AllegroGraph Python client Documentation

Release 100.2.1.dev0

Franz Inc.

Aug 07, 2018

Contents

1 Installation **3**
- 1.1 Requirements . . . 3
- 1.2 Installation . . . 3
- 1.3 Testing . . . 4
- 1.4 Proxy setup . . . 4
- 1.5 Optional requirements . . . 5
- 1.6 Windows . . . 6
- 1.7 Troubleshooting . . . 7

2 Tutorial **9**
- 2.1 Prerequisites . . . 9
- 2.2 Setting the environment for the tutorial . . . 9
- 2.3 Terminology . . . 10
- 2.4 Creating Users with WebView . . . 11
- 2.5 Example 1: Creating a repository and triple indices . . . 12
- 2.6 Example 2: Asserting and retracting triples . . . 16
- 2.7 Example 3: A SPARQL query . . . 18
- 2.8 Example 4: Statement matching . . . 20
- 2.9 Example 5: Literal values . . . 20
- 2.10 Example 6: Importing triples . . . 36
- 2.11 Example 7: Querying multiple contexts . . . 38
- 2.12 Example 8: Exporting triples . . . 39
- 2.13 Example 9: Exporting query results . . . 40
- 2.14 Example 10: Graphs in SPARQL . . . 42
- 2.15 Example 11: Namespaces . . . 46
- 2.16 Example 12: Free Text indexing . . . 48
- 2.17 Example 13: SPARQL query forms . . . 51
- 2.18 Example 14: Parametric queries . . . 55
- 2.19 Example 15: Range queries . . . 57
- 2.20 Example 16: Federated repositories . . . 59
- 2.21 Example 17: Triple attributes . . . 60
- 2.22 Example 18: Pandas support . . . 69
- 2.23 Running AG on AWS EC2 . . . 71

3 AllegroGraph Python API Reference **81**
- 3.1 AllegroGraphServer class . . . 81
- 3.2 Catalog class . . . 83

3.3 Spec module . . . 83
3.4 Repository class . . . 83
3.5 Utility connection functions . . . 84
3.6 RepositoryConnection class . . . 84
3.7 Query Class (and Subclasses) . . . 88
3.8 QueryResult Class . . . 90
3.9 RepositoryResult class . . . 91
3.10 Statement Class . . . 91
3.11 ValueFactory Class . . . 92

4 AllegroGraph Python client release history **93**
4.1 Release 100.3.0 . . . 93
4.2 Release 100.2.0 . . . 93
4.3 Release 100.1.2 . . . 93
4.4 Release 100.1.1 . . . 94
4.5 Release 100.1.0 . . . 94
4.6 Release 100.0.4 . . . 95
4.7 Release 100.0.3 . . . 95
4.8 Release 100.0.2 . . . 95
4.9 Release 100.0.1 . . . 97
4.10 Release 100.0.0 . . . 97
4.11 Release 6.2.2.0.4 . . . 98
4.12 Release 6.2.2.0.1 . . . 98
4.13 Release 6.2.2.0.0 . . . 98

5 Indices and tables **99**

Python Module Index **101**

The AllegroGraph Python API offers convenient and efficient access to an AllegroGraph server from a Python-based application. This API provides methods for creating, querying and maintaining RDF data, and for managing the stored triples. The AllegroGraph Python API deliberately emulates the Eclipse RDF4J (formerly Aduna Sesame) API to make it easier to migrate from RDF4J to AllegroGraph. The AllegroGraph Python API has also been extended in ways that make it easier and more intuitive than the RDF4J API.

Contents:

CHAPTER 1

Installation

1.1 Requirements

Python versions 2.6+ and 3.3+ are supported. The installation method described here uses the pip package manager. On some systems this might require installing an additional package (e.g. `python-pip` on RHEL/CentOS systems). All third-party libraries used by the Python client will be downloaded automatically during installation.

See also the *Optional requirements* section below.

1.2 Installation

Important: It is highly recommended to perform the install in a virtualenv environment.

The client can be installed from PyPI using the `pip` package manager:

```
pip install agraph-python
```

Alternatively, a distribution archive can be obtained from ftp://ftp.franz.com/pub/agraph/python-client/ and installed using *pip*:

```
pip install agraph-python-<VERSION>.tar.gz
```

Warning: Python 2.6 users should consider installing the simplejson package:

```
pip install simplejson
```

since the built-in JSON module in that version of Python offers unsatisfactory performance. The AllegroGraph Python client will detect and use simplejson automatically if it is installed.

1.2.1 Offline installation

If it is not possible to access PyPI from the target machine, the following steps should be taken:

- In a compatible environment with unrestricted network access run:

```
pip wheel agraph-python
```

- If desired do the same for optional dependencies:

```
pip wheel pycurl simplejson
```

- This will generate a number of `.whl` files in the current directory. These files must be transferred to the target machine.
- On the target machine use this command to install:

```
pip install --no-index --find-links=<DIR> agraph-python
```

  where `<DIR>` is the directory containing the `.whl` files generated in the previous step.

1.3 Testing

To validate the installation make sure that you have access to an AllegroGraph server and run the following Python script:

```
from franz.openrdf.connect import ag_connect
with ag_connect('repo', host='HOST', port='PORT',
                user='USER', password='PASS') as conn:
    print conn.size()
```

Substitute appropriate values for the HOST/PORT/USER/PASS placeholders. If the script runs successfully a new repository named *repo* will be created.

1.4 Proxy setup

It is possible to configure the AllegroGraph Python client to use a proxy for all its connection to the server. This can be achieved by setting the `AGRAPH_PROXY` environment variable, as in the following example:

```
# Create a SOCKS proxy for tunneling to an internal network
ssh -fN -D 1080 user@gateway.example.com
# Configure agraph-python to use this proxy
export AGRAPH_PROXY=socks://localhost:1080
```

The format of the `AGRAPH_PROXY` value is `TYPE://HOST:PORT`, where `TYPE` can be either `http`, `socks4`, `socks5` or `socks` (a synonym for `socks5`). Note that if a SOCKS proxy is used, DNS lookups will be performed by the proxy server.

1.4.1 Unit tests

The Python client includes a suite of unit tests that can be run after installation. The tests are executed using the `pytest` framework and also use a few utilities from `nose`, so these two packages have to be installed:

```
pip install pytest nose
```

The tests require a running AllegroGraph server instance. The configuration of this server is passed to the tests through environment variables:

```
# Host and port where the server can be reached. These values are the
# default, it is only necessary to define the variables below if your
# setup is different
export AGRAPH_HOST=localhost
export AGRAPH_PORT=10035

# Tests will create repositories in this catalog.
# It must exist on the server. Use "/" for the root catalog.
export AGRAPH_CATALOG=tests

# Login credentials for the AG server.
# The user must have superuser privileges.
export AGRAPH_USER=test

# Use a prompt to read the password
read -s -r -p "Password for user ${AGRAPH_USER}: " AGRAPH_PASSWORD
export AGRAPH_PASSWORD
```

To run the tests, type:

```
pytest --pyargs franz.openrdf.tests.tests --pyargs franz.openrdf.tests.newtests
```

1.5 Optional requirements

The Python client can utilize a few additional third-party libraries if these are available on the host system:

- pycurl: can be used as the HTTP backend. It might offer better performance than `requests` (the default backend).
- simplejson: recommended for Python 2.6 users to significantly improve performance. Has negligible impact on other Python versions.

These can be downloaded and installed from PyPI:

```
pip install pycurl
pip install simplejson
```

Since the packages discussed here use native extensions, it is necessary to have a proper development environment set up so that the compilation may succeed. This environment must include:

- A C compiler
- Python development headers
- libcurl development headers

Below we describe more detailed setup instructions for some of the supported systems.

1.6 Windows

The required packages are available in binary form for Windows, so it is not necessary to install any compilers or headers.

1.6.1 RHEL 6/7 and compatible systems

On RHEL-like systems the requirements mentioned above can be satisfied by following these steps (note that root privileges are required):

- Enable EPEL repositories, since some of the required packages are only available there. The detailed instructions can be found here. On CentOS systems, simply run:

```
yum -y install epel-release
```

- Install the required packages:

```
yum -y install python-devel python-pip libcurl-devel gcc
```

- Before installing the AllegroGraph Python client make sure that the following environment variable is set:

```
export PYCURL_SSL_LIBRARY=nss
```

- To use virtual environments (recommended) an additional package is needed:

```
yum -y install python-virtualenv
```

1.6.2 Ubuntu

The following packages are required to use the client with Python 2:

```
apt-get install python-pip libcurl-gnutls libcurl4-gnutls-dev libgnutls28-dev
```

For Python 3 this becomes:

```
apt-get install python3-pip libcurl-gnutls libcurl4-gnutls-dev libgnutls28-dev
```

Note: *Using different SSL backends.*

Ubuntu offers three variants of curl, built using different SSL libraries. These variants differ in their licensing and SSL related capabilities (see https://curl.haxx.se/docs/ssl-compared.html for more details). The instructions above use the GnuTLS version. In most cases this is an acceptable choice, but it is possible to use a different SSL implementation by installing appropriate packages before installing the AllegroGraph Python client.

To use the OpenSSL backend in curl:

```
apt-get install libcurl4-openssl-dev libssl-dev
```

For GnuTLS:

```
apt-get install libcurl4-gnutls-dev libgnutls28-dev
```

For NSS:

```
apt-get install libcurl4-nss-dev libnss3-dev
```

Note that if the client has already been installed it is necessary to reinstall the `pycurl` package in order to switch SSL backends:

```
# Uninstall old package
pip uninstall pycurl

# Reinstall, ignoring PIP cache (to force recompilation)
pip install --no-cache-dir pycurl
```

1.6.3 Arch Linux

On Arch the following packages are needed to use the client with Python 2:

```
pacman -S gcc python2 python2-pip libcurl
```

For Python 3 use:

```
pacman -S gcc python python-pip libcurl
```

1.7 Troubleshooting

If you see an error similar to the following:

```
ImportError: pycurl: libcurl link-time ssl backend (nss) is
different from compile-time ssl backend (none/other)
```

Perform this procedure (replacing <*VERSION*> with the actual version):

```
# Uninstall pycurl:
pip uninstall pycurl

# Set the required compile-time option for pycurl
export PYCURL_SSL_LIBRARY=nss

# Reinstall, but ignore cached packages (force recompile)
pip install --no-cache-dir agraph-<VERSION>-python-client.tar.gz
```

CHAPTER 2

Tutorial

This is an introduction to the Python client API to the Python Application Programmer's Interface (API) to Franz Inc.'s AllegroGraph.

The Python client tutorial rests on a simple architecture consisting of the following components:

- An AllegroGraph server
- A Python interpreter
- The AllegroGraph Python API package which allows Python to communicate with an AllegroGraph server over HTTP or HTTPS.
- This document - containing example code that can be pasted into the interpreter.

2.1 Prerequisites

The tutorial examples can be run on any system which supports Python 2.6, 2.7 and 3.4+. The tutorial assumes that AllegroGraph and the Python client have been installed and configured using the procedure described in the *Installation* document.

To make the code in this document compatible with Python 2 we need the following import statement (not required on Python 3):

```
from __future__ import print_function
```

2.2 Setting the environment for the tutorial

Before running any of the tutorial examples it is necessary to set a few variables that describe the location of the AllegroGraph server and credentials used to access it. By default the Python client looks for these parameters in environment variables.

The variables that must be set are:

- AGRAPH_HOST: Specifies the IP address or hostname of the agraph server. Defaults to '127.0.0.1'.
- AGRAPH_PORT: Specifies the port of the agraph server at AGRAPH_HOST. The port used by the server is specified in the agraph.cfg file. The default is 10035.
- AGRAPH_USER: Specifies the username for authentication. For best results, this should name an agraph user with superuser privileges. If a non-superuser is specified, that user must have **Start Session** privileges to run example 6 and examples dependent on that example, and **Evaluate Arbitrary Code** privileges to run example 17. There is no default.
- AGRAPH_PASSWORD: Specifies the password for authentication of AGRAPH_USER. There is no default..

The agraph.cfg file (see the Server Installation document) specifies values like the port used by the server and the catalogs defined. Refer to that file if necessary (or ask your system administrator if you do not have access to the file).

2.2.1 Troubleshooting

If the tutorial examples throw a connection error in *Example 6: Importing triples*, see the discussion Session Port Setup in the Server Installation document.

If you see an error similar to the following

```
ImportError: pycurl: libcurl link-time ssl backend (nss) is
different from compile-time ssl backend (none/other)
```

Perform this procedure (replacing {agraph-version} with the actual version)

```
# Uninstall pycurl
pip uninstall pycurl

# Set the required compile-time option for pycurl
export PYCURL_SSL_LIBRARY=nss

# Reinstall, but ignore cached packages (force recompile)
pip install --no-cache-dir agraph-{agraph-version}-client-python.tar.gz
```

2.3 Terminology

We need to clarify some terminology before proceeding.

"RDF" is the Resource Description Framework defined by the World Wide Web Consortium (W3C). It provides an elegantly simple means for describing multi-faceted resource objects and for linking them into complex relationship graphs. AllegroGraph Server creates, searches, and manages such RDF graphs.

A "URI" is a Uniform Resource Identifier. It is label used to uniquely identify various types of entities in an RDF graph. A typical URI looks a lot like a web address: <http://www.company.com/project/class#number>. In spite of the resemblance, a URI is not a web address. It is simply a unique label.

A "triple" is a data statement, a "fact", stored in RDF format. It states that a resource has an attribute with a value. It consists of three fields:

- Subject: The first field contains the URI that uniquely identifies the resource that this triple describes.
- Predicate: The second field contains the URI identifying a property of this resource, such as its color or size, or a relationship between this resource and another one, such as parentage or ownership.
- Object: The third field is the value of the property. It could be a literal value, such as "red", or the URI of a linked resource.

A "quad" is a triple with an added "context" field, which is used to divide the repository into "subgraphs." This context or subgraph is just a URI label that appears in the fourth field of related triples.

A "resource description" is defined as a collection of triples that all have the same URI in the subject field. In other words, the triples all describe attributes of the same thing.

A "statement" is a client-side Python object that describes a triple.

In the context of AllegroGraph Server:

- A "catalog" is a list of repositories owned by an AllegroGraph server.
- A "repository" is a collection of triples within a catalog.
- A "context" is a subgraph of the triples in a repository.
- If contexts are not in use, the triples are stored in the default graph.

2.4 Creating Users with WebView

Each connection to an AllegroGraph server runs under the credentials of a registered AllegroGraph user account.

2.4.1 Initial Superuser Account

The installation instructions for AllegroGraph advise you to create a **superuser** called "test". This is the user which is used by the Python tutorial if the environment variables `AGRAPH_USER` and `AGRAPH_PASSWORD` are not set.

2.4.2 Users, Permissions, Access Rules, and Roles

AllegroGraph user accounts may be given any combination of the following three permissions:

- Superuser
- Start Session
- Evaluate Arbitrary Code

In addition, a user account may be given read, write or read/write access to individual repositories.

It is easiest to run the Python tutorial as a Superuser. That way you do not have to worry about permissions.

If you run as a non-Superuser, you need **Start Session** permission for *Example 6: Importing triples* and other examples that require a session. You also need read/write access on the appropriate catalogs and repositories.

You can also define a **role** (such as "librarian") and give the role a set of permissions and access rules. Then you can assign this shared role to several users. This lets you manage their permissions and access by editing the role instead of the individual user accounts.

A **superuser** automatically has all possible permissions and unlimited access. A superuser can also create, manage and delete other user accounts. Non-superusers cannot view or edit account settings.

A user with the **Start Sessions** permission can use the AllegroGraph features that require spawning a dedicated session, such as transactions and Social Network Analysis (SNA). If you try to use these features without the appropriate permission, you'll encounter errors.

A user with permission to **Evaluate Arbitrary Code** can run Prolog Rule Queries. This user can also do anything else that allows executing Lisp code, such as defining select-style generators, doing eval-in-server, or loading server-side files.

2.4.3 WebView

WebView is AllegroGraph's browser-based graphical user interface for user and repository management. It allows you to create, query, and maintain repositories interactively.

To connect to WebView, simply direct your web browser to the AllegroGraph port of your server. If you have installed AllegroGraph locally (and used the default port number), use:

```
http://localhost:10035
```

You will be asked to log in. Use the superuser credentials described in the previous section.

The first page of WebView is a summary of your catalogs and repositories. Select *Admin* → *Users* from the navigation menu at the top of the page.

This exposes the `Users and Roles` page. This is the page for creating and managing user accounts.

To create a new user, click the *[add a user]* link.

This exposes a small form where you can enter the username and password. Click *OK* to save the new account.

The new user will appear in the list of users. Click the *[edit]* link to open a control panel for the new user account:

Use the checkboxes to apply permissions to this account (start session is needed by *Example 6: Importing triples*).

It is important that you set up access permissions for the new user. Use the form to create an access rule by selecting read, write or read/write access, naming a catalog (or * for all), and naming a repository within that catalog (or * for all). Click the *[add]* link.

This creates an access rule for your new user. The access rule will appear in the permissions display:

This new user can log in and perform transactions on any repository in the system.

To repeat, the "test" superuser is all you need to run all of the tutorial examples. This section is for the day when you want to issue more modest credentials to some of your users.

2.5 Example 1: Creating a repository and triple indices

2.5.1 Listing catalogs

The first task is to attach to our AllegroGraph Server and open a repository. To achieve this we build a chain of Python objects, ending in a "connection" object that lets us manipulate triples in a specific repository. The overall process of generating the connection object follows this diagram:

The first example opens (or creates) a repository by building a series of client-side objects, culminating in a "connection" object. The connection object contains the methods that let us manipulate triples in a specific repository.	

