]]

Finds a set of ontology individuals following the given predicates.

Uses the given relationships and annotations for traversal.

Parameters

- **criterion** – Function that returns True on the ontology individual that is searched.
- **root** – Starting point of the search.
- **rel** – The relationship(s) (incl. sub-relationships) to consider for traversal.
- **annotation** – The annotation(s) (incl. sub-annotations) to consider for traversal. Can also take boolean values: when set to *True* any annotation is followed. When set to *False* no annotations are followed.
- **find_all** – Whether to find all ontology individuals satisfying the criterion.
- **max_depth** – The maximum depth for the search. Defaults to float("inf") (unlimited).

Returns The element(s) found. One element (or *None*) is returned when *find_all* is *False*, a generator when *find_all* is True.

simphony_osp.tools.search.**find_by_identifier**(*root:* simphony_osp.ontology.OntologyIndividual, *identifier: Union[rdflib.term.Node, simphony_osp.utils.datatypes.UID, str]*, *rel: Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node]]] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty')*, *annotation: Union[bool, *simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node]]] = True*) → Optional[*simphony_osp.ontology.OntologyIndividual*]

Recursively finds an ontology individual with given identifier.

Only uses the given relationship for traversal.

Parameters

- **root** – Starting point of search.
- **identifier** – The identifier of the entity that is searched.
- **rel** – The relationship (incl. sub-relationships) to consider.
- **annotation** – The annotation(s) (incl. sub-annotations) to consider. Can also take boolean values: when set to *True* any annotation is followed. When set to *False* no annotations are followed.

Returns The resulting individual.

simphony_osp.tools.search.**find_by_class**(*root:* simphony_osp.ontology.OntologyIndividual, *oclass:* simphony_osp.ontology.OntologyClass, *rel: Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node]]] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty')*, *annotation: Union[bool, *simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node]]] = True*) → Iterator[*simphony_osp.ontology.OntologyIndividual*]

Recursively finds ontology individuals with given class.

Only uses the given relationship for traversal.

Parameters

- **root** – Starting point of search.
- **oclass** – The ontology class of the entity that is searched.
- **rel** – The relationship (incl. sub-relationships) to consider for traversal.
- **annotation** – The annotation(s) (incl. sub-annotations) to consider for traversal. Can also take boolean values: when set to *True* any annotation is followed. When set to *False* no annotations are followed.

Returns The individuals found.

simphony_osp.tools.search.**find_by_attribute**(*root:* simphony_osp.ontology.OntologyIndividual, *attribute:* simphony_osp.ontology.OntologyAttribute, *value: Union[str, float, fractions.Fraction, decimal.Decimal, int, bool, bytes, datetime.datetime, simphony_osp.utils.datatypes.Vector, simphony_osp.utils.datatypes.UID]*, *rel: Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node]]] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty')*, *annotation: Union[bool,* simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node]]] = True*) → Iterator[*simphony_osp.ontology.OntologyIndividual*]

Recursively finds ontology individuals by attribute and value.

Only the given relationship will be used for traversal.

Parameters

- **root** – The root for the search.
- **attribute** – The attribute to look for.
- **value** – The corresponding value to filter by.
- **rel** – The relationship (incl. sub-relationships) to consider.
- **annotation** – The annotation(s) (incl. sub-annotations) to consider. Can also take boolean values: when set to *True* any annotation is followed. When set to *False* no annotations are followed.

Returns The individuals found.

simphony_osp.tools.search.**find_relationships**(*root:* simphony_osp.ontology.OntologyIndividual, *find_rel:* simphony_osp.ontology.OntologyRelationship, *find_sub_relationships: bool = False*, *rel: Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyRelationship*, rdflib.term.Node]]] = rdflib.term.URIRef('http://www.w3.org/2002/07/owl#topObjectProperty')*, *annotation: Union[bool,* simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node, Iterable[Union[*simphony_osp.ontology.OntologyAnnotation*, rdflib.term.Node]]] = True*) → Iterator[*simphony_osp.ontology.OntologyIndividual*]

Find given relationship in the subgraph reachable from the given root.

Parameters

- **root** – Only consider the subgraph of individuals reachable from this root.
- **find_rel** – The relationship to find.

- **find_sub_relationships** – Treat relationships that are a sub-relationship of the relationship to find as valid results. Defaults to *False*.
- **rel** – Only consider these relationships (incl. sub-relationships) when searching.
- **annotation** – Only consider these annotations (incl. sub-annotations) when searching. Can also take boolean values: when set to *True* any annotation is followed. When set to *False* no annotations are followed.

Returns The ontology individuals having the given relationship.

simphony_osp.tools.search.**sparql**(*query: str*, *ontology: bool = False*, *session: Optional[simphony_osp.session.Session] = None*) → simphony_osp.session.session.QueryResult

Performs a SPARQL query on a session.