Before we start, we will extract the location of the AG server from environment variables

```
import os

AGRAPH_HOST = os.environ.get('AGRAPH_HOST')
AGRAPH_PORT = int(os.environ.get('AGRAPH_PORT', '10035'))
AGRAPH_USER = os.environ.get('AGRAPH_USER')
AGRAPH_PASSWORD = os.environ.get('AGRAPH_PASSWORD')
```

AllegroGraph connection functions use these environment variables as defaults, but we will pass the values explicitly to illustrate how to specify connection parameters in Python.

The example first connects to an AllegroGraph Server by providing the endpoint (host IP address and port number) of an already-launched AllegroGraph server. This creates a client-side server object, which can access the AllegroGraph server's list of available catalogs through the `listCatalogs()` method. Note that the name of the root catalog will be represented by `None`:

```
from franz.openrdf.sail.allegrographserver import AllegroGraphServer

print("Connecting to AllegroGraph server --",
      "host:'%s' port:%s" % (AGRAPH_HOST, AGRAPH_PORT))
server = AllegroGraphServer(AGRAPH_HOST, AGRAPH_PORT,
                            AGRAPH_USER, AGRAPH_PASSWORD)
print("Available catalogs:")
for cat_name in server.listCatalogs():
    if cat_name is None:
        print('  - <root catalog>')
    else:
        print('  - ' + str(cat_name))
```

This is the output so far:

```
Connecting to AllegroGraph server -- host:'...
Available catalogs:
  - <root catalog>
  ...
```

This output says that the server has the root catalog and possibly also some other catalogs that someone has created for some experimentation.

2.5.2 Listing repositories

In the next part of this example, we use the `openCatalog()` method to create a client-side catalog object. In this example we will connect to the root catalog. When we look inside that catalog, we can see which repositories are available:

```
catalog = server.openCatalog('')
print("Available repositories in catalog '%s':" % catalog.getName())
for repo_name in catalog.listRepositories():
    print('  - ' + repo_name)
```

The corresponding output lists the available repositories. When you run the examples, you may see a different list of repositories.

```
Available repositories in catalog 'None':
 - pythontutorial
 - greenthings
 - redthings
```

2.5.3 Creating repositories

The next step is to create a client-side repository object representing the respository we wish to open, by calling the `getRepository()` method of the catalog object. We have to provide the name of the desired repository (`'python-tutorial'`), and select one of four access modes:

- `Repository.RENEW` clears the contents of an existing repository before opening. If the indicated repository does not exist, it creates one.
- `Repository.OPEN` opens an existing repository, or throws an exception if the repository is not found.
- `Repository.ACCESS` opens an existing repository, or creates a new one if the repository is not found.
- `Repository.CREATE` creates a new repository, or throws an exception if one by that name already exists.

```
from franz.openrdf.repository.repository import Repository

mode = Repository.RENEW
my_repository = catalog.getRepository('python-tutorial', mode)
my_repository.initialize()
```

A new or renewed repository must be initialized, using the `initialize()` method of the repository object.

2.5.4 Connecting to a repository

The goal of all this object-building has been to create a client-side connection object, whose methods let us manipulate the triples of the repository. The repository object's `getConnection()` method returns this connection object.

```
conn = my_repository.getConnection()
print('Repository %s is up!' % my_repository.getDatabaseName())
print('It contains %d statement(s).' % conn.size())
```

The `size()` method of the connection object returns how many triples are present. In the `example1()` function, this number will always be zero because we "renewed" the repository.

This is the output so far:

```
Repository python-tutorial is up!
It contains 0 statement(s).
```

2.5.5 Managing indices

Whenever you create a new repository, you should stop to consider which kinds of triple indices you will need. This is an important efficiency decision. AllegroGraph uses a set of sorted indices to quickly identify a contiguous range of triples that are likely to match a specific query pattern.

These indices are identified by names that describe their organization. The default set of indices are called **spogi, posgi, ospgi, gspoi, gposi, gospi**, and **i** , where:

- **S** stands for the subject URI.
- **P** stands for the predicate URI.
- **O** stands for the object URI or literal.
- **G** stands for the graph URI.
- **I** stands for the triple identifier (its unique id number within the triple store).

The order of the letters denotes how the index has been organized. For instance, the **spogi** index contains all of the triples in the store, sorted first by subject, then by predicate, then by object, and finally by graph. The triple id number is present as a fifth column in the index. If you know the URI of a desired resource (the *subject* value of the query pattern), then the **spogi** index lets you quickly locate and retrieve all triples with that subject.

The idea is to provide your respository with the indices that your queries will need, and to avoid maintaining indices that you will never need.

We can use the connection object's `listValidIndices()` method to examine the list of all possible AllegroGraph triple indices:

```
indices = conn.listValidIndices()
group_size = 5
print('All valid triple indices:')
for offset in range(0, len(indices), group_size):
    group = indices[offset:offset + group_size]
    print('  ', ' '.join(group))
```

This is the list of all possible valid indices:

```
All valid triple indices:
   spogi spgoi sopgi sogpi sgpoi
   sgopi psogi psgoi posgi pogsi
   pgsoi pgosi ospgi osgpi opsgi
   opgsi ogspi ogpsi gspoi gsopi
   gpsoi gposi gospi gopsi i
```

AllegroGraph can generate any of these indices if you need them, but it creates only seven indices by default. We can see the current indices by using the connection object's `listIndices()` method:

```
indices = conn.listIndices()
print('Current triple indices:', ', '.join(indices))
```

There are currently seven indices

```
Current triple indices: i, gospi, gposi, gspoi, ospgi, posgi, spogi
```

The indices that begin with "g" are sorted primarily by subgraph (or "context"). If your application does not use subgraphs, you should consider removing these indices from the repository. You don't want to build and maintain triple indices that your application will never use. This wastes CPU time and disk space. The connection object has a convenient `dropIndex()` method:

```
print("Removing graph indices...")
conn.dropIndex("gospi")
conn.dropIndex("gposi")
conn.dropIndex("gspoi")
indices = conn.listIndices()
print('Current triple indices:', ', '.join(indices))
```

Having dropped three of the triple indices, there are now four remaining:

```
Removing graph indices...
Current triple indices: i, ospgi, posgi, spogi
```

The **i** index is for deleting triples by using the triple id number. It is also required for *free text indexing*. The **ospgi** index is sorted primarily by object value, which makes it possible to efficiently retrieve a range of object values from the index. Similarly, the **posgi** index lets us quickly reach for a triples that all share the same predicate. We mentioned previously that the **spogi** index speeds up the retrieval of triples that all have the same subject URI.

As it happens, we may have been overly hasty in eliminating all of the graph indices. AllegroGraph can find the right matches as long as there is *any* one index present, but using the "right" index is much faster. Let's put one of the graph indices back, just in case we need it. We'll use the connection object's `addIndex()` method:

```
print("Adding one graph index back in...")
conn.addIndex("gspoi")
indices = conn.listIndices()
print('Current triple indices:', ', '.join(indices))
```

```
Adding one graph index back in...
Current triple indices: i, gspoi, ospgi, posgi, spogi
```

2.5.6 Releasing resources

Both the connection and the repository object must be closed to release resources once they are no longer needed. We can use the `shutDown()` and `close()` methods to do this:

```
conn.close()
my_repository.shutDown()
```

It is safer and more convenient to ensure that the resources are released by using the `with` statement:

```
with catalog.getRepository('python-tutorial', Repository.OPEN) as repo:
    # Note: an explicit call to initialize() is not required
    # when using the `with` statement.
    with repo.getConnection() as conn:
        print('Statements:', conn.size())
```

```
Statements: 0
```

2.5.7 Utility functions

Creating the intermediate server, catalog and repository objects can be tedious when the only thing required is a single connection to one repository. In such circumstances it might be more convenient to use the `ag_connect()` function. That is what we will do in further examples. Here is a brief example of using `ag_connect()`

```
from franz.openrdf.connect import ag_connect

with ag_connect('python-tutorial', create=True, clear=True) as conn:
    print('Statements:', conn.size())
```

This function take care of creating all required objects and the returned context manager ensures that all necessary initialization steps are taken and no resources are leaked. The `create` and `clear` arguments ensure that the repository is empty and that it is created if necessary.

```
Statements: 0
```

2.6 Example 2: Asserting and retracting triples

In this example we show how to create resources describing two people, Bob and Alice, by asserting triples into the repository. The example also retracts and replaces a triple. Assertions and retractions to the triple store are executed by `add()` and `remove()` methods belonging to the connection object, which we obtain by calling the `ag_connect()` function described in *Example 1: Creating a repository and triple indices*.

Before asserting a triple, we have to generate the URI values for the subject, predicate and object fields. The AllegroGraph Python client API predefines a number of classes and predicates for the RDF, RDFS, XSD, and OWL ontologies. `RDF.TYPE` is one of the predefined predicates we will use.

The `add()` and `remove()` methods take an optional `contexts` argument that specifies one or more subgraphs that are the target of triple assertions and retractions. When the context is omitted, triples are asserted/retracted to/from the default graph. In the example below, facts about Alice and Bob reside in the default graph.

The second example begins by calling `ag_connect()` to create the appropriate connection object, which is bound to the variable `conn`.

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

The next step is to begin assembling the URIs we will need for the new triples. The `createURI()` method generates a URI from a string. These are the subject URIs identifying the resources "Bob" and "Alice":

```
alice = conn.createURI("http://example.org/people/alice")
bob = conn.createURI("http://example.org/people/bob")
```

Bob and Alice will be members of the "person" class (rdf type `person`).

```
person = conn.createURI("http://example.org/ontology/Person")
```

Both Bob and Alice will have a "name" attribute.

```
name = conn.createURI("http://example.org/ontology/name")
```

The name attributes will contain literal values. We have to generate the `Literal` objects from strings:

```
bobsName = conn.createLiteral("Bob")
alicesName = conn.createLiteral("Alice")
```

The next line prints out the number of triples currently in the repository - we expect that to be zero, since we have not yet added any triples and the `connect` function should have removed any existing statements from the repository.

```
print("Triple count before inserts:", conn.size())
```

```
Triple count before inserts: 0
```

Now we assert four triples, two for Bob and two more for Alice, using the connection object's `add()` method. After the assertions, we count triples again (there should be four) and print out the triples for inspection.

```
from franz.openrdf.vocabulary import RDF

# alice is a person
conn.add(alice, RDF.TYPE, person)
# alice's name is "Alice"
conn.add(alice, name, alicesName)
# bob is a person
conn.add(bob, RDF.TYPE, person)
# bob's name is "Bob":
conn.add(bob, name, bobsName)

print("Triple count:", conn.size())
for s in conn.getStatements(None, None, None, None):
    print(s)
```

The `None` arguments to the `getStatements()` method say that we don't want to restrict what values may be present in the subject, predicate, object or context positions. Just print out all the triples.

This is the output at this point. We see four triples, two about Alice and two about Bob

```
Triple count: 4
(<http://example.org/people/alice>, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
↪ <http://example.org/ontology/Person>)
(<http://example.org/people/alice>, <http://example.org/ontology/name>, "Alice")
(<http://example.org/people/bob>, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
↪<http://example.org/ontology/Person>)
(<http://example.org/people/bob>, <http://example.org/ontology/name>, "Bob")
```

We see two resources of type "person," each with a literal name.

The next step is to demonstrate how to remove a triple. Use the `remove()` method of the connection object, and supply a triple pattern that matches the target triple. In this case we want to remove Bob's name triple from the repository. Then we'll count the triples again to verify that there are only three remaining.

```
conn.remove(bob, name, bobsName)
print("Triple count:", conn.size())
```

```
Triple count: 3
```

A potentially less verbose way of adding triples is to use the `addData()` method of the connection object with a string containing triples in Turtle, N-Triples or another RDF format.

Let us see how the data used in this example could be added using `addData()`. We will also wrap the whole process in a function that we'll use later:

```
def add_bob_and_alice(conn):
   conn.addData("""
       @base <http://example.org/> .

       <people/alice> a <ontology/Person> ;
                      <ontology/name> "Alice" .
       <people/bob> a <ontology/Person> ;
                    <ontology/name> "Bob" .
   """)
```

The string used here is in the Turtle format. It is also possible to use other formats by passing the `rdf_format` argument to `addData()`.

We should check if the new function behaves as expected by creating a fresh connection (recall that the `clear` parameter causes all existing triples to be deleted):

```
with ag_connect('python-tutorial', clear=True) as conn:
    add_bob_and_alice(conn)
    print("Triple count:", conn.size())
```

```
Triple count: 4
```

2.7 Example 3: A SPARQL query

SPARQL stands for the "SPARQL Protocol and RDF Query Language," a recommendation of the World Wide Web Consortium (W3C) . SPARQL is a query language for retrieving RDF triples.

Our next example illustrates how to evaluate a SPARQL query. This is the simplest query, the one that returns all triples. Note that we will use the same triples that were used in *Example 2: Asserting and retracting triples*.

Let's create the connection first:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

And now we can add our data and define the query:

```
conn.addData("""
    @base <http://example.org/> .

    <people/alice> a <ontology/Person> ;
                   <ontology/name> "Alice" .
    <people/bob> a <ontology/Person> ;
                 <ontology/name> "Bob" .
""")
query_string = "SELECT ?s ?p ?o  WHERE {?s ?p ?o .}"
```

The `SELECT` clause returns the variables `?s`, `?p` and `?o` in the binding set. The variables are bound to the subject, predicate and objects values of each triple that satisfies the WHERE clause. In this case the WHERE clause is unconstrained. The dot (`.`) in the fourth position signifies the end of the pattern.

The connection object's `prepareTupleQuery()` method creates a query object that can be evaluated one or more times. The results are returned in an iterator that yields a sequence of binding sets.

```
from franz.openrdf.query.query import QueryLanguage

tuple_query = conn.prepareTupleQuery(QueryLanguage.SPARQL, query_string)
result = tuple_query.evaluate()
```

Below we illustrate one method for extracting the values from a binding set, indexed by the name of the corresponding column variable in the `SELECT` clause.

```
with result:
   for binding_set in result:
        s = binding_set.getValue("s")
        p = binding_set.getValue("p")
        o = binding_set.getValue("o")
        print("%s %s %s" % (s, p, o))
```

```
<http://example.org/people/bob> <http://example.org/ontology/name> "Bob"
<http://example.org/people/bob> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
↪<http://example.org/ontology/Person>
<http://example.org/people/alice> <http://example.org/ontology/name> "Alice"
<http://example.org/people/alice> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
↪<http://example.org/ontology/Person>
```

Note that we have wrapped the whole result processing in a `with` statement. The reason is that result objects must be closed after processing to release resources. The most convenient way to ensure this is the `with` statement, but it is also possible to explicitly call `close()` (e.g. in a `finally` block).

2.8 Example 4: Statement matching

The `getStatements()` method of the connection object provides a simple way to perform unsophisticated queries. This method lets you enter a mix of required values and wildcards, and retrieve all matching triples. (If you need to perform sophisticated tests and comparisons you should use a SPARQL query instead.)

Below, we illustrate two kinds of `getStatements()` calls. The first mimics traditional RDF4J syntax, and returns a `Statement` object at each iteration. We will reuse data from in previous examples to create a connection object and populate the repository with four triples describing Bob and Alice. We're going to search for triples that mention Alice, so we have to create an "Alice" URI to use in the search pattern:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

```
conn.addData("""
    @base <http://example.org/> .

    <people/alice> a <ontology/Person> ;
                   <ontology/name> "Alice" .
    <people/bob> a <ontology/Person> ;
                 <ontology/name> "Bob" .
""")
alice = conn.createURI("http://example.org/people/alice")
```

Now we search for triples with Alice's URI in the subject position. The `None` values are wildcards for the predicate and object positions of the triple.

```
statements = conn.getStatements(alice, None, None)
```

The `getStatements()` method returns a `RepositoryResult` object (bound to the variable `statements` in this case). This object can be iterated over, exposing one result statement at a time. It is sometimes desirable to screen the results for duplicates, using the `enableDuplicateFilter()` method. Note, however, that duplicate filtering can be expensive. Our example does not contain any duplicates, but it is possible for them to occur.

```
with statements:
    statements.enableDuplicateFilter()
    for statement in statements:
        print(statement)
```

This prints out the two matching triples for "Alice."

```
(<http://example.org/people/alice>, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
↪ <http://example.org/ontology/Person>)
(<http://example.org/people/alice>, <http://example.org/ontology/name>, "Alice")
```

Notice how we used the `with` keyword to ensure that the `RepositoryResult` object is closed after the results are fetched. This is necessary to release resources used during result retrieval. The same goal could be accomplished by calling the `RepositoryResult.close()` method (preferably in a `finally` block to ensure exception safety).

2.9 Example 5: Literal values

The next example illustrates some variations on what we have seen so far. The example creates and asserts plain, data-typed, and language-tagged literals, and then conducts searches for them in three ways:

- `getStatements()` search, which is an efficient way to match a single triple pattern.
- SPARQL direct match, for efficient multi-pattern search.
- SPARQL filter match, for sophisticated filtering such as performing range matches.

The `getStatements()` and SPARQL direct searches return exactly the datatype you ask for. The SPARQL filter queries can sometimes return multiple datatypes. This behavior will be one focus of this section.

If you are not explicit about the datatype of a value, either when asserting the triple or when writing a search pattern, AllegroGraph will deduce an appropriate datatype and use it. This is another focus of this section. This helpful behavior can sometimes surprise you with unanticipated results.

2.9.1 Setup

We begin by obtaining a connection object and remvoing all existing data from the repository

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

For the sake of coding efficiency, it is good practice to create variables for namespace strings. We'll use this namespace again and again in the following example. We have made the URIs in this example very short to keep the result displays compact.

```
exns = "ex://"
conn.setNamespace('ex', exns)
```

Namespace handling, including the `setNamespace()` method, is described in *Example 11: Namespaces*.