Parameters

- **query** – A string with the SPARQL query to perform.
- **ontology** – Whether to include the ontology in the query or not. When the ontology is included, only read-only queries are possible.
- **session** – 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 SimPhoNy default session.

Returns A QueryResult 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']*.

12.4 Visualization

simphony_osp.tools.**semantic2dot**(**elements: Union[simphony_osp.ontology.OntologyIndividual, simphony_osp.ontology.OntologyNamespace, simphony_osp.session.Session], rel: Optional[Union[simphony_osp.ontology.OntologyRelationship, Iterable[simphony_osp.ontology.OntologyRelationship]]] = None*) → *simphony_osp.tools.semantic2dot.Semantic2Dot*

Utility for plotting ontology entities.

Note: If you are reading the SimPhoNy documentation API Reference, it is likely that you cannot read this docstring. As a workaround, click the *source* button to read it in its raw form.

Plot assertional knowledge (ontology individuals and the relationships between them), plot terminological knowledge (classes, relationships and attributes), or a combination of them.

Parameters

- **elements** – Elements to plot: (Session) plot the whole contents of a session; (OntologyNamespace) plot all the ontology entities contained

  in the ontology namespace;

  (OntologyIndividual) plots an ontology individual, or a collection of them, and the relationships between them if multiple are provided;

- **rel** – When not *None* and when plotting an ontology individual, calls uses the method *find(individual, rel=rel, find_all=True)* from *simphony_osp.tools.search* to additionally plot such individuals.

class simphony_osp.tools.semantic2dot.**Semantic2Dot**(**elements: Union[simphony_osp.ontology.OntologyIndividual, simphony_osp.ontology.OntologyNamespace, simphony_osp.session.Session], rel: Optional[Union[simphony_osp.ontology.OntologyRelationship, Iterable[simphony_osp.ontology.OntologyRelationship]]] = None*)

Class for ojects returned by the *semantic2dot* plotting tool.

Objects of this class produced as outcome of calling the *semantic2dot* plotting tool. They hold the graph information and can be used either to display it in a Jupyter notebook or render the graph to a file.

_repr_mimebundle_(*include: Optional[Iterable[str]]*, *exclude: Optional[Iterable[str]]*)

Render the graph as an image on IPython (e.g. Jupyter notebooks).

render(*filename: Optional[str] = None*, ***kwargs*) → None

Save the graph to a dot and png file.

simphony_osp.tools.**pretty_print**(*entity: simphony_osp.ontology.OntologyEntity*, *rel: typing.Union[simphony_osp.ontology.OntologyRelationship, typing.Iterable[simphony_osp.ontology.OntologyRelationship]] = <OntologyRelationship: http://www.w3.org/2002/07/owl#topObjectProperty>*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Print a tree-like, text representation stemming from an individual.

Generates a tree-like, text-based representation stemming from a given ontology individual, that includes the IRI, ontology classes and attributes of the involved individuals, as well as the relationships connecting them.

Parameters

- **entity** – Ontology individual to be used as starting point of the text-based representation.
- **file** – A file to print the text to. Defaults to the standard output.
- **rel** – Restrict the relationships to consider when searching for attached individuals to subclasses of the given relationships.

12.5 Tools

simphony_osp.tools.**host**(*wrapper: Type[simphony_osp.session.wrapper.WrapperSpawner]*, *configuration_string: str = ''*, *create: bool = False*, *hostname: str = '127.0.0.1'*, *port: int = 6537*, *username: Optional[str] = None*, *password: Optional[str] = None*, ***kwargs: Union[str, int, float, bool, None, Iterable[Optional[Union[str, int, float, bool]]]]*)

Host a server based on a wrapper.

Opens the specified wrapper and starts listening for clients. The clients can connect to the wrapper's session and perform actions.

Parameters

- **wrapper** – The wrapper to be used.
- **configuration_string** – The configuration string of the wrapper.
- **create** – The value of the argument *create* for the wrapper.
- **hostname** – Hostname where the server will listen.
- **port** – The port that the server will use to listen.
- **username** – A username for authenticating the client.
- **password** – A password for authenticating the client.
- ****kwargs** – Keyword arguments for the wrapper.

simphony_osp.tools.**branch**(*individual*, **individuals*, *rel:* simphony_osp.ontology.OntologyRelationship) → *simphony_osp.ontology.OntologyIndividual*

Like *connect*, but returns the element you connect to.

This makes it easier to create large structures involving ontology individuals.

Parameters

- **individual** – The ontology individual that is the subject of the connections to be created.
- **individuals** – Ontology individuals to connect to.
- **rel** – Relationship to use.