The example will use an artificial data set consisting of eight statements, each illustrating a different kind of literal. The subject will describe the nature of the literal used as the object, while the predicate will always be `<ex://p>`. The example shows how to enter a full URI string, or alternately how to combine a namespace with a local resource name.

```
ex_integer = conn.createURI("ex://integer")
ex_double = conn.createURI("ex://double")
ex_int = conn.createURI("ex://int")
ex_long = conn.createURI(
    namespace=exns, localname="long")
ex_float = conn.createURI(
    namespace=exns, localname="float")
ex_decimal = conn.createURI(
    namespace=exns, localname="decimal")
ex_string = conn.createURI(
    namespace=exns, localname="string")
ex_plain = conn.createURI(
    namespace=exns, localname="plain")
```

The predicate for all our statements will be the same.

```
pred = conn.createURI(namespace=exns, localname="p")
```

Now we construct the objects, illustrating various kinds of RDF literals.

```
from franz.openrdf.vocabulary.xmlschema import XMLSchema

# Type will be XMLSchema.INTEGER
```

(continues on next page)

(continued from previous page)

```
forty_two = conn.createLiteral(42)
# Type will be XMLSchema.DOUBLE
forty_two_double = conn.createLiteral(42.0)
forty_two_int = conn.createLiteral(
    '42', datatype=XMLSchema.INT)
forty_two_long = conn.createLiteral(
    '42', datatype=XMLSchema.LONG)
forty_two_float = conn.createLiteral(
    '42', datatype=XMLSchema.FLOAT)
forty_two_decimal = conn.createLiteral(
    '42', datatype=XMLSchema.DECIMAL)
forty_two_string = conn.createLiteral(
    '42', datatype=XMLSchema.STRING)
# Creates a plain (untyped) literal.
forty_two_plain = conn.createLiteral('42')
```

In four of these statements, we explicitly identified the datatype of the value in order to create an INT, a LONG, a FLOAT and a STRING. This is the best practice.

In three other statements, we just handed AllegroGraph numeric-looking values to see what it would do with them. As we will see in a moment, `42` creates an INTEGER, `42.0` becomes a DOUBLE, and `'42'` becomes a "plain" (untyped) literal value.

Warning: Note that plain literals are not *quite* the same thing as typed literal strings. A search for a plain literal will not always match a typed string, and *vice versa*.)

Now we will now assemble the URIs and values into `statements` (which are client-side triples):

```
stmt1 = conn.createStatement(ex_integer, pred, forty_two)
stmt2 = conn.createStatement(ex_double, pred, forty_two_double)
stmt3 = conn.createStatement(ex_int, pred, forty_two_int)
stmt4 = conn.createStatement(ex_long, pred, forty_two_long)
stmt5 = conn.createStatement(ex_float, pred, forty_two_float)
stmt6 = conn.createStatement(ex_decimal, pred, forty_two_decimal)
stmt7 = conn.createStatement(ex_string, pred, forty_two_string)
stmt8 = conn.createStatement(ex_plain, pred, forty_two_plain)
```

And then add the statements to the triple store on the AllegroGraph server. We can use either `add()` or `addStatement()` for this purpose.

```
conn.add(stmt1)
conn.add(stmt2)
conn.add(stmt3)
conn.addStatement(stmt4)
conn.addStatement(stmt5)
conn.addStatement(stmt6)
conn.addStatement(stmt7)
conn.addStatement(stmt8)
```

Now we'll complete the round trip to see what triples we get back from these assertions. This is where we use `getStatements()` in this example to retrieve and display triples for us:

```
print("Showing all triples using getStatements(). Eight matches.")
conn.getStatements(None, pred, None, output=True,)
```

This code prints out all triples from the store. The `output` parameter causes the result to be printed on `stdout` (it is also possible to pass a file name or a file-like object as the value of this parameter to print to other destinations). Without `output` the result would have been returned as a `RepositoryResult` object.

Note that the retrieved literals are of eight types: an int (a 32-bit integer), an integer (arbitrary precision), a decimal, a long, a float, a double, a string, and a "plain literal."

```
Showing all triples using getStatements(). Eight matches.
<ex://plain> <ex://p> "42" .
<ex://string> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#string> .
<ex://decimal> <ex://p> "42.0"^^<http://www.w3.org/2001/XMLSchema#decimal> .
<ex://float> <ex://p> "4.2E1"^^<http://www.w3.org/2001/XMLSchema#float> .
<ex://long> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#long> .
<ex://int> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#int> .
<ex://double> <ex://p> "4.2E1"^^<http://www.w3.org/2001/XMLSchema#double> .
<ex://integer> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

If you ask for a specific datatype, you will get it. If you leave the decision up to AllegroGraph, you might get something unexpected such as a plain literal value.

2.9.2 Numeric literal values

Matching 42 without explicit type

This section explores `getStatements()` and SPARQL matches against numeric triples. We ask AllegroGraph to find an untyped number, `42`.

```
print('getStatements():')
conn.getStatements(None, pred, 42, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p 42 .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery(
    'SELECT ?s ?p ?o WHERE {?s ?p ?o . filter (?o = 42)}',
    output=True)
print()
```

We use the `executeQuery()` method to retrieve the result of a SPARQL query. Like `getStatements()`, it accepts an `output` parameter that causes the result to be printed (instead of being returned as a `TupleQueryResult` object). Here is what the query methods discussed in this example would return:

```
getStatements():
<ex://integer> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#integer> .

SPARQL direct match
--------------
| s          |
==============
| ex:integer |
--------------
```

(continues on next page)

(continued from previous page)

```
SPARQL filter match
-----------------------------
| s           | p    | o     |
=============================
| ex:integer | ex:p | 42    |
| ex:double  | ex:p | 4.2E1 |
| ex:int     | ex:p | 42    |
| ex:long    | ex:p | 42    |
| ex:float   | ex:p | 4.2E1 |
| ex:decimal | ex:p | 42.0  |
-----------------------------
```

The `getStatements()` query returned triples containing longs only. The SPARQL direct match treated the numeric literal as if it had the type of `<http://www.w3.org/2001/XMLSchema#integer>` (see the SPARQL specification for information on how literals are parsed in queries) and returned only triples with exactly the same type. The SPARQL filter match, however, opened the doors to matches of multiple numeric types, and returned ints, floats, longs and doubles.

Matching 42.0 without explicit type

Now we will try the same queries using `42.0`.

```
print('getStatements():')
conn.getStatements(None, pred, 42.0, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s <ex://p> 42.0 .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery(
    'SELECT ?s ?p ?o WHERE {?s ?p ?o . filter (?o = 42.0)}',
    output=True)
print()
```

Here is what the query methods discussed in this example would return:

```
getStatements():
<ex://double> <ex://p> "4.2E1"^^<http://www.w3.org/2001/XMLSchema#double> .

SPARQL direct match
--------------
| s           |
==============
| ex:decimal |
--------------

SPARQL filter match
-----------------------------
| s           | p    | o     |
=============================
```

(continues on next page)

(continued from previous page)

```
| ex:integer | ex:p | 42    |
| ex:double  | ex:p | 4.2E1 |
| ex:int     | ex:p | 42    |
| ex:long    | ex:p | 42    |
| ex:float   | ex:p | 4.2E1 |
| ex:decimal | ex:p | 42.0  |
-----------------------------
```

The `getStatements()` search returned a double but not the similar float. Direct SPARQL match treated `42.0` as a decimal (in accordance with the SPARQL specification). The filter match returned all numeric types that were equal to 42.0.

Matching "42"^^xsd:int

The next section shows the results obtained when querying for a literal with explicitly specified type. Note that doing this with `getStatements()` requires passing in a `Literal` object, not a raw value.

```
print('getStatements():')
conn.getStatements(None, pred, forty_two_int, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p "42"^^xsd:int .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?p ?o WHERE {
       ?s ?p ?o .
       filter (?o = "42"^^xsd:int)
    }''',
    output=True)
print()
```

Here is what the query methods discussed in this example would return:

```
getStatements():
<ex://int> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#int> .

SPARQL direct match
----------
| s      |
==========
| ex:int |
----------

SPARQL filter match
-----------------------------
| s          | p    | o     |
=============================
| ex:integer | ex:p | 42    |
| ex:double  | ex:p | 4.2E1 |
| ex:int     | ex:p | 42    |
```

(continues on next page)

(continued from previous page)

```
| ex:long    | ex:p | 42    |
| ex:float   | ex:p | 4.2E1 |
| ex:decimal | ex:p | 42.0  |
-----------------------------
```

We would get similar results when asking for any other typed literal (`forty_two_long`, `forty_two_float`, ...).

2.9.3 Numeric strings and plain literals

At this point we are transitioning from tests of numeric matches to tests of string matches, but there is a gray zone to be explored first. What do we find if we search for strings that contain numbers? In particular, what about "plain literal" values that are almost, but not quite, strings?

Matching "42" as a typed string

Let's start with a typed string literal.

```
print('getStatements():')
conn.getStatements(None, pred, forty_two_string, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p "42"^^xsd:string .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?p ?o WHERE {
       ?s ?p ?o .
       filter (?o = "42"^^xsd:string)
    }''',
    output=True)
print()
```

Here are the results:

```
 getStatements():
 <ex://string> <ex://p> "42"^^<http://www.w3.org/2001/XMLSchema#string> .

 SPARQL direct match
 -------------
 | s         |
 =============
 | ex:plain  |
 | ex:string |
 -------------

 SPARQL filter match
 ------------------------------------------------------------------
 | s         | p    | o                                           |
 ==================================================================
```

(continues on next page)

(continued from previous page)

```
| ex:string | ex:p | 42^^http://www.w3.org/2001/XMLSchema#string |
| ex:plain  | ex:p | 42                                          |
-----------------------------------------------------------------
```

SPARQL matched both plain and literal strings, but a `getStatements()` search returned only typed matches. In both cases numeric literals were ignored.

Matching "42" as a plain literal

If we try to match a plain (untyped) string value

```
print('getStatements():')
conn.getStatements(None, pred, forty_two_plain, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p "42" .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?p ?o WHERE {
       ?s ?p ?o .
       filter (?o = "42")
    }''',
    output=True)
print()
```

We will get results consistent with that we saw in the typed case:

```
getStatements():
<ex://plain> <ex://p> "42" .

SPARQL direct match
-------------
| s         |
=============
| ex:plain  |
| ex:string |
-------------

SPARQL filter match
-----------------------------------------------------------------
| s         | p    | o                                         |
=================================================================
| ex:string | ex:p | 42^^http://www.w3.org/2001/XMLSchema#string |
| ex:plain  | ex:p | 42                                          |
-----------------------------------------------------------------
```

In SPARQL both kinds of string literals were matched, while `getStatements()` returned only direct matches.

2.9.4 Matching strings

In this section we'll set up a variety of string triples and experiment with matching them using `getStatements()` and SPARQL.

Note: *Example 12: Free Text indexing* is a different topic. In this section we're doing simple matches of whole strings.

Sample data

For these examples we will use a different data set.

```
name = conn.createURI('ex://name')
upper_g = conn.createLiteral('Galadriel')
lower_g = conn.createLiteral('galadriel')
typed_g = conn.createLiteral('Galadriel', XMLSchema.STRING)
lang_g = conn.createLiteral('Galadriel', language='sjn')
upper_a = conn.createLiteral('Artanis')
lower_a = conn.createLiteral('artanis')
typed_a = conn.createLiteral('Artanis', XMLSchema.STRING)
lang_a = conn.createLiteral('Artanis', language='qya')
conn.addTriple('<ex://upper_g>', name, upper_g)
conn.addTriple('<ex://lower_g>', name, lower_g)
conn.addTriple('<ex://typed_g>', name, typed_g)
conn.addTriple('<ex://lang_g>', name, lang_g)
conn.addTriple('<ex://upper_a>', name, upper_a)
conn.addTriple('<ex://lower_a>', name, lower_a)
conn.addTriple('<ex://typed_a>', name, typed_a)
conn.addTriple('<ex://lang_a>', name, lang_a)
```

We have two literals, each in four variants:

- Upper case (plain literal)
- Lower case (plain literal)
- Typed
- Tagged with a BCP47 language tag appropriate for its language (Quenya or Sindarin) according to the the registry

Matching a plain string

We've seen a similar case when looking at matches for `"42"`, but this time we have more similar literals in the store.

```
print('getStatements():')
conn.getStatements(None, name, upper_g, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s <ex://name> "Galadriel" .}',
    output=True)
print()
```

(continues on next page)

(continued from previous page)

```
print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s <ex://name> ?o .
       filter (?o = "Galadriel")
    }''',
    output=True)
print()
```

Here's the result:

```
getStatements():
<ex://upper_g> <ex://name> "Galadriel" .

SPARQL direct match
--------------
| s          |
==============
| ex:typed_g |
| ex:upper_g |
--------------

SPARQL filter match
----------------------------------------------------------------------
| s          | o                                                     |
======================================================================
| ex:typed_g | Galadriel^^http://www.w3.org/2001/XMLSchema#string |
| ex:upper_g | Galadriel                                             |
----------------------------------------------------------------------
```

We can see that the match is case-sensitive and ignores the language-tagged literal in all cases. As usual `getStatements()` matches only the exact kind of literal that we've provided, while SPARQL is more liberal.

Matching a language-tagged string

To retrieve the language-tagged variant we can ask for it explicitly:

```
print('getStatements():')
conn.getStatements(None, name, lang_g, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s <ex://name> "Galadriel"@sjn .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s <ex://name> ?o .
       filter (?o = "Galadriel"@sjn)
    }''',
    output=True)
print()
```

Unsurprisingly we get exactly what we have asked for

```
getStatements():
<ex://lang_g> <ex://name> "Galadriel"@sjn .

SPARQL direct match
-------------
| s         |
=============
| ex:lang_g |
-------------

SPARQL filter match
-----------------------------
| s         | o             |
=============================
| ex:lang_g | Galadriel@sjn |
-----------------------------
```

You may be wondering how to perform a string match where language and capitalization don't matter. You can do that with a SPARQL filter query using the `str()` function, which strips out the string portion of a literal, leaving behind the datatype or language tag. Then the `fn:lower-case()` function eliminates case issues:

```
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s <ex://name> ?o .
       filter (fn:lower-case(str(?o)) = "artanis")
    }''',
    output=True)
```

This query returns all variants of the selected literal

```
-------------------------------------------------------------
| s          | o                                            |
=============================================================
| ex:lang_a  | Artanis@qya                                  |
| ex:typed_a | Artanis^^http://www.w3.org/2001/XMLSchema#string |
| ex:lower_a | artanis                                      |
| ex:upper_a | Artanis                                      |
-------------------------------------------------------------
```

Remember that the SPARQL `filter` queries are powerful, but they are also the slowest queries. SPARQL direct queries and getStatements() queries are faster.

2.9.5 Booleans

Boolean values in SPARQL are represented by literals of type `<http://www.w3.org/2001/XMLSchema#boolean>`. There are two ways to create such literals in Python:

1. From corresponding Python boolean values (`True` and `FAlse`):

   ```
   true1 = conn.createLiteral(True)
   false1 = conn.createLiteral(False)
   ```

2. By creating a typed literal with the value of `"true"` or `"false"`. The type must be `xsd:boolean`:

```
true2 = conn.createLiteral("true", datatype=XMLSchema.BOOLEAN)
false2 = conn.createLiteral("false", datatype=XMLSchema.BOOLEAN)
```

Both ways of creating boolean literals produce equivalent results:

```
print(true1)
print(true2)
```

As we can see the literals are identical.

```
"true"^^<http://www.w3.org/2001/XMLSchema#boolean>
"true"^^<http://www.w3.org/2001/XMLSchema#boolean>
```

Let us add some boolean data to the store:

```
conn.addData("""
    <ex://f> <ex://p>
        "false"^^<http://www.w3.org/2001/XMLSchema#boolean> .
    # In Turtle 'true' is the same as '"true"^^xsd:boolean"'
    <ex://t> <ex://p> true .
""")
```

When querying for boolean values using SPARQL one can use the literals `true` and `false` as a shorthand for `"true"^^<http://www.w3.org/2001/XMLSchema#boolean>` and `"false"^^<http://www.w3.org/2001/XMLSchema#boolean>`. The code below illustrates various ways of querying for boolean values:

```
print('getStatements():')
conn.getStatements(None, None, true1, output=True)
print()

print('SPARQL direct match (true)')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p true.}',
    output=True)
print()

print('SPARQL direct match ("false"^^xsd:boolean)')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p "false"^^xsd:boolean .}',
    output=True)
print()

print('SPARQL filter match ("false"^^xsd:boolean)')
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s ?p ?o .
       filter (?o = "false"^^xsd:boolean)
    }''',
    output=True)
print()
```

Here's the output from that script:

```
getStatements():
<ex://t> <ex://p> "true"^^<http://www.w3.org/2001/XMLSchema#boolean> .

SPARQL direct match (true)
```

(continues on next page)

(continued from previous page)

```
--------
| s    |
========
| ex:t |
--------

SPARQL direct match ("false"^^xsd:boolean)
--------
| s    |
========
| ex:f |
--------

SPARQL filter match ("false"^^xsd:boolean)
----------------
| s    | o     |
================
| ex:f | false |
----------------
```

2.9.6 Dates and times

SPARQL represents dates and times using three literal types: `xsd:date`, `xsd:time` and `xsd:dateTime`. These can be created either explicitly from strings in the ISO 8601 format or from Python `datetime.date`, `datetime.time` and `datetime.datetime` values.

Let's create a few sample literals:

```
from datetime import date, time, datetime
import iso8601

d = conn.createLiteral(date(1944, 8, 1))
t = conn.createLiteral(time(15, 0, 0))
dt = conn.createLiteral('1944-08-01T17:00:00+02:00',
                        datatype=XMLSchema.DATETIME)
```

Creating `time` and `datetime` literals from Python values can yield somewhat unexpected results if time zones are involved:

```
surprise = conn.createLiteral(iso8601.parse_date(
    '1944-08-01T17:00:00+02:00'))
# Should be the same...
print(dt)
print(surprise)
```

The output is

```
"1944-08-01T17:00:00+02:00"^^<http://www.w3.org/2001/XMLSchema#dateTime>
"1944-08-01T15:00:00Z"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

The time has been converted to UTC. While both `dt` and `surprise` refer to the same moment in time, this conversion might still lead to problems if the user is not aware that it takes place.

We will now add the newly created literals to the store:

```
conn.addTriple('<ex://d>', '<ex://p>', d)
conn.addTriple('<ex://t>', '<ex://p>', t)
conn.addTriple('<ex://dt>', '<ex://p>', dt)
```

The following sections illustrate how date and time values behave during queries.

Matching dates

Let's try the usual mix of query methods and see what is returned:

```
print('getStatements():')
conn.getStatements(None, None, d, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p "1944-08-01"^^xsd:date .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s ?p ?o .
       filter (?o = "1944-08-01"^^xsd:date)
    }''',
    output=True)
print()
```

The result is not surprising. It is worth noting that the `datetime` value has not been returned, even though it refers to the same date.

```
 getStatements():
 <ex://d> <ex://p> "1944-08-01"^^<http://www.w3.org/2001/XMLSchema#date> .

 SPARQL direct match
 --------
 | s    |
 ========
 | ex:d |
 --------

 SPARQL filter match
 ----------------------
 | s    | o          |
 ======================
 | ex:d | 1944-08-01 |
 ----------------------
```

Matching times

Times can be queried in a similar fashion.

```
print('getStatements():')
conn.getStatements(None, None, t, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery(
    'SELECT ?s WHERE {?s ?p "15:00:00Z"^^xsd:time .}',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s ?p ?o .
       filter (?o = "15:00:00Z"^^xsd:time)
    }''',
    output=True)
print()
```

Again, only the value of the appropriate type is returned.

```
getStatements():
<ex://t> <ex://p> "15:00:00Z"^^<http://www.w3.org/2001/XMLSchema#time> .

SPARQL direct match
--------
| s    |
========
| ex:t |
--------

SPARQL filter match
--------------------
| s    | o         |
====================
| ex:t | 15:00:00Z |
--------------------
```

Matching datetimes

Datetimes work just like times and dates:

```
print('getStatements():')
conn.getStatements(None, None, dt, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery('''
    SELECT ?s WHERE {
       ?s ?p "1944-08-01T17:00:00+02:00"^^xsd:dateTime .
    }''',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
```

(continues on next page)

(continued from previous page)

```
    SELECT ?s ?o WHERE {
       ?s ?p ?o .
       filter (?o = "1944-08-01T17:00:00+02:00"^^xsd:dateTime)
    }''',
    output=True)
print()
```

The result:

```
getStatements():
<ex://dt> <ex://p> "1944-08-01T17:00:00+02:00"^^<http://www.w3.org/2001/XMLSchema
↪#dateTime> .

SPARQL direct match
---------
| s     |
=========
| ex:dt |
---------

SPARQL filter match
-------------------------------------
| s     | o                           |
=====================================
| ex:dt | 1944-08-01T17:00:00+02:00 |
-------------------------------------
```

Matching datetimes with offsets

We saw that times created from Python values are converted to UTC. So what happens when we query for Zulu time, while the value in the store is still in CEST?

```
zulu = conn.createLiteral("1944-08-01T15:00:00Z",
                          datatype=XMLSchema.DATETIME)
print('getStatements():')
conn.getStatements(None, None, zulu, output=True)
print()

print('SPARQL direct match')
conn.executeTupleQuery('''
    SELECT ?s WHERE {
       ?s ?p "1944-08-01T15:00:00Z"^^xsd:dateTime .
    }''',
    output=True)
print()

print('SPARQL filter match')
conn.executeTupleQuery('''
    SELECT ?s ?o WHERE {
       ?s ?p ?o .
       filter (?o = "1944-08-01T15:00:00Z"^^xsd:dateTime)
    }''',
    output=True)
print()
```

AllegroGraph still finds our value when using SPARQL

```
getStatements():

SPARQL direct match
---------
| s     |
=========
| ex:dt |
---------

SPARQL filter match
-------------------------------------
| s     | o                           |
=====================================
| ex:dt | 1944-08-01T17:00:00+02:00 |
-------------------------------------
```

When evaluating SPARQL queries AllegroGraph treats `datetime` objects that refer to the same point in time as equivalent, regardless of the timezone used in their representation. `getStatements()` performs exact matching, so will not return a value with different timezone.

2.10 Example 6: Importing triples

AllegroGraph can import files in multiple RDF `formats`, such as Turtle or N-Triples. The example below calls the connection object's `add()` method to load an N-Triples file, and `addFile()` to load an RDF/XML file. Both methods work, but the best practice is to use `addFile()`.

The `RDF/XML file` contains a short list of v-cards (virtual business cards), like this one:

```
<rdf:Description rdf:about="http://somewhere/JohnSmith/">
  <vCard:FN>John Smith</vCard:FN>
  <vCard:N rdf:parseType="Resource">
    <vCard:Family>Smith</vCard:Family>
    <vCard:Given>John</vCard:Given>
  </vCard:N>
</rdf:Description>
```

Save this file in `./data/vcards.rdf` (or choose another path and adjust the code below).

The `N-Triples file` contains a graph of resources describing the Kennedy family, the places where they were each born, their colleges, and their professions. A typical entry from that file looks like this:

```
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#first-name> "Joseph
↪" .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#middle-initial>
↪"Patrick" .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#last-name> "Kennedy
↪" .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#suffix> "none" .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#alma-mater> <http:/
↪/www.franz.com/simple#Harvard> .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#birth-year> "1888"␣
↪.
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#death-year> "1969"␣
↪.
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#sex> <http://www.
↪franz.com/simple#male> .
```

(continues on next page)

(continued from previous page)

```
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#spouse> <http://
↪www.franz.com/simple#person2> .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#has-child> <http://
↪www.franz.com/simple#person3> .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#profession> <http:/
↪/www.franz.com/simple#banker> .
<http://www.franz.com/simple#person1> <http://www.franz.com/simple#birth-place>
↪<http://www.franz.com/simple#place5> .
<http://www.franz.com/simple#person1> <http://www.w3.org/1999/02/22-rdf-syntax-ns
↪#type> <http://www.franz.com/simple#person> .
```

Save the file to `./data/kennedy.ntriples`.

Note that AllegroGraph can segregate triples into contexts (subgraphs) by treating them as quads, but the N-Triples and RDF/XML formats cannot include context information (unlike e.g N-Quads or Trig). They deal with triples only, so there is no place to store a fourth field in those formats. In the case of the `add()` call, we have omitted the context argument so the triples are loaded into the default graph (sometimes called the "null context.") The `addFile()` call includes an explicit context setting, so the fourth field of each VCard triple will be the context named `http://example.org#vcards`. The connection `size()` method takes an optional context argument. With no argument, it returns the total number of triples in the repository. Below, it returns the number `16` for the `context` context argument, and the number `28` for the null context (`None`) argument.

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

The variables `path1` and `path2` are bound to the RDF/XML and N-Triples files, respectively.

```
import os.path

# We assume that our data files live in this directory.
DATA_DIR = 'data'
path1 = os.path.join(DATA_DIR, 'vcards.rdf')
path2 = os.path.join(DATA_DIR, 'kennedy.ntriples')
```

The triples about the VCards will be added to a specific context, so naturally we need a URI to identify that context.

```
context = conn.createURI("http://example.org#vcards")
```

In the next step we use `addFile()` to load the VCard triples into the `#vcards` context:

```
from franz.openrdf.rio.rdfformat import RDFFormat

conn.addFile(path1, None, format=RDFFormat.RDFXML, context=context)
```

Then we use `add()` to load the Kennedy family tree into the default context:

```
conn.add(path2, base=None, format=RDFFormat.NTRIPLES, contexts=None)
```

Now we'll ask AllegroGraph to report on how many triples it sees in the default context and in the *#vcards* context:

```
print('VCard triples (in {context}): {count}'.format(
      count=conn.size(context), context=context))

print('Kennedy triples (default graph): {count}'.format(
      count=conn.size('null')))
```

The output of this report was:

```
VCard triples (in <http://example.org#vcards>): 16
Kennedy triples (default graph): 1214
```

2.11 Example 7: Querying multiple contexts

The purpose of this example is to see how data imported into multiple contexts (like that from *Example 6: Importing triples*) behaves when queried using various methods. This exampe covers only the results of basic queries. The subject is explored in more detail in *Example 10: Graphs in SPARQL*.

Let us start by creating a connection:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

and adding a few triples in the default context:

```
from franz.openrdf.query.query import QueryLanguage

conn.addData("""
   <ex://default1> <ex://p1> 1 .
   <ex://default2> <ex://p2> 2 .
   <ex://default3> <ex://p3> 3 .""")
```

We can add data to another contect by using the optional `context` parameter of `addData()`:

```
context = conn.createURI('ex://context')
conn.addData("""
   <ex://context1> <ex://p1> 1 .
   <ex://context2> <ex://p2> 2 .
   <ex://context3> <ex://p3> 3 .""",
   context=context)
```

Let's try a `getStatements()` call first:

```
p1 = conn.createURI('ex://p1')
with conn.getStatements(None, p1, None, None) as result:
    for row in result:
        print(row.getSubject())
```

This loop prints out a mix of triples from the default context and from the named context.

```
<ex://context1>
<ex://default1>
```

SPARQL queries behave in a different way. When a graph clause is present, as in the following code, triples that are not in a named context will not be examined:

```
query_string = """
    SELECT DISTINCT ?s WHERE {
      graph ?g { ?s ?p ?o filter(?o > 2).
    }} order by ?s"""
tuple_query = conn.prepareTupleQuery(
```

(continues on next page)

(continued from previous page)

```
    QueryLanguage.SPARQL, query_string)
with tuple_query.evaluate() as result:
    for bindings in result:
        print(bindings[0])
```

Only the `context3` triple is printed:

```
<ex://context3>
```

What happens if we issue a trivial query without mentioning `graph`?

```
query_string = """
    SELECT DISTINCT ?s WHERE {
      ?s ?p ?o .
    } order by ?s"""
tuple_query = conn.prepareTupleQuery(
    QueryLanguage.SPARQL, query_string)
with tuple_query.evaluate() as result:
    for bindings in result:
        print(bindings[0])
```

This prints all triples, just like a `getStatements()` call.

```
<ex://context1>
<ex://context2>
<ex://context3>
<ex://default1>
<ex://default2>
<ex://default3>
```

But this behavior can be altered by setting a query option. AllegroGraph allows such options to be set by defining a prefix.

```
query_string = """
    PREFIX franzOption_defaultDatasetBehavior: <franz:rdf>
    SELECT DISTINCT ?s WHERE {
      ?s ?p ?o .
    } order by ?s"""
tuple_query = conn.prepareTupleQuery(
    QueryLanguage.SPARQL, query_string)
with tuple_query.evaluate() as result:
    for bindings in result:
        print(bindings[0])
```

Now only the default context is matched by simple pattern (i.e. ones not wrapped in `graph ?g { ... }`)

```
<ex://default1>
<ex://default2>
<ex://default3>
```

2.12 Example 8: Exporting triples

This example shows how to serialize contents of a repository to a file. As usual we'll start with obtaining a connection to the repository:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

Now let's import some data:

```
conn.addData("""
  <ex://s> <ex://p1> <ex://o1> , <ex://o2> ;
           <ex://p2> <ex://o3> .""")
```

Data can be exported by passing a file name or a file-like object as the `output` parameter of `getStatements()`. In this case we'll want to print all statements to standard output. We can do this by passign `True` as the output file:

```
from franz.openrdf.rio.rdfformat import RDFFormat

conn.getStatements(output=True, output_format=RDFFormat.NTRIPLES)
```

We can see that results are printed in the specified format:

```
<ex://s> <ex://p1> <ex://o1> .
<ex://s> <ex://p1> <ex://o2> .
<ex://s> <ex://p2> <ex://o3> .
```

We can also use other arguments of `getStatements()` to constrain the set of exported tripes:

```
conn.getStatements(None, conn.createURI('ex://p1'), None,
                   output=True,
                   output_format=RDFFormat.NTRIPLES)
```

As expected, the result contains only two triples.

```
<ex://s> <ex://p1> <ex://o1> .
<ex://s> <ex://p1> <ex://o2> .
```

A file path can also be passed as the `output` argument:

```
import os
import sys

conn.getStatements(output='example8.nt')
with open('example8.nt', 'r') as f:
    sys.stdout.write(f.read())

os.remove('example8.nt')
```

This outputs data read from the file:

```
<ex://s> <ex://p2> <ex://o3> .
<ex://s> <ex://p1> <ex://o2> .
<ex://s> <ex://p1> <ex://o1> .
```

2.13 Example 9: Exporting query results

The *previous example* showed how to serialize statements to a file or a stream. It is also possible to perform a similar operation on the result of a query.

As usual, we'll start by opening a connection:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

and importing sample data - in this case containing birth and (when applicable) coronation dates of the sons of Henry II.

```
conn.addData("""
  @prefix : <ex://> .
  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

  :Henry :born "1155-02-28"^^xsd:date .
  :Richard :born "1157-09-08"^^xsd:date .
  :Geoffrey :born "1158-09-23"^^xsd:date .
  :John :born "1166-12-24"^^xsd:date .

  :Henry :crowned "1170-06-14"^^xsd:date .  # sort of...
  :Richard :crowned "1189-09-03"^^xsd:date .
  :John :crowned "1199-05-27"^^xsd:date .""")
```

Query results can be exported by passing a file name or a file-like object as the `output` parameter of the `TupleQuery.evaluate()` method of the query object. In this case we'll want to print all kings born in or after 1156 from our dataset to standard output (we can use `True` as the file name to indicate stdout):

```
from franz.openrdf.query.query import QueryLanguage
from franz.openrdf.rio.tupleformat import TupleFormat

query = conn.prepareTupleQuery(
    QueryLanguage.SPARQL,
    """
    select ?name ?crowned {
       ?name <ex://born> ?birth .
       ?name <ex://crowned> ?crowned .
       filter(?birth >= "1156-01-01"^^xsd:date) .
    }""")
query.evaluate(output=True,
               output_format=TupleFormat.CSV)
```

We can see that results are printed in the specified format:

```
name,crowned
"ex://Richard","1189-09-03"
"ex://John","1199-05-27"
```

We can export the result of a `CREATE` or `DESCRIBE` query in a similar fashion. The difference is that we need to supply an `RDFFormat` instead of a `TupleFormat`, since the result is a set of triples.

```
from franz.openrdf.rio.rdfformat import RDFFormat

query = conn.prepareGraphQuery(
    QueryLanguage.SPARQL, "describe <ex://Richard> where {}")
query.evaluate(output=True,
               output_format=RDFFormat.NTRIPLES)
```

As expected, the result contains two triples:

```
<ex://Richard> <ex://born> "1157-09-08"^^<http://www.w3.org/2001/XMLSchema#date> .
<ex://Richard> <ex://crowned> "1189-09-03"^^<http://www.w3.org/2001/XMLSchema#date> .
```

A file path can also be passed as the `output` argument:

```
import os
import sys

query = conn.prepareTupleQuery(
    QueryLanguage.SPARQL,
    """
    select ?name ?birth ?coronation {
      ?name <ex://born> ?birth ;
            <ex://crowned> ?coronation .
    }""")
query.evaluate(output='example9.csv',
               output_format=TupleFormat.CSV)
with open('example9.csv', 'r') as f:
    sys.stdout.write(f.read())

os.remove('example9.csv')
```

This outputs data read from the file:

```
name,birth,coronation
"ex://Henry","1155-02-28","1170-06-14"
"ex://Richard","1157-09-08","1189-09-03"
"ex://John","1166-12-24","1199-05-27"
```

2.14 Example 10: Graphs in SPARQL

In *Example 6: Importing triples* and *Example 7: Querying multiple contexts* we've seen how to import data to a non-default context and run queries against such data. In this example we'll explore facilities for handling multiple contexts provided by SPARQL and the AllegroGraph Python client.

We'll start by opening a connection:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

Now we will create two URIs that will represent named contexts.

```
context1 = conn.createURI("ex://context1")
context2 = conn.createURI("ex://context2")
```

The first context will be filled using the `addData()` method:

```
conn.addData("""
    @prefix : <ex://> .
    :alice a :person ;
           :name "Alice" .""",
    context=context1)
```

The second context will be filled using `addTriple()`. Notice how we use a constant defined in the `RDF` class to obtain the URI of the `type` predicate:

```
from franz.openrdf.vocabulary.rdf import RDF

bob = conn.createURI('ex://bob')
bob_name = conn.createLiteral('Bob')
name = conn.createURI('ex://person')
person = conn.createURI('ex://person')
conn.addTriple(bob, RDF.TYPE, person,
               contexts=[context2])
conn.addTriple(bob, name, bob_name,
               contexts=[context2])
```

Finally we'll add two triples to the default context using `addStatement()`:

```
from franz.openrdf.model import Statement

ted = conn.createURI('ex://ted')
ted_name = conn.createLiteral('Ted')
stmt1 = Statement(ted, name, ted_name)
stmt2 = Statement(ted, RDF.TYPE, person)
conn.addStatement(stmt1)
conn.addStatement(stmt2)
```

Warning: The `Statement` object contains a *context* field. This field is *ignored* by `addStatement()`. If you wish to add a statement object to a specific context, use the `contexts` parameter.