Returns The ontology individual that is the subject of the connections to be created (the first argument).

simphony_osp.tools.**relationships_between**(*subj:* simphony_osp.ontology.OntologyIndividual, *obj:* simphony_osp.ontology.OntologyIndividual) → Set[*simphony_osp.ontology.OntologyRelationship*]

Get the set of relationships between two ontology individuals.

Parameters

- **subj** – The subject of the relationship.
- **obj** – The object (target) of the relationship.

Returns The set of relationships between the given subject and object individuals.

12.6 Development

class simphony_osp.development.**Wrapper**(***kwargs: Union[str, int, float, bool, None, Iterable[Optional[Union[str, int, float, bool]]]]*)

To be implemented by interface/wrapper developers.

This is the most generic type of interface.

__init__(***kwargs: Union[str, int, float, bool, None, Iterable[Optional[Union[str, int, float, bool]]]]*)

Initialize the wrapper.

The *__init__* method accepts JSON-serializable keyword arguments in order to let the user configure parameters of the wrapper that are not configurable via the ontology. For example, the type of solver used by a simulation engine.

Save such parameters to private attribute to use them later (e.g. in the *open* method).

Parameters **kwargs** – JSON-serializable keyword arguments that contain no nested JSON objects (check the type hint for this argument).

add(*triple: Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]*) → bool

Inspect and control the addition of triples to the base graph.

Parameters **triple** – The triple being added.

Returns True when the triple should be added to the base graph. False when the triple should be caught, and therefore not added to the base graph. This triple will be latter available during commit on the buffer so that the changes that it introduces can be translated to the data structure.

abstract close() → None

Close the data source that the interface interacts with.

This method should NOT commit uncommitted changes.

This method should close the connection that was obtained in *open*, and free any locked up resources.

You can expect calls to this method even when the data source is already closed. Therefore, an implementation like the following is recommended.

```
>>> def close(self):
>>>     if your_data_source_is_already_closed:
>>>         return
>>>
>>>     # Close the connection to your data source.
>>>     # your_data_source_is_already_closed is for now True
```

abstract commit() → None

This method commits the changes made by the user.

Within this method, you have access to the following resources:

- *self.base*: The base graph (rw). You are not expected to modify it.
- *self.old_graph*: The old graph (ro).
- *self.new_graph*: The new graph (ro).
- *self.buffer*: The buffer of triples caught by *add* and *remove* (rw) that you now have to reflect on the data structures of your software.
- *self.session_base*: A session based on the base graph (rw). You are not expected to modify it.
- *self.session_old*: A session based on the old graph (ro).
- *self.session_new*: A session based on the new graph (ro).
- *self.session*: same as *self.session_new*.
- *self.added*: A list of added individuals (rw). You are not expected to modify the entities.
- *self.updated*: A list of updated individuals (rw). You are not expected to modify the entities.
- *self.deleted*: A list of deleted individuals (rw). You are not expected to modify the entities.

Before updating the data structures, check that the changes provided by the user do not leave them in a consistent state. This necessary because SimPhoNy cannot revert the changes you make to your data structures. Raise an AssertionError if the check fails.

Raises **AssertionError** – When the data provided by the user would produce an inconsistent or unpredictable state of the data structures.

compute(***kwargs: Union[str, int, float, bool, None, Iterable[Optional[Union[str, int, float, bool]]]]*) → None

Compute new information (e.g. run a simulation).

Compute the new information on the backend and reflect the changes on the base graph. The default session is the base session.

The base graph is available on *self.base*, and a session based on the base graph is available on *self.session* and *self.session_base*.

delete(*key: str*) → None

Delete a file.

Delete the file associated with the provided key.

Parameters **key** – Identifier of the individual associated with the file.

load(*key: str*) → BinaryIO

Retrieve a file.

Provide a file handle associated with the provided key.

Parameters **key** – Identifier of the individual associated with the file.

Returns File handle associated with the provided key.

abstract open(*configuration: str*, *create: bool = False*) → None

Open the data source that the wrapper interacts with.

You can expect calls to this method even when the data source is already accesible, therefore, an implementation similar to the one below is recommended.

```
>>> def open(self, configuration: str, create: bool = False):
>>>     if your_data_source_is_already_open:
>>>         return
>>>         # To improve the user experience you can check if the
>>>         # configuration string leads to a resource different from
>>>         # the current one and raise an error informing the user.
>>>
>>>     # Connect to your data source...
>>>     # your_data_source_is_already_open is for now True.
```

If you are using a custom base graph, please set *self.base = your_graph* within this method. Otherwise, an empty base graph will be created instead.

Parameters

- **configuration** – Used to locate or configure the data source to be opened.
- **create** – Whether the data source should be created if it does not exist. When false, if the data source does not exist, you should raise an exception. When true, create an empty data source.

abstract populate() → None

Populate the base session so that it represents the data source.