As we've seen already in *Example 7: Querying multiple contexts*, a call to `getStatements()` will return triples from all contexts:

```
with conn.getStatements() as result:
    print('getStatements(): {0}'.format(len(result)))
print('size(): {0}'.format(conn.size()))
```

`size()` will also process all contexts by default.

```
getStatements(): 6
size(): 6
```

Both `getStatements()` and `size()` accept a `contexts` parameter that can be used to limit processing to a specified list of graphs:

```
contexts = [context1, context2]
with conn.getStatements(contexts=contexts) as result:
    print('getStatements(): {0}'.format(len(result)))
print('size(): {0}'.format(conn.size(contexts=contexts)))
```

As expected, triples from the default context are not processed:

```
getStatements(): 4
size(): 4
```

To include the default graph when using the `contexts` parameter use `None` as a graph URI:

```
contexts = [context1, None]
with conn.getStatements(contexts=contexts) as result:
```

(continues on next page)

(continued from previous page)

```
    print('getStatements(): {0}'.format(len(result)))
print('size(): {0}'.format(conn.size(contexts=contexts)))
```

Now triples from the default context and from one of our named contexts are processed:

```
getStatements(): 4
size(): 4
```

2.14.1 SPARQL using FROM, FROM DEFAULT, and FROM NAMED

In many of our examples we have used a simple SPARQL query to retrieve triples from AllegroGraph's default graph. This has been very convenient but it is also misleading. As soon as we tell SPARQL to search a specific graph, we lose the ability to search AllegroGraph's default graph! Triples from the null graph vanish from the search results. Why is that?

It is important to understand that AllegroGraph and SPARQL use the phrase "default graph" to identify two very different things.

- AllegroGraph's default graph, or null context, is simply the set of all triples that have *null* in the fourth field of the "triple." The *default graph* is an unnamed subgraph of the AllegroGraph triple store.
- SPARQL uses *default graph* to describe something that is very different. In SPARQL, the *default graph* is a temporary pool of triples imported from one or more *named* graphs. SPARQL's *default graph* is constructed and discarded in the service of a single query. Standard SPARQL was designed for named graphs only, and has no syntax to identify a truly unnamed graph. AllegroGraph's SPARQL, however, has been extended to allow the unnamed graph to participate in multi-graph queries.

We can use AllegroGraph's SPARQL to search specific subgraphs in three ways.

- We can create a temporary *default graph* using the `FROM` operator.
- We can put AllegroGraph's unnamed graph into SPARQL's default graph using `FROM DEFAULT`.
- Or we can target specific named graphs using the `FROM NAMED` operator.

Here's an example of a query that accesses the unnamed graph explicitly:

```
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?s FROM DEFAULT {
        ?s ?p ?o
    }""")
query.evaluate(output=True)
```

This will not process any of the triples in named contexts:

```
------------
| s        |
============
| ex://ted |
------------
```

Here's an example of a query that uses `FROM`. It instructs SPARQL to regard `context1` as the default graph for the purposes of this query.

```
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?s FROM <ex://context1> {
        ?s ?p ?o
```

(continues on next page)

(continued from previous page)

```
    }""")
query.evaluate(output=True)
```

Now only one context is processed:

```
--------------
| s          |
==============
| ex://alice |
--------------
```

The next example changes `FROM` to `FROM NAMED` in the same query:

```
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?s FROM NAMED <ex://context1> {
        ?s ?p ?o
    }""")
query.evaluate(output=True)
```

There are no matches now! The pattern `{ ?s ?p ?o . }` only matches the SPARQL default graph. We declared `context1` to be a *named* graph, so it is no longer the default graph.

```
-----
| s |
=====
-----
```

To match triples in named graphs, SPARQL requires a `GRAPH` pattern:

```
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?s ?g FROM NAMED <ex://context1> {
        GRAPH ?g { ?s ?p ?o }
    }""")
query.evaluate(output=True)
```

This time we'll also print the graph:

```
------------------------------
| s          | g             |
==============================
| ex://alice | ex://context1 |
------------------------------
```

We can also combine all the forms presented above:

```
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?s ?g
    FROM DEFAULT
    FROM <ex://context1>
    FROM NAMED <ex://context2> {
        { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o } }
    }""")
query.evaluate(output=True)
```

This query puts AllegroGraph's unnamed graph and the `context1` graph into SPARQL's default graph, where the triples can be found by using a simple `{?s ?p ?o . }` query. Then it identifies `context2` as a named graph,

which can be searched using a GRAPH pattern. In the final line, we used a UNION operator to combine the matches of the simple and GRAPH patterns.

This query should find all three subjects:

```
------------------------------
| s           | g              |
==============================
| ex://alice  | ---            |
| ex://ted    | ---            |
| ex://bob    | ex://context2  |
------------------------------
```

2.14.2 SPARQL with Dataset object

A Dataset object is a construct that contains two lists of named graphs. There is one list of graphs that will become the SPARQL default graph, just like using FROM in the query. There is a second list of graphs that will be *named graphs* in the query, just like using FROM NAMED. To use the dataset, we put the graph URIs into the dataset object, and then add the dataset to the query object. When we evaluate the query, the results will be confined to the graphs listed in the dataset.

```
from franz.openrdf.query.dataset import Dataset

dataset = Dataset()
dataset.addDefaultGraph(context1)
dataset.addNamedGraph(context2)
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?s ?g {
      { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o } }
    }""")
query.setDataset(dataset)
query.evaluate(output=True)
```

Note that, since we're explicitly specifying graphs (through a dataset object), we need a GRAPH pattern to match triples from the named graphs. Triples from the unnamed graph are not matched at all, since that graph is not a part of the dataset.

```
------------------------------
| s           | g              |
==============================
| ex://alice  | ---            |
| ex://bob    | ex://context2  |
------------------------------
```

2.15 Example 11: Namespaces

A *namespace* is that portion of a URI that preceeds the last #, /, or : character, inclusive. The remainder of a URI is called the localname. For example, with respect to the URI http://example.org/people/alice, the namespace is http://example.org/people/ and the localname is alice. When writing SPARQL queries, it is convenient to define prefixes or nicknames for the namespaces, so that abbreviated URIs can be specified. For example, if we define ex to be a nickname for http://example.org/people/, then the string ex:alice is a recognized abbreviation for http://example.org/people/alice. This abbreviation is called a *qname* (qualified name).

In the SPARQL query discussed in this chapter we see two qnames, `rdf:type` and `ex:alice`. Ordinarily, we would expect to see `PREFIX` declarations in SPARQL that define namespaces for the `rdf` and `ex` nicknames. However, the connection and query machinery can do that job for you. The mapping of prefixes to namespaces includes the built-in prefixes `rdf`, `rdfs`, `xsd`, and `owl`. Hence, we can write `rdf:type` in a SPARQL query, and the system already knows its meaning. In the case of the `ex` prefix, we need to instruct it. The `setNamespace()` method of the connection object registers a new namespace.

Note: It is legal, although not recommended, to redefine the built-in prefixes (RDF, XSD etc. . .).

We start by opening a connection

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

and creating two URIs. Note how `createURI()` allows us to compose URIs from namespaces and local names.

```
exns = "http://example.org/people/"
alice = conn.createURI(namespace=exns, localname="alice")
person = conn.createURI(namespace=exns, localname="Person")
```

Now we can assert Alice's RDF:TYPE triple.

```
from franz.openrdf.vocabulary.rdf import RDF

conn.add(alice, RDF.TYPE, person)
```

Now we register the `exns` namespace with the connection object, so we can use it in a SPARQL query. The query looks for triples that have `rdf:type` in the predicate position, and `ex:Person` in the object position.

```
conn.setNamespace('ex', exns)
conn.executeTupleQuery("""
    SELECT ?s ?p ?o WHERE {
        ?s ?p ?o .
        FILTER (?p = rdf:type && ?o = ex:Person)
    }""", output=True)
```

The output shows the single triple that we expected to find. This demonstrates that the qnames in the SPARQL query successfully matched the fully-expanded URIs in the triple. Note that the namespace prefix is also used in the table below.

```
-----------------------------------
| s         | p          | o          |
===================================
| ex:alice | rdf:type | ex:Person |
-----------------------------------
```

It should be mentioned here that the prefix of a namespace can be an empty string. This allows the resulting qnames to be very concise and readable:

```
conn.setNamespace('', 'http://a-long-and-often-used-namespace/')
conn.executeUpdate('insert data { :this :looks :nice }')
conn.executeTupleQuery('select ?s { ?s :looks :nice }',
                       output=True)
```

```
---------
| s     |
=========
| :this |
---------
```

2.16 Example 12: Free Text indexing

It is common for users to build RDF applications that combine some form of "keyword search" with their queries. For example, a user might want to retrieve all triples for which the string "Alice" appears as a word within the third (object) field of the triple. AllegroGraph provides a capability for including free text matching within a SPARQL query, and also by using the `evalFreeTextSearch()` method of the connection object. It requires, however, that you create and configure indexes appropriate to the searches you want to pursue.

First let's open a connection

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

We will start this example by importing some sample data

```
conn.addData("""
    @prefix : <ex://> .

    :alice a :Person ;
             :fullname "Alice B. Toklas" .
    :book1 a :Book ;
             :title "Alice in Wonderland" ;
             :author :carroll .

    :carroll a :Person ;
               :fullname "Lewis Carroll" .""")
```

We have to create an index. AllegroGraph lets you create any number of text indexes, each for a specific purpose. In this case we are indexing the literal values we find in the `fullname` predicate, which we have used in resources that describe people. The `createFreeTextIndex()` method has many configurable parameters. Their default settings are appropriate to this situation. All we have to provide is a name for the index and the URI of the predicate (or predicates) that contain the text to be indexed.

```
fullname = conn.createURI(namespace='ex://',
                          localname='fullname')
conn.createFreeTextIndex(
    "index1", predicates=[fullname])
```

We can view the index configuration using the `getFreeTextIndexConfiguration()` method:

```
config = conn.getFreeTextIndexConfiguration("index1")
for key, value in config.items():
    if isinstance(value, list):
        value = ', '.join(str(x) for x in value)
    print('{key}: {value}'.format(key=key, value=value))
```

```
tokenizer: default
indexLiterals: True
minimumWordSize: 3
indexFields: object
stopWords: ...
innerChars:
predicates: <ex://fullname>
wordFilters:
indexResources: False
borderChars:
```

This configuration says that `index1` will operate on the literal values it finds in the object position of the `<ex://fullname>` predicate. It ignores words smaller than three characters in length. It will ignore the words in its `stopWords` list (elided from sample output). If it encounters a resource URI in the object position, it will ignore it. This index doesn't use any `wordFilters`, which are sometimes used to remove accented letters and to perform stemming on indexed text and search strings.

The text match occurs through a "magic" predicate called fti:match. This predicate has two arguments. One is the subject URI of the resources to search. The other is the string pattern to search for, such as "Alice". Only full-word matches will be found.

```
query = conn.prepareTupleQuery(query="""
    SELECT ?s WHERE {
        ?s fti:match "Alice" .
    }""")
query.evaluate(output=True)
```

There is no need to include a prefix declaration for the `fti` namespace. That is because `fti` is included among the built-in namespace mappings in AllegroGraph.

When we execute our SPARQL query, it matches the `"Alice"` within the literal `"Alice B. Toklas"` because that literal occurs in a triple having the `fullname` predicate, but it does not match the "Alice" in the literal `"Alice in Wonderland"` because the `title` predicate was not included in our index.

```
--------------
| s          |
==============
| ex://alice |
--------------
```

By default `fti:match` searches in all text indexes. It is possible to specify a single index name when searching. We'll illustrate this be creating another index, this time on the `title` predicate:

```
title = conn.createURI(namespace='ex://',
                       localname='title')
conn.createFreeTextIndex(
    "index2", predicates=[title])

query = conn.prepareTupleQuery(query="""
    SELECT ?s WHERE {
        ?s fti:match ( "Alice" "index2" ) .
    }""")
query.evaluate(output=True)
```

This time only the book title will match our query

```
--------------
| s          |
==============
| ex://book1 |
--------------
```

Another way of searching text indexes is the `evalFreeTextSearch()` method:

```
for triple in conn.evalFreeTextSearch(
        "Alice", index="index1"):
    print(triple[0])
```

This works just like our first query. Note that `evalFreeTextSearch()` returns a list of lists of strings (in N-Triples format), not a list of `Statement` objects.

```
<ex://alice>
```

The text index supports simple wildcard queries. The asterisk (`*`) may be appended to the end of the pattern to indicate "any number of additional characters." For instance, this query looks for whole words that begin with "Ali":

```
for triple in conn.evalFreeTextSearch("Ali*"):
    print(triple[0])
```

This search runs across both indexes, so it will find both the `:title` and the `:fullname` triples.

```
<ex://alice>
<ex://book1>
```

There is also a single-character wildcard, the question mark. It will match any single character. You can add as many question marks as you need to the string pattern. This query looks for a five-letter word that has "l" in the second position, and "c" in the fourth position:

```
for triple in conn.evalFreeTextSearch("?l?c?*"):
    print(triple[0])
```

The result is the same as for the previous query

```
<ex://alice>
<ex://book1>
```

Text indexes are not the only way of matching text values available in SPARQL. One may also filter results using regular expressions. This approach is more flexible, but at the price of performance. Regular expression filters do not use any form of indexing to speed up the query.

```
query = conn.prepareTupleQuery(query="""
    SELECT ?s ?p ?o WHERE {
        ?s ?p ?o .
        FILTER regex(?o, "lic|oll")
    }""")
query.evaluate(output=True)
```

Note how this search matches the provided pattern inside words.

```
-----------------------------------------------------
| s            | p             | o                  |
=====================================================
```

(continues on next page)

(continued from previous page)

```
| ex://carroll | ex://fullname | Lewis Carroll       |
| ex://book1   | ex://title    | Alice in Wonderland |
| ex://alice   | ex://fullname | Alice B. Toklas     |
-------------------------------------------------------
```

In addition to indexing literal values, AllegroGraph can also index resource URIs. `index3` is an index that looks for URIs in the object position of the `author` predicate, and then indexes only the local name of the resource (the characters following the rightmost `/`, `#` or `:` in the URI). This lets us avoid indexing highly-repetitive namespace strings, which would fill the index with data that would not be very useful.

```
author = conn.createURI(namespace='ex://',
                        localname='author')

conn.createFreeTextIndex(
    "index3", predicates=[author],
    indexResources="short", indexFields=["object"])

for triple in conn.evalFreeTextSearch("carroll",
                                      index="index3"):
    print(triple[0])
```

The text search located the triple that has `carroll` in the URI in the object position:

```
<ex://book1>
```

2.17 Example 13: SPARQL query forms

SPARQL provides alternatives to the standard SELECT query. This example exercises these alternatives to show how AllegroGraph Server and the Python client handle them.

Let's connect to the database:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

We'll need some sample data to illustrate all the query types. Our dataset will contain information about rulers of 17th century England.

```
conn.addData("""
    @prefix : <ex://> .
    @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

    :james_i :reigned_from "1603-03-24"^^xsd:date ;
             :reigned_to "1625-03-27"^^xsd:date .
    :charles_i :reigned_from "1625-03-27"^^xsd:date ;
               :reigned_to "1649-01-30"^^xsd:date ;
               :child_of :james_i .
    :charles_ii :reigned_from "1649-01-30"^^xsd:date ;
             :reigned_to "1685-02-06"^^xsd:date ;
             :child_of :charles_i .
    :james_ii :reigned_from "1685-02-06"^^xsd:date ;
             :reigned_to "1688-12-11"^^xsd:date ;
             :child_of :charles_i .
```

(continues on next page)

(continued from previous page)

```
    :mary_ii :reigned_from "1689-02-13"^^xsd:date ;
             :reigned_to "1694-12-28"^^xsd:date ;
             :child_of :james_ii .
    :william_iii :reigned_from "1672-07-04"^^xsd:date ;
                 :reigned_to "1702-03-08"^^xsd:date .
    :anne :reigned_from "1707-05-01"^^xsd:date ;
          :reigned_to "1714-08-01"^^xsd:date ;
          :child_of :james_ii .
""")
```

2.17.1 SELECT

This kind of query returns a sequence of tuples, binding variables to matching elements of a search pattern. `SELECT` queries are created using `prepareTupleQuery()` and return results of type `TupleQueryResult`. Query result can also be serialized in a supported `TupleFormat` - in previous examples we used `output=True` and relied on the default `TupleFormat.TABLE`.

Here's a sample query which locates all rulers whose grandchildren inherited the crown:

```
conn.setNamespace('', 'ex://')
query = conn.prepareTupleQuery(query="""
    SELECT DISTINCT ?name WHERE {
        ?grandchild :child_of/:child_of ?name .
    } ORDER BY ?name """)

with query.evaluate() as result:
    for bindings in result:
        print(bindings.getValue('name'))
```

Two names are returned:

```
<ex://charles_i>
<ex://james_i>
```

We can also serialize the output instead of processing the result object. This time let us reverse the query and ask for rulers whose grandparents are also in the dataset:

```
from franz.openrdf.rio.tupleformat import TupleFormat

query = conn.prepareTupleQuery(query="""
   SELECT DISTINCT ?name WHERE {
      ?name :child_of/:child_of ?grandparent .
   } ORDER BY ?name """)

query.evaluate(output=True, output_format=TupleFormat.CSV)
```

We get four results, serialized as CSV:

```
name
"ex://anne"
"ex://charles_ii"
"ex://james_ii"
"ex://mary_ii"
```

2.17.2 ASK

The `ASK` query returns a Boolean, depending on whether the triple pattern matched any triples. Queries of this type are created using `prepareBooleanQuery()`.