This command is run after the data source is opened. Here you are expected to populate the base graph so that its information mimics the information on the data source, unless you are generating triples on the fly using the *triples* method. The default session inside this method is a session based on the base graph.

The base graph is available on *self.base*, and a session based on the base graph is available on *self.session* and *self.session_base*.

remove(*pattern: Tuple[Optional[rdflib.term.Node], Optional[rdflib.term.Node], Optional[rdflib.term.Node]]*) → Iterator[Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]]

Inspect and control the removal of triples from the base graph.

Parameters **pattern** – The pattern being removed.

Returns An iterator with the triples that should be removed from the base graph. Any triples not included will not be removed, and will be available on the buffer during commit.

save(*key: str*, *file: BinaryIO*) → None

Save a file.

Read the bytestream offered as a file handle and save the contents somewhere, associating them with the provided key for later retrieval.

Parameters

- **key** – Identifier of the individual associated with the file.
- **file** – File (as a file-like object) to be saved.

triples(*pattern: Tuple[Optional[rdflib.term.Node], Optional[rdflib.term.Node], Optional[rdflib.term.Node]]*) → Iterator[Tuple[rdflib.term.Node, rdflib.term.Node, rdflib.term.Node]]

Intercept a triple pattern query.

Can be used to produce triples that do not exist on the base graph on the fly.

Parameters **pattern** – The pattern being queried.

Returns An iterator yielding triples

class simphony_osp.development.**BufferType**(*value*)

Bases: enum.IntEnum

Enum representing the two possible types of triple buffers.

- ADDED: For triples that have been added.
- DELETED: For triples that have been deleted.

ADDED = 0

DELETED = 1

class simphony_osp.development.**Operations**(*individual:* OntologyIndividual)

Bases: abc.ABC

Define operations for an ontology class.

__init__(*individual:* OntologyIndividual)

Initialization of your instance of the operations.

It is recommended to save the individual that is received as an argument to an instance attribute, as the operations to be executed are supposed to be related to it.

abstract property iri: Union[str, Iterable[str]]

IRI of the ontology class for which operations should be registered.

It is also possible to define several IRIs at once (by returning an iterable).

simphony_osp.development.**find_operations**(*packages: Optional[Union[List[str], str]] = None*) → Set[str]

Generates the entry point definitions for operations.

Scans one or several packages and generates sets of strings that can be used in *setup.py* to register SimPhoNy ontology operations.

This method is meant to ease the work that operation developers have to do in their *setup.py* files.

Parameters **packages** – name(s) of the package(s) to scan. When left empty, all packages on the working directory are scanned.

Returns

Set of strings that can be used with the "simphony_osp.ontology.operations" entry point.

Example

{"File = simphony_osp.ontology.operations.file:File"}

CHAPTER

THIRTEEN

CONTRIBUTE

This section aims to explain how we develop and organize, in order to help those that want to contribute to SimPhoNy.

13.1 Development 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

13.2 Code organization

There are 3 main categories of repositories:

- The *SimPhoNy* repository contains the core the SimPhoNy Open Simulation Platform, that the wrappers depend on.
- Each *wrapper* is in its own repository on GitHub or GitLab, mimicking the wrapper-development repository.
- *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.

There may be `master/main` and `dev` branches for several major releases.

13.3 Development 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 repository.
- 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. A maintainer 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 developed, `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 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 synchronize them:

13.4 Coding

13.4.1 Documentation

- All code must be properly documented with meaningful comments.
- For readability, we 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.

13.4.2 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.

13.4.3 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.

13.4.4 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 `simphony-osp` 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.

13.5 Contribute to SimPhoNy

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.

13.6 Contribute to the development of a wrapper

For a sample wrapper, visit the wrapper-development repo.

README files should include:

- Information regarding the purpose of the wrapper and the backend used.
- Installation instructions.
- Any other necessary information for users and other developers.

13.7 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

FOURTEEN

RELATED LINKS

Here are links to other projects relevant for this documentation.

SimPhoNy:

- GitLab's SimPhoNy group
- GitHub's SimPhoNy group
- SimPhoNy repository
- SimPhoNy Wrappers on GitLab
- Wrapper development repository

Technologies used:

- Docker, used for the CI and engines
- Sphinx, used for the documentation
- PlantUML, used for the diagrams

CHAPTER

FIFTEEN

ACKNOWLEDGEMENTS

SimPhoNy 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

The SimPhoNy-OSP 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 SimPhoNy, which were originally demonstrated under the project simphony-common.

CHAPTER

SIXTEEN

LICENSE

CHAPTER

SEVENTEEN

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 Informatics Team at Fraunhofer IWM.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W