Let's check if there were any co-regencies in the time period described by our dataset:

```
query = conn.prepareBooleanQuery(query="""

    ASK { ?ruler1 :reigned_from ?r1from ;
                  :reigned_to ?r1to .
          ?ruler2 :reigned_from ?r2from ;
                  :reigned_to ?r2to .
          FILTER (?ruler1 != ?ruler2 &&
                  ?r1from >= ?r2from &&
                  ?r1from < ?r2to)
    }""")

print(query.evaluate())
```

There was one (William and Mary):

```
True
```

2.17.3 CONSTRUCT

The `CONSTRUCT` query creates triples by substantiating provided templates with values resulting from matching a pattern. Queries of this kind are created using `prepareGraphQuery()` and return a `RepositoryResult` - which is an iterator over the constructed triples.

Note: Executing a `CONSTRUCT` query will *not* add any triples to the store. To insert the data we have to iterate over the result and add each triple using `addStatement()` (or use an `INSERT` query).

Let us consider a query that calculates a `:sibling_of` relationship:

```
print('Size before: {0}'.format(conn.size()))
query = conn.prepareGraphQuery(query="""
   CONSTRUCT {
       ?person1 :sibling_of ?person2 .
   } WHERE {
       ?person1 :child_of ?parent .
       ?person2 :child_of ?parent .
       filter (?person1 != ?person2) .
   }""")
for stmt in query.evaluate():
    print('{0} <-> {1}'.format(stmt.getSubject(),
                               stmt.getObject()))
print('Size after: {0}'.format(conn.size()))
```

The returned object is an iterator over `Statement` objects. We can also see that no data has been added to the repository.

```
Size before: 19
<ex://james_ii> <-> <ex://charles_ii>
```

(continues on next page)

(continued from previous page)

```
<ex://charles_ii> <-> <ex://james_ii>
<ex://anne> <-> <ex://mary_ii>
<ex://mary_ii> <-> <ex://anne>
Size after: 19
```

We can also serialize the result using any of the supported `RDFFormats`:

```
from franz.openrdf.rio.rdfformat import RDFFormat

query.evaluate(output=True,
               output_format=RDFFormat.NTRIPLES)
```

Here we use the N-Triples format. This happens to be the default, so we could have omitted the `output_format` argument.

```
<ex://james_ii> <ex://sibling_of> <ex://charles_ii> .
<ex://charles_ii> <ex://sibling_of> <ex://james_ii> .
<ex://anne> <ex://sibling_of> <ex://mary_ii> .
<ex://mary_ii> <ex://sibling_of> <ex://anne> .
```

2.17.4 DESCRIBE

The `DESCRIBE` query returns triples that 'describe' a given set of resources. Such queries are created using `prepareGraphQuery()` and return `RepositoryResult` objects.

The set of resources to be processed is specified by a query pattern. The SPARQL standard does not say what triples constitute a 'description' of a particular resource. AllegroGraph will return the Concise Bounded Description of the queried resources.

Let's use a `DESCRIBE` query to see what data do we have regarding the children of Charles I:

```
query = conn.prepareGraphQuery(query="""
    DESCRIBE ?child WHERE {
        ?child :child_of :charles_i
    }""")
for stmt in query.evaluate():
    print(stmt)
```

In this case AllegroGraph will simply return all triples with subject in the specified set:

```
(<ex://charles_ii>, <ex://reigned_from>, "1649-01-30"^^<http://www.w3.org/2001/
↪XMLSchema#date>)
(<ex://charles_ii>, <ex://reigned_to>, "1685-02-06"^^<http://www.w3.org/2001/XMLSchema
↪#date>)
(<ex://charles_ii>, <ex://child_of>, <ex://charles_i>)
(<ex://james_ii>, <ex://reigned_from>, "1685-02-06"^^<http://www.w3.org/2001/XMLSchema
↪#date>)
(<ex://james_ii>, <ex://reigned_to>, "1688-12-11"^^<http://www.w3.org/2001/XMLSchema
↪#date>)
(<ex://james_ii>, <ex://child_of>, <ex://charles_i>)
```

DESCRIBE queries can be useful for exploring a dataset and learning what properties a certain object might have. The results of such queries can be serialized to any supported `RDFFormat`:

```
query.evaluate(output=True,
               output_format=RDFFormat.NTRIPLES)
```

```
<ex://charles_ii> <ex://reigned_from> "1649-01-30"^^<http://www.w3.org/2001/XMLSchema
↪#date> .
<ex://charles_ii> <ex://reigned_to> "1685-02-06"^^<http://www.w3.org/2001/XMLSchema
↪#date> .
<ex://charles_ii> <ex://child_of> <ex://charles_i> .
<ex://james_ii> <ex://reigned_from> "1685-02-06"^^<http://www.w3.org/2001/XMLSchema
↪#date> .
<ex://james_ii> <ex://reigned_to> "1688-12-11"^^<http://www.w3.org/2001/XMLSchema
↪#date> .
<ex://james_ii> <ex://child_of> <ex://charles_i> .
```

2.18 Example 14: Parametric queries

In previous examples our SPARQL queries were always fixed strings. In practice it is often necessary to include some variable elements (e.g. user input, results from another query, . . .) in the query strings.

To illustrate, let us create a connection

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

and populate the repository with sample data:

```
conn.addData(r"""
  @prefix : <ex://> .

  :cipher42 :label "RSA" ; :code 1 .
  :hedge1 :label "\\/\\/\\/\\/\\/\\/\\" ; :code 2 .
  :hedge2 :label "/\\/\\/\\/\\/\\/\\//" .  # No code
  :has_no_label :secret "squeamish ossifrage" .
""")
```

Suppose that we need a function that will take a search text and find all subjects that have a label containing a given search pattern. To make the query a little more interesting we'll also print the value of the *:code* predicate for all subjects found.

```
conn.setNamespace('', 'ex://')
query = conn.prepareTupleQuery(query="""
    SELECT ?s ?code WHERE {
       ?s :label ?o .
       FILTER contains(?o, ?search)
       OPTIONAL { ?s <ex://code> ?code } .
    }""")

def print_labelled_subjects(search_text):
   query.setBinding('search', search_text)
   query.evaluate(output=True)
```

We have created a query object with a variable (`?search`) in place of the search pattern. To use it we have to provide a value for the variable. We do that using the `setBinding()` method.

Let's check if our function works as expected:

```
print_labelled_subjects(r'\/\/')
```

This will return the 'hedge' subjects:

```
------------------
| s       | code |
==================
| :hedge1 | 2    |
| :hedge2 | ---  |
------------------
```

2.18.1 String formatting

Another way to achieve our goal would be to use formatting or string concatenation, like this:

```
import sys

conn.setNamespace('', 'ex://')
def print_labelled_subjects(search_text):
    query = conn.prepareTupleQuery(query="""
        SELECT ?s ?code WHERE {
            ?s :label ?o .
            FILTER contains(?o, "%s")
            OPTIONAL { ?s <ex://code> ?code } .
        }""" % search_text)
    query.evaluate(output=True)

print_labelled_subjects('RS')
```

This seems to work

```
--------------------
| s         | code |
====================
| :cipher42 | 1    |
--------------------
```

But attempting to use a trickier input reveals a problem:

```
print_labelled_subjects(r'\/\/')
```

The query is now invalid

```
Traceback (most recent call last):
  ...
RequestError: Server returned 400: ...
```

A devious user could take advantage of this bug to access data that is not supposed to be available

```
print_labelled_subjects(
    r'S") optional { ?x <ex://secret> ?code } # ')
```

It should not be possible to reveal this literal by searching labels, and yet:

```
-----------------------------------
| s         | code                |
===================================
| :cipher42 | squeamish ossifrage |
-----------------------------------
```

We can work around this by ensuring proper escaping:

```
def print_labelled_subjects(search_text):
    search_lit = conn.createLiteral(search_text)
    query = conn.prepareTupleQuery(query="""
        SELECT ?s ?code WHERE {
            ?s :label ?o .
            FILTER contains(?o, %s)
            OPTIONAL { ?s <ex://code> ?code } .
        }""" % search_lit.toNTriples())
    query.evaluate(output=True)

print_labelled_subjects(r'\/\/')
```

The function now works as expected:

```
------------------
| s       | code |
==================
| :hedge1 | 2    |
| :hedge2 | ---  |
------------------
```

2.19 Example 15: Range queries

In many of the previous examples we have used the `getStatements()` method to find all triples conforming to a given pattern. The patterns we have used so far matched each triple component against a single value. It is possible to use more complex patterns that can match a range of values for each component. To illustrate this let us first create a connection:

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

and construct some data:

```
conn.addData("""
    @prefix : <ex://> .

    :mercury a :planet ; :moons 0 .
    :venus a :planet ; :moons 0 .
    :earth a :planet ; :moons 1 .
    :mars a :planet ; :moons 2 .
    :jupiter a :planet ; :moons 67 .
    :saturn a :planet ; :moons 62 .
    :uranus a :planet ; :moons 27 .
    :neptune a :planet ; :moons 14 .
    :pluto a :dwarf_planet ; :moons 5 .
""")
```

Suppose that we want to locate all planets that have at least one, but no more than five moons. To issue such a query we need to create a `Range` object:

```
one_to_five = conn.createRange(1, 5)
```

We can pass the range object to `getStatements()`:

```
moons = conn.createURI('ex://moons')
with conn.getStatements(
        None, moons, one_to_five) as result:
    for statement in result:
        print(statement.getSubject())
```

This will find two planets and one dwarf planet, as expected:

```
<ex://earth>
<ex://mars>
<ex://pluto>
```

The arguments to `createRange()` can be either RDF terms or regular Python values that will be converted to typed `literals`. In our example we used have used values of type `int`, which will be mapped to literals of type `<http://www.w3.org/2001/XMLSchema#integer>`. Range queries will only match values of exactly the same type. For instance if we add another triple to our store:

```
conn.addData("""
    @prefix : <ex://> .
    @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

    :coruscant a :planet ; :moons "4"^^xsd:long .
""")
```

And then reissue our query:

```
with conn.getStatements(
        None, moons, one_to_five) as result:
    for statement in result:
        print(statement.getSubject())
```

we will find that the result has not changed:

```
<ex://earth>
<ex://mars>
<ex://pluto>
```

Range queries can also be performed with SPARQL, using `FILTER`:

```
conn.executeTupleQuery('''
    SELECT ?planet {
        ?planet <ex://moons> ?moons .
        filter (?moons <= 5 && ?moons >= 1)
    }''', output=True)
```

The result is the same as in the previous example.

```
------------------
| planet         |
==================
| ex://coruscant |
| ex://earth     |
| ex://mars      |
| ex://pluto     |
------------------
```

When the filter expression is a simple set of inequalities, as it is in this case, the query engine will use indices to optimize the query execution, similaraly to the way `getStatements()` does for range queries.

2.20 Example 16: Federated repositories

AllegroGraph lets you split up your triples among repositories on multiple servers and then search them all in parallel. To do this we query a single "federated" repository that automatically distributes the queries to the secondary repositories and combines the results. From the point of view of your Python code, it looks like you are working with a single repository.

To illustrate this, let us first create two repositories and import some data. The data will represent positive numbers below 15. The first repository will contain all Fibonacci numbers in that range, while the second one will contain all other numbers.

```
from franz.openrdf.connect import ag_connect

with ag_connect('python_fib', create=True, clear=True) as conn:
    conn.addData("""
        @prefix : <ex://> .

        :one :value 1 .
        :two :value 2 .
        :three :value 3 .
        :five :value 5 .
        :eight :value 8 .
        :thirteen :value 13 .
    """)

with ag_connect('python_boring', create=True, clear=True) as conn:
    conn.addData("""
        @prefix : <ex://> .

        :four :value 4 .
        :six :value 6 .
        :seven :value 7 .
        :nine :value 9 .
        :ten :value 10 .
        :eleven :value 11 .
        :twelve :value 12 .
        :fourteen :value 14 .
        :fifteen :value 15 .
    """)
```

To create a federated repository, we first have to connect to the server that will be used to aggregate results. We do this by creating an `AllegroGraphServer` instance.

```
from franz.openrdf.sail.allegrographserver import AllegroGraphServer

server = AllegroGraphServer()
```

We are using default server address and credentials, as described in the *Setting the environment for the tutorial* section of the tutorial.

The next step is to use the `openFederated()` method to create a federated session. We will pass the list of repositories to federate as an argument. Elements of this list could be

- `Repository` objects
- `RepositoryConnection` objects
- strings (naming a store in the root catalog, or the URL of a store)

- (storename, catalogname) tuples.

We'll use the third option

```
conn = server.openFederated(['python_fib', 'python_boring'])
```

Now we can query the combined repository.

```
query = conn.prepareTupleQuery(query="""
    select (avg(?v) as ?avg)
           (min(?v) as ?min)
           (max(?v) as ?max) where {
       ?number <ex://value> ?v .
    }""")
query.evaluate(output=True)
```

As we can see, data from both repositories has been returned and aggregates have been correctly computed over the whole dataset.

```
--------------------
| avg | min | max |
====================
| 8.0 | 1   | 15  |
--------------------
```

Another example of using federated repositories, this time with multiple server machines, can be found in *Running AG on AWS EC2*.

2.21 Example 17: Triple attributes

Triples offer a way of describing model elements and relationships between them. In come cases, however, it is also convenient to be able to store data that is associated with a triple as a whole rather than with a particular element. For instance one might wish to record the source from which a triple has been imported or access level necessary to include it in query results. Traditional solutions of this problem include using graphs, RDF reification or triple IDs. All of these approaches suffer from various flexibility and performance issues. For this reason AllegroGraph offers an alternative: triple attributes.

Attributes are key-value pairs associated with a triple. Keys refer to attribute definitions that must be added to the store before they are used. Values are strings. The set of legal values of an attribute can be constrained by the definition of that attribute. It is possible to associate multiple values of a given attribute with a single triple.

Possible uses for triple attributes include:

- Access control: It is possible to instruct AllegroGraph to prevent an user from accessing triples with certain attributes.
- Sharding: Attributes can be used to ensure that related triples are always placed in the same shard when AllegroGraph acts as a distributed triple store.

Like all other triple components, attribute values are immutable. They must be provided when the triple is added to the store and cannot be changed or removed later.

To illustrate the use of triple attributes we will construct an artificial data set containing a log of information about contacts detected by a submarine at a single moment in time.

2.21.1 Managing attribute definitions

Before we can add triples with attributes to the store we must create appropriate attribute definitions.

First let's open a connection

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

Attribute definitions are represented by `AttributeDefinition` objects. Each definition has a name, which must be unique, and a few optional properties (that can also be passed as constructor arguments):

- `allowed_values`: a list of strings. If this property is set then only the values from this list can be used for the defined attribute.
- `ordered`: a boolean. If true then attribute value comparisons will use the ordering defined by `allowed_values`. The default is false.
- `minimum_number`, `maximum_number`: integers that can be used to constrain the cardinality of an attribute. By default there are no limits.

Let's define a few attributes that we will later use to demonstrate various attribute-related capabilities of AllegroGraph. To do this, we will use the `setAttributeDefinition()` method of the connection object.

```
from franz.openrdf.repository.attributes import AttributeDefinition

# A simple attribute with no constraints governing the set
# of legal values or the number of values that can be
# associated with a triple.
tag = AttributeDefinition(name='tag')

# An attribute with a limited set of legal values.
# Every bit of data can come from multiple sources.
# We encode this information in triple attributes,
# since it refers to the tripe as a whole. Another
# way of achieving this would be to use triple ids
# or RDF reification.
source = AttributeDefinition(
    name='source',
    allowed_values=['sonar', 'radar', 'esm', 'visual'])

# Security level - notice that the values are ordered
# and each triple *must* have exactly one value for
# this attribute. We will use this to prevent some
# users from accessing classified data.
level = AttributeDefinition(
    name='level',
    allowed_values=['low', 'medium', 'high'],
    ordered=True,
    minimum_number=1,
    maximum_number=1)

# An attribute like this could be used for sharding.
# That would ensure that data related to a particular
# contact is never partitioned across multiple shards.
# Note that this attribute is required, since without
# it an attribute-sharded triple store would not know
# what to do with a triple.
```

(continues on next page)

(continued from previous page)

```
contact = AttributeDefinition(
    name='contact',
    minimum_number=1,
    maximum_number=1)

# So far we have created definition objects, but we
# have not yet sent those definitions to the server.
# Let's do this now.
conn.setAttributeDefinition(tag)
conn.setAttributeDefinition(source)
conn.setAttributeDefinition(level)
conn.setAttributeDefinition(contact)

# This line is not strictly necessary, because our
# connection operates in autocommit mode.
# However, it is important to note that attribute
# definitions have to be committed before they can
# be used by other sessions.
conn.commit()
```

It is possible to retrieve the list of attribute definitions from a repository by using the `getAttributeDefinitions()` method:

```
for attr in conn.getAttributeDefinitions():
    print('Name: {0}'.format(attr.name))
    if attr.allowed_values:
        print('Allowed values: {0}'.format(
            ', '.join(attr.allowed_values)))
        print('Ordered: {0}'.format(
            'Y' if attr.ordered else 'N'))
    print('Min count: {0}'.format(attr.minimum_number))
    print('Max count: {0}'.format(attr.maximum_number))
    print()
```

Notice that in cases where the maximum cardinality has not been explicitly defined, the server replaced it with a default value. In practice this value is high enough to be interpreted as 'no limit'.

```
Name: tag
Min count: 0
Max count: 1152921504606846975

Name: source
Allowed values: sonar, radar, esm, visual
Min count: 0
Max count: 1152921504606846975
Ordered: N

Name: level
Allowed values: low, medium, high
Ordered: Y
Min count: 1
Max count: 1

Name: contact
Min count: 1
Max count: 1
```

Attribute definitions can be removed (provided that the attribute is not used by the static attribute filter, which will be discussed later) by calling `deleteAttributeDefinition()`:

```
conn.deleteAttributeDefinition('tag')
defs = conn.getAttributeDefinitions()
print(', '.join(sorted(a.name for a in defs)))
```

```
contact, level, source
```

2.21.2 Adding triples with attributes

Now that the attribute definitions have been established we can demonstrate the process of adding triples with attributes. This can be achieved using various methods. A common element of all these methods is the way in which triple attributes are represented. In all cases dictionaries with attribute names as keys and strings or lists of strings as values are used.

When `addTriple()` is used it is possible to pass attributes in a keyword parameter, as shown below:

```
ex = conn.namespace('ex://')
conn.addTriple(ex.S1, ex.cls, ex.Udaloy, attributes={
    'source': 'sonar',
    'level': 'low',
    'contact': 'S1'
})
```

The `addStatement()` method works in similar way. Note that it is not possible to include attributes in the `Statement` object itself.

```
from franz.openrdf.model import Statement

s = Statement(ex.M1, ex.cls, ex.Zumwalt)
conn.addStatement(s, attributes={
    'source': ['sonar', 'esm'],
    'level': 'medium',
    'contact': 'M1'
})
```

When adding multiple triples with `addTriples()` one can add a fifth element to each tuple to represent attributes. Let us illustrate this by adding an aircraft to our dataset.

```
conn.addTriples(
    [(ex.R1, ex.cls, ex['Ka-27'], None,
      {'source': 'radar',
       'level': 'low',
       'contact': 'R1'}),
     (ex.R1, ex.altitude, 200, None,
      {'source': 'radar',
       'level': 'medium',
       'contact': 'R1'})])
```

When all or most of the added triples share the same attribute set it might be convenient to use the `attributes` keyword parameter. This provides default values, but is completely ignored for all tuples that already contain attributes (the dictionaries are not merged). In the example below we add a triple representing an aircraft carrier and a few more triples that specify its position. Notice that the first triple has a lower security level and multiple sources. The common 'contact' attribute could be used to ensure that all this data will remain on a single shard.

```
conn.addTriples(
    [(ex.M2, ex.cls, ex.Kuznetsov, None, {
        'source': ['sonar', 'radar', 'visual'],
        'contact': 'M2',
        'level': 'low',
     }),
     (ex.M2, ex.position, ex.pos343),
     (ex.pos343, ex.x, 430.0),
     (ex.pos343, ex.y, 240.0)],
    attributes={
       'contact': 'M2',
       'source': 'radar',
       'level': 'medium'
    })
```

Another method of adding triples with attributes is to use the NQX file format. This works both with `addFile()` and `addData()` (illustrated below):

```
from franz.openrdf.rio.rdfformat import RDFFormat

conn.addData('''
    <ex://S2> <ex://cls> <ex://Alpha> \
    {"source": "sonar", "level": "medium", "contact": "S2"} .
    <ex://S2> <ex://depth> "300" \
    {"source": "sonar", "level": "medium", "contact": "S2"} .
    <ex://S2> <ex://speed_kn> "15.0" \
    {"source": "sonar", "level": "medium", "contact": "S2"} .
''', rdf_format=RDFFormat.NQX)
```

When importing from a format that does not support attributes, it is possible to provide a common set of attribute values with a keyword parameter:

```
from franz.openrdf.rio.rdfformat import RDFFormat

conn.addData('''
    <ex://V1> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 100 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V2> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 200 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V3> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 300;
              <ex://speed_kn> 12.0e+8 .
    <ex://V4> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 400 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V5> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 500 ;
              <ex://speed_kn> 12.0e+8 .
    <ex://V6> <ex://cls> <ex://Walrus> ;
              <ex://altitude> 600 ;
              <ex://speed_kn> 12.0e+8 .
''', attributes={
    'source': 'visual',
    'level': 'high',
    'contact': 'a therapist'})
```

The data above represents six visually observed Walrus-class submarines, flying at different altitudes and well above the speed of light. It has been highly classified to conceal the fact that someone has clearly been drinking while on duty - after all there are only four Walrus-class submarines currently in service, so the observation is obviously incorrect.

2.21.3 Retrieving attribute values

We will now print all the data we have added to the store, including attributes, to verify that everything worked as expected. The only way to do that is through a SPARQL query using the appropriate magic property to access the attributes. The query below binds a literal containing a JSON representation of triple attributes to the *?a* variable:

```
import json

r = conn.executeTupleQuery('''
   PREFIX attr: <http://franz.com/ns/allegrograph/6.2.0/>
   SELECT ?s ?p ?o ?a {
       ?s ?p ?o .
       ?a attr:attributes (?s ?p ?o) .
   } ORDER BY ?s ?p ?o''')
with r:
    for row in r:
        print(row['s'], row['p'], row['o'])
        print(json.dumps(json.loads(row['a'].label),
                         sort_keys=True,
                         indent=4))
```

The result contains all the expected triples with pretty-printed attributes.

```
<ex://M1> <ex://cls> <ex://Zumwalt>
{
    "contact": "M1",
    "level": "medium",
    "source": [
        "esm",
        "sonar"
    ]
}
<ex://M2> <ex://cls> <ex://Kuznetsov>
{
    "contact": "M2",
    "level": "low",
    "source": [
        "visual",
        "radar",
        "sonar"
    ]
}
<ex://M2> <ex://position> <ex://pos343>
{
    "contact": "M2",
    "level": "medium",
    "source": "radar"
}
<ex://R1> <ex://altitude> "200"^^...
{
    "contact": "R1",
    "level": "medium",
```

(continues on next page)

(continued from previous page)

```
    "source": "radar"
}
<ex://R1> <ex://cls> <ex://Ka-27>
{
    "contact": "R1",
    "level": "low",
    "source": "radar"
}
<ex://S1> <ex://cls> <ex://Udaloy>
{
    "contact": "S1",
    "level": "low",
    "source": "sonar"
}
<ex://S2> <ex://cls> <ex://Alpha>
{
    "contact": "S2",
    "level": "medium",
    "source": "sonar"
}
<ex://S2> <ex://depth> "300"
{
    "contact": "S2",
    "level": "medium",
    "source": "sonar"
}
<ex://S2> <ex://speed_kn> "15.0"
{
    "contact": "S2",
    "level": "medium",
    "source": "sonar"
}
<ex://V1> <ex://altitude> "100"^^...
{
    "contact": "a therapist",
    "level": "high",
    "source": "visual"
}
<ex://V1> <ex://cls> <ex://Walrus>
{
    "contact": "a therapist",
    "level": "high",
    "source": "visual"
}
<ex://V1> <ex://speed_kn> "1.2E9"^^...
{
    "contact": "a therapist",
    "level": "high",
    "source": "visual"
}
...
<ex://pos343> <ex://x> "4.3E2"^^...
{
    "contact": "M2",
    "level": "medium",
    "source": "radar"
}
```

(continues on next page)

(continued from previous page)

```
<ex://pos343> <ex://y> "2.4E2"^^...
{
    "contact": "M2",
    "level": "medium",
    "source": "radar"
}
```

2.21.4 Attribute filters

Triple attributes can be used to provide fine-grained access control. This can be achieved by using static attribute filters.

Static attribute filters are simple expressions that control which triples are visible to a query based on triple attributes. Each repository has a single, global attribute filter that can be modified using `setAttributeFilter()`. The values passed to this method must be either strings (the syntax is described in the documentation of static attribute filters) or filter objects.

Filter objects are created by applying set operators to 'attribute sets'. These can then be combined using filter operators.

An attribute set can be one of the following:

- a string or a list of strings: represents a constant set of values.
- *TripleAttribute.name*: represents the value of the *name* attribute associated with the currently inspected triple.
- *UserAttribute.name*: represents the value of the *name* attribute associated with current query. User attributes will be discussed in more detail later.

Available set operators are shown in the table below. All classes and functions mentioned here can be imported from the `franz.openrdf.repository.attributes` package:

Syntax	Meaning
`Empty(x)`	True if the specified attribute set is empty.
`Overlap(x, y)`	True if there is at least one matching value between the two attribute sets.
`Subset(x, y)`, `x << y`	True if every element of *x* can be found in *y*
`Superset(x, y)`, `x >> y`	True if every element of *y* can be found in *x*
`Equal(x, y)`, `x == y`	True if *x* and *y* have exactly the same contents.
`Lt(x, y)`, `x < y`	True if both sets are singletons, at least one of the sets refers to a triple or user attribute, the attribute is ordered and the value of the single element of *x* occurs before the single value of *y* in the `lowed_values` list of the attribute.
`Le(x, y)`, `x <= y`	True if $y < x$ is false.
`Eq(x, y)`	True if both $x < y$ and $y < x$ are false. Note that using the == Python operator translates to *Eqauls*, not *Eq*.
`Ge(x, y)`, `x >= y`	True if $x < y$ is false.
`Gt(x, y)`, `x > y`	True if $y < x$.

Note that the overloaded operators only work if at least one of the attribute sets is a `UserAttribute` or `TripleAttribute` reference - if both arguments are strings or lists of strings the default Python semantics for each operator are used. The prefix syntax always produces filters.

Filters can be combined using the following operators:

Syntax	Meaning
`Not(x), ~x`	Negates the meaning of the filter.
`And(x, y, ...), x & y`	True if all subfilters are true.
`Or(x, y, ...), x | y`	True if at least one subfilter is true.

Filter operators also work with raw strings, but overloaded operators will only be recognized if at least one argument is a filter object.

2.21.5 Using filters and user attributes

The example below displays all classes of vessels from the dataset after establishing a static attribute filter which ensures that only sonar contacts are visible:

```
from franz.openrdf.repository.attributes import *

conn.setAttributeFilter(TripleAttribute.source >> 'sonar')
conn.executeTupleQuery(
    'select ?class { ?s <ex://cls> ?class } order by ?class',
    output=True)
```

The output contains neither the visually observed Walruses nor the radar detected ASW helicopter.

```
------------------
| class          |
==================
| ex://Alpha     |
| ex://Kuznetsov |
| ex://Udaloy    |
| ex://Zumwalt   |
------------------
```

To avoid having to set a static filter before each query (which would be inefficient and cause concurrency issues) we can employ user attributes. User attributes are specific to a particular connection and are sent to the server with each query. The static attribute filter can refer to these and compare them with triple attributes. Thus we can use code presented below to create a filter which ensures that a connection only accesses data at or below the chosen clearance level.

```
conn.setUserAttributes({'level': 'low'})
conn.setAttributeFilter(
    TripleAttribute.level <= UserAttribute.level)
conn.executeTupleQuery(
    'select ?class { ?s <ex://cls> ?class } order by ?class',
    output=True)
```

We can see that the output here contains only contacts with the access level of *low*. It omits the destroyer and Alpha submarine (these require *medium* level) as well as the top-secret Walruses.

```
------------------
| class          |
```

(continues on next page)

(continued from previous page)

```
==================
| ex://Ka-27     |
| ex://Kuznetsov |
| ex://Udaloy    |
------------------
```

The main advantage of the code presented above is that the filter can be set globally during the application setup and access control can then be achieved by varying user attributes on connection objects.

Let us now remove the attribute filter to prevent it from interfering with other examples. We will use the `clearAttributeFilter()` method.

```
conn.clearAttributeFilter()
```

It might be useful to change connection's attributes temporarily for the duration of a single code block and restore prior attributes after that. This can be achieved using the `temporaryUserAttributes()` method, which returns a context manager. The example below illustrates its use. It also shows how to use `getUserAttributes()` to inspect user attributes.

```
with conn.temporaryUserAttributes({'level': 'high'}):
    print('User attributes inside the block:')
    for k, v in conn.getUserAttributes().items():
        print('{0}: {1}'.format(k, v))
    print()
print('User attributes outside the block:')
for k, v in conn.getUserAttributes().items():
    print('{0}: {1}'.format(k, v))
```

```
User attributes inside the block:
level: high

User attributes outside the block:
level: low
```

2.22 Example 18: Pandas support

The SPARQL query language has somewhat limited capabilities when it comes to advanced numerical data analysis, data mining and other similar tasks. In these cases it is best to only use SPARQL to extract, filter and normalize data (perhaps coming from diverse sources - the ability to work with such data is one of the key advantages of the RDF data model) and rely on other tools to perform further analysis. One of the more popular tools that can be used in this context is the Pandas framework. The AllegroGraph Python client contains basic support for processing query results with this library. Let us see how this support can be leveraged in a simplified scenario.

As usual, we will start by opening a connection.

```
from franz.openrdf.connect import ag_connect

conn = ag_connect('python-tutorial', create=True, clear=True)
```

Now we will add some data. The first data set describes per capita cheese consumption in the United States in years 2000-2009 according to USDA. The values are expressed in pounds:

```
conn.addData('''
    prefix ex: <ex://>
    ex:c2000 ex:year 2000; ex:cheese 29.8 .
    ex:c2001 ex:year 2001; ex:cheese 30.1 .
    ex:c2002 ex:year 2002; ex:cheese 30.5 .
    ex:c2003 ex:year 2003; ex:cheese 30.6 .
    ex:c2004 ex:year 2004; ex:cheese 31.3 .
    ex:c2005 ex:year 2005; ex:cheese 31.7 .
    ex:c2006 ex:year 2006; ex:cheese 32.6 .
    ex:c2007 ex:year 2007; ex:cheese 33.1 .
    ex:c2008 ex:year 2008; ex:cheese 32.7 .
    ex:c2009 ex:year 2009; ex:cheese 32.8 .
''')
```

Our second set of samples is derived from NSF data and describes the number of civil engineering doctorates awarded each year.

```
conn.addData('''
    prefix ex: <ex://>
    ex:d2000 ex:year 2000; ex:doctorates 480 .
    ex:d2001 ex:year 2001; ex:doctorates 501 .
    ex:d2002 ex:year 2002; ex:doctorates 540 .
    ex:d2003 ex:year 2003; ex:doctorates 552 .
    ex:d2004 ex:year 2004; ex:doctorates 547 .
    ex:d2005 ex:year 2005; ex:doctorates 622 .
    ex:d2006 ex:year 2006; ex:doctorates 655 .
    ex:d2007 ex:year 2007; ex:doctorates 701 .
    ex:d2008 ex:year 2008; ex:doctorates 712 .
    ex:d2009 ex:year 2009; ex:doctorates 708 .
''')
```

We can use a SPARQL query to extract all this data and create a Pandas DataFrame from it:

```
query = '''
prefix ex: <ex://>
select ?year ?cheese ?doctorates {
    _:b1 ex:year ?year ; ex:cheese ?cheese .
    _:b2 ex:year ?year ; ex:doctorates ?doctorates .
}'''
with conn.executeTupleQuery(query) as result:
    df = result.toPandas()
print(df)
```

```
   year cheese  doctorates
0  2000   29.8         480
1  2001   30.1         501
2  2002   30.5         540
3  2003   30.6         552
4  2004   31.3         547
5  2005   31.7         622
6  2006   32.6         655
7  2007   33.1         701
8  2008   32.7         712
9  2009   32.8         708
```

Notice that the DataFrame can be used after the result has been discarded, since all required data has been copied.

At this point the `TupleQueryResult.toPandas()` method does not allow fine-grained control over types of the

returned columns. The `'cheese'` column contains decimal values, but floats would be more convenient for further computation. Thus we will modify the dataframe and convert the data:

```
df['cheese'] = df['cheese'].astype(float)
```

Now that we have the data in a form suitable for Pandas, we can perform some analysis. To keep this tutorial simple we will just measure the correlation between the number of civil engineering doctorates awarded and per capita cheese consumption:

```
correlation = df.corr()['cheese']['doctorates']
print("Correlation: %.5f" % correlation)
```

```
Correlation: 0.97433
```

The interpretation of this result is left as an exercise to the reader.

2.23 Running AG on AWS EC2

Federating multiple repositories located on the same server (as shown in *Example 16: Federated repositories*) can be useful for organizing data, but to truly explore the scalability potential of federated repositories we should look at a system consisting of multiple machines. This example will illustrate this by instantiating multiple AllegroGraph virtual machines on AWS, joining all servers in a single federated repository and issuing a simple query.

Warning: This example involves instantiating multiple AWS resources. These are *not* free and you *will* be charged by Amazon. While the cost should not exceed a few dollars, we are not responsible for any charges you might incur while running this tutorial.

To allow our script to interact with AWS we will need to install an additional dependency - boto3.

```
pip install boto3
```

We will also need to configure credentials to allow the script to access the Amazon account that it should use. This can be done in multiple ways. One possibility is to create a file named `~/.aws/credentials` with the following contents:

```
[default]
aws_access_key_id = YOUR_KEY
aws_secret_access_key = YOUR_SECRET
```

The key and key id can be obtained from the web interface of AWS.

We're now ready to begin writing the actual script. Let's start with some imports:

```
import boto3
import atexit
import time
```

We will need the `boto3` module to interact with AWS. We also want to make sure that any resources we have provisioned will be properly discarded when the scripts exits, even if it exits because of an error. The `atexit` module provides a convenient way of doing that.

Warning: While the mechanism of releasing resources used in this tutorial is reasonably robust, it might still fail in rare circumstances (e.g. if your computer loses power). It is important to manually check your AWS account for any stray resources after executing this tutorial to avoid unnecessary costs.

Now we will set a few configuration constants:

```
REGION = 'us-west-2'
AMI = 'ami-5616d82e'
TYPE = 't2.medium'
```

These describe the kind of servers that we will be provisioning, as well as the AWS region in which our resources will reside. *AMI* should be the image id of an AllegroGraph image appropriate for the chosen region. Image identifiers can be found here. The actual value shown above refers to the AllegroGraph 6.3.0 HVM image for the `us-west-2` region. `TYPE` is the 'instance type' and determines the amount of compute resources (RAM and CPU) that will be available for each of our servers. The list of instance types can be found here.

We will create a specified number of servers plus an additional one to manage the federation. The number below is the number of federated instances (i.e. it does *not* include the management instance).

```
INSTANCE_COUNT = 2
```

It is desirable to limit the connections to the system that we are creating so that only your machine will be able to access it. If you have a static IP address, fill it below. If not, leave the value as it is.

```
MY_IP = '0.0.0.0/0'
```

The next set of constants describe various aspect of the infrastructure that we are about to provision. There should be no need to adjust any values here.

```
VPC_CIDR = '10.0.0.0/16'
PUBLIC_CIDR = '10.0.0.0/24'
PRIVATE_CIDR = '10.0.1.0/24'
KEY_NAME = 'AG-TUTORIAL-KEYPAIR'
PROFILE_NAME = 'agprofile'
USER = 'ec2-user'
```

The first three values describe the structure of the virtual network that will contain our machines. The network will consist of two subnets - a public one which will contain the management instance and a private one which will contain all other servers. Only machines in the public subnet will be directly accessible over the Internet. `KEY_NAME` is a name that will be assigned to the keypair that we will generate and use to connect to our servers. `PROFILE_NAME` is the name for an instance profile that we will create for our AG servers. `USER` is the system user on the machines we want to connect to. Since AllegroGraph's images are based on Amazon Linux, that username must be set to `ec2_user`.

We will now create handles for Amazon services that we want to use.

```
ec2 = boto3.resource('ec2', region_name=REGION)
ec2_client = boto3.client('ec2', region_name=REGION)
iam = boto3.resource('iam', region_name=REGION)
ssm = boto3.client('ssm', region_name=REGION)
```

These come in two variants, called `resources` and `clients`. Clients provide low-level access to the Amazon API, while resources are a high-level abstraction built over clients. Using resources is slightly easier and thus preferable, but some operations can only be performed with clients. In our case `ec2_client` and `ssm` objects are clients for the respective AWS services, while other handles are resources.

The first thing that we will create is a security role that will be assigned to our machines. This is necessary to allow the use of SSM, which we will need to execute scripts.

```
ssm_role = iam.create_role(RoleName='ssm',
                           AssumeRolePolicyDocument="""{
    "Version":"2012-10-17",
    "Statement":[
       {
          "Effect":"Allow",
          "Principal":{
             "Service": ["ssm.amazonaws.com", "ec2.amazonaws.com"]
          },
          "Action":"sts:AssumeRole"
       }]}""")
atexit.register(ssm_role.delete)
```

The role created above can be assumed by EC2 instances and allows access to SSM. We have also installed an `atexit` handler to make sure that the role is deleted once we are done. We will do the same thing for every other AWS resource that we create.

To make all instances that have assumed the role defined above accessible to SSM we need to connect that role to the appropriate policy document.

```
ssm_arn = 'arn:aws:iam::aws:policy/service-role/AmazonEC2RoleforSSM'
ssm_role.attach_policy(PolicyArn=ssm_arn)
atexit.register(ssm_role.detach_policy, PolicyArn=ssm_arn)
```

Again, we have installed an atexit handler to undo the association. Without that we would not be able to remove the role itself.

Now we create an *instance profile* that we will use to launch instances using our new role.

```
instance_profile = iam.create_instance_profile(
  InstanceProfileName=PROFILE_NAME
)
atexit.register(instance_profile.delete)

instance_profile.add_role(RoleName=ssm_role.name)
atexit.register(instance_profile.remove_role,
                RoleName=ssm_role.name)
```

Now it is time to create the virtual network infrastructure for our system.

```
vpc = ec2.create_vpc(CidrBlock=VPC_CIDR)
atexit.register(vpc.delete)

public_subnet = vpc.create_subnet(CidrBlock=PUBLIC_CIDR)
atexit.register(public_subnet.delete)

private_subnet = vpc.create_subnet(CidrBlock=PRIVATE_CIDR)
atexit.register(private_subnet.delete)
```

We have two subnets - one for things that should be accessible from the Internet and one for all other machines. To make the public subnet work we need to add an Internet gateway to it.

```
internet_gateway = ec2.create_internet_gateway()
atexit.register(internet_gateway.delete)

```

(continues on next page)

(continued from previous page)

```
internet_gateway.attach_to_vpc(VpcId=vpc.vpc_id)
atexit.register(internet_gateway.detach_from_vpc,
                VpcId=vpc.vpc_id)
```

As usual, we have to install `atexit` handlers to undo all operations. This is important not only for operations that create resources, but also for those that add associations, since AWS will not allow us to remove a resource with existing associations.

Now we need to define a route table for the public subnet. The route table will basically tell our instances to use the Internet gateway that we have just created to communicate with the Internet.

```
public_route_table = vpc.create_route_table()
atexit.register(public_route_table.delete)

public_route_table.create_route(
    DestinationCidrBlock='0.0.0.0/0',
    GatewayId=internet_gateway.internet_gateway_id)

public_rt_assoc = public_route_table.associate_with_subnet(
    SubnetId=public_subnet.id)
atexit.register(public_rt_assoc.delete)
```

Machines in the private subnet should not be accessible from the Internet, but must still be able to access external resources. To facilitate that we will create a NAT gateway in the private subnet. A NAT gateway must have a public IP address, so will get one first.

```
nat_eip = ec2_client.allocate_address(Domain='vpc')
atexit.register(ec2_client.release_address,
                AllocationId=nat_eip['AllocationId'])
```

Before we actually create the gateway, we must ensure that we will be able to decommission it in a safe manner. The issue is that deleting a NAT gateway is not instantaneous and we cannot remove other resources (the VPC and subnet) until it is gone. So we will define a function that periodically checks the status of our gateway and blocks until it has been fully removed.

```
def wait_for_nat_gateway_termination(nat):
    for repeat in range(40):
        time.sleep(10)
        response = ec2_client.describe_nat_gateways(
            NatGatewayIds=[nat['NatGateway']['NatGatewayId']])
        if not response.get('NatGateways', False):
            return
        if response['NatGateways'][0]['State'] in ('deleted', 'failed'):
            return
    raise Exception('NAT gateway refuses to go away')
```

`boto3` uses so called `waiters` to automate the process of wating for a state change of an AWS resource. Unfortunately there is no `waiter` that checks for the termination of a NAT gateway, so we had to write our own. We will use `boto3` waiters in other parts of the code.

We are now ready to create the gateway. Notce that the gateway itself must be a part of the public subent, even though it is meant to be used by the private subnet.

```
nat = ec2_client.create_nat_gateway(
    AllocationId=nat_eip['AllocationId'],
    SubnetId=public_subnet.id
```

(continues on next page)

(continued from previous page)

```
)
atexit.register(wait_for_nat_gateway_termination, nat)
atexit.register(ec2_client.delete_nat_gateway,
                NatGatewayId=nat['NatGateway']['NatGatewayId'])
```

Notice two `atexit` handlers - one issues the delete command, the other one waits for its completion. The handlers are executed in reverse registration order, so the 'waiting' handler is registered first. We will see this pattern again in the future.

Now we will wait until the NAT gateway is functional This might take a while. Fortunately this time we can take advantage of a `boto3` waiter:

```
ec2_client.get_waiter('nat_gateway_available').wait(
    NatGatewayIds=[nat['NatGateway']['NatGatewayId']])
```

Now we need a route table for the private subnet

```
private_route_table = vpc.create_route_table()
atexit.register(private_route_table.delete)

private_route_table.create_route(
    DestinationCidrBlock='0.0.0.0/0',
    NatGatewayId=nat['NatGateway']['NatGatewayId'])

private_rt_assoc = private_route_table.associate_with_subnet(
    SubnetId=private_subnet.id)
atexit.register(private_rt_assoc.delete)
```

The next pair of resources to create will be the security groups. These determine what is allowed to connect to what over the network. We will want different rules for private and public subnets.

```
public_security_group = vpc.create_security_group(
    GroupName="ag-http-global",
    Description="Allow SSH + HTTP connections to AG")
atexit.register(public_security_group.delete)

public_ip_permissions = [{
    'IpProtocol': 'TCP',
    'FromPort': 10035,
    'ToPort': 10035,
    'IpRanges': [{'CidrIp': MY_IP}]
}, {
    'IpProtocol': 'TCP',
    'FromPort': 16000,
    'ToPort': 17000,
    'IpRanges': [{'CidrIp': MY_IP}]
}, {
    'IpProtocol': 'TCP',
    'FromPort': 22,
    'ToPort': 22,
    'IpRanges': [{'CidrIp': MY_IP}]
}]

public_security_group.authorize_ingress(
    IpPermissions=public_ip_permissions)
```

We allow SSH connections and HTTP connections to AG (both the frontend (`10035`) and the session ports

(16000-17000)) from the address we defined above. Note that if you have not changed the default value of MY_IP the the system will be accessible from anywhere, which is a security risk.

Private instances shall accept *all* connections, but only from machines within our virtual network.

```
private_security_group = vpc.create_security_group(
    GroupName="all-internal",
    Description="Allow all access from our network")
atexit.register(private_security_group.delete)

private_ip_permissions = [{
    'IpProtocol': '-1',
    'UserIdGroupPairs': [{'GroupId': private_security_group.id},
                         {'GroupId': public_security_group.id}]
}]

private_security_group.authorize_ingress(
    IpPermissions=private_ip_permissions)
```

Now it is time to generate a keypair. This is simply the SSH key that we will use to control access to our machines.

```
key_pair = ec2.create_key_pair(KeyName=KEY_NAME)
atexit.register(key_pair.delete)
```

As mentioned above, we will want to use SSM to execute scripts on our servers. To do that we have to ensure that an SSM agent is installed on each machine. We can do this with the following script.

```
user_data = """#!/bin/bash
    cd /tmp
    sudo yum install -y https://s3.amazonaws.com/ec2-downloads-windows/SSMAgent/
↪latest/linux_amd64/amazon-ssm-agent.rpm
    sudo start amazon-ssm-agent"""
```

The script itself will be executed with cloud-init.

Now it is finally time to create our machines. Let us start with the management node.

```
public_instance = ec2.create_instances(
    ImageId=AMI, InstanceType=TYPE,
    UserData=user_data,
    MinCount=1, MaxCount=1,
    KeyName=KEY_NAME,
    SecurityGroupIds=[
        public_security_group.id,
    ],
    IamInstanceProfile={
        "Arn": instance_profile.arn
    },
    SubnetId=public_subnet.id)[0]
atexit.register(
    ec2_client.get_waiter('instance_terminated').wait,
    InstanceIds=[public_instance.id])
atexit.register(public_instance.terminate)
```

We pass our SSM installation script in the UserData parameter. Notice how the atexit handlers wait for the instance to be fully deleted before proceeding to shutdown other systems.

Instances in the private subnet can be created in a similar way.

```
private_instances = ec2.create_instances(
    ImageId=AMI, InstanceType=TYPE,
    MinCount=INSTANCE_COUNT, MaxCount=INSTANCE_COUNT,
    UserData=user_data,
    KeyName=KEY_NAME,
    SecurityGroupIds=[
        private_security_group.id,
    ],
    IamInstanceProfile={
        "Arn": instance_profile.arn
    },
    SubnetId=private_subnet.id)
private_ids = [i.id for i in private_instances]
all_ids = [public_instance.id] + private_ids
atexit.register(
    ec2_client.get_waiter('instance_terminated').wait,
    InstanceIds=private_ids)
for instance in private_instances:
    atexit.register(instance.terminate)
```

We gather the ids of all our instances in two lists. Now let us wait until all our instances are operational.

```
ec2_client.get_waiter('instance_status_ok').wait(
    InstanceIds=all_ids)
```

We will need one more public IP address, to be used by the management instance that we need to connect to.

```
eip = ec2_client.allocate_address(Domain='vpc')
atexit.register(ec2_client.release_address,
                AllocationId=eip['AllocationId'])

eip_assoc = ec2_client.associate_address(
    AllocationId=eip['AllocationId'],
    InstanceId=public_instance.id)

atexit.register(ec2_client.disassociate_address,
                AssociationId=eip_assoc['AssociationId'])
```

We have installed the SSM agent on our machines, but it takes a moment for it to become operational. We will define yet another wait function to suspend the execution until then.

```
def wait_for_ssm_agents(instance_ids):
   for repeat in range(40):
       time.sleep(10)
       response = ssm.describe_instance_information(
           InstanceInformationFilterList=[{
               'key': 'InstanceIds',
               'valueSet': instance_ids
           }])
       num_responses = len(response.get(
           'InstanceInformationList', []))
       if num_responses != len(instance_ids):
           # It is normal for an instance to not show up
           # even if requested explicitly.
           continue
       for instance in response.get('InstanceInformationList'):
           if instance['PingStatus'] != 'Online':
```

(continues on next page)

(continued from previous page)

```
                break
        else:
            return
    raise Exception('Timed out waiting for SSM agents to activate.')

wait_for_ssm_agents(all_ids)
```

We will need one more function to wait for an SSM command to complete its execution (unfortunately there is no built-in waiter for this in `boto3`).

```
def wait_for_ssm_command(command):
    for repeat in range(40):
        time.sleep(10)
        response = ssm.list_commands(
            CommandId=command['Command']['CommandId'])
        if not response.get('Commands', False):
            return
        if response['Commands'][0]['Status'] in \
                ('Success', 'Cancelled', 'Failed', 'TimedOut'):
            return
    raise Exception(
        'Timed out waiting for an SSM command to finish.')
```

Before we proceed, we should wait until all AG servers have started (at this point we know that the machines are up, but it takes a moment for AG itself to start processing requests). In a production system we would probably achieve this by using a dedicated monitoring solution or some other advanced mechanism, but to keep this tutorial simple we will just use SSM commands to poll the AG port until it starts responding.

```
wait_for_ag = [
  'until curl -f http://127.0.0.1:10035/version; do sleep 2; done'
]
cmd = ssm.send_command(
    InstanceIds=all_ids,
    DocumentName='AWS-RunShellScript',
    Parameters={'commands': wait_for_ag},
    TimeoutSeconds=120)
wait_for_ssm_command(cmd)
```

We will now use SSM to send a script that will load sample data into our instances. To keep things simple we will add just a single triple per instance. That triple will include the index of the instance (a number between `0` and `INSTANCE_COUNT - 1`).

```
script = [
    'ID=$(curl http://169.254.169.254/latest/meta-data/ami-launch-index)',
    'curl -X PUT -u test:xyzzy http://127.0.0.1:10035/repositories/r',
    ' '.join(['curl -X PUT -u test:xyzzy http://127.0.0.1:10035/repositories/r/
↪statement',
              '--data-urlencode "subj=<ex://instance${ID}>"',
              '--data-urlencode "pred=<ex://id>"',
              '--data-urlencode "obj=\\"${ID}\\"^^<http://www.w3.org/2001/XMLSchema
↪#integer>"'])
]

cmd = ssm.send_command(InstanceIds=private_ids,
                       DocumentName='AWS-RunShellScript',
                       Parameters={'commands': script})
```

(continues on next page)

(continued from previous page)

```
wait_for_ssm_command(cmd)
```

Note that the waiter functions defined above are not particularly robust in their error handling. In a production system consisting of a larger number of instances a more elaborate mechanism for error detection and handling (retries) would have to be implemented.

Now we are ready to create the federation by connecting to the management machine. Notice that we use the default credentials for the AllegroGraph AMI.

```
from franz.openrdf.sail.allegrographserver import AllegroGraphServer

server = AllegroGraphServer(host=eip['PublicIp'],
                            user='test', password='xyzzy')
conn = server.openSession(
    '+'.join('<http://test:xyzzy@%s/repositories/r>'
                % i.private_ip_address
             for i in private_instances))
```

And now we can issue a query:

```
from franz.openrdf.query.query import QueryLanguage

query = conn.prepareTupleQuery(QueryLanguage.SPARQL, """
    SELECT (AVG(?o) as ?avg) { ?s ?p ?o }""")
query.evaluate(output=True)
```

This should print the average instance id, which should equal `(INSTANCE_COUNT - 1) / 2`.

```
-------
| avg |
=======
| 0.5 |
-------
```

That is all - the script is now done and will start tearing down the whole infrastructure.

Warning: Remember to check your AWS account for any leftover resources after running the tutorial to avoid unnecessary costs.

CHAPTER 3

AllegroGraph Python API Reference

This is a description of the Python Application Programmer's Interface (API) to Franz Inc.'s AllegroGraph.

The Python API offers convenient and efficient access to an AllegroGraph server from a Python-based application. This API provides methods for creating, querying and maintaining RDF data, and for managing the stored triples.

Note: The Python API deliberately emulates the Eclipse RDF4J API (formerly Aduna Sesame) to make it easier to migrate from RDF4J to AllegroGraph. The Python API has also been extended in ways that make it easier and more intuitive than the RDF4J API.

3.1 AllegroGraphServer class

class franz.openrdf.sail.allegrographserver.**AllegroGraphServer**(*host=None*, *port=None*, *user=None*, *password=None*, *cainfo=None*, *sslcert=None*, *verifyhost=None*, *verifypeer=None*, *protocol=None*, *proxy=None*, ***options*)

The AllegroGraphServer object represents a remote AllegroGraph server on the network. It is used to inventory and access the catalogs of that server.

Example:

```
server = AllegroGraphServer(host="localhost", port=8080,
                            user="test", password="pw")
```

__init__	Define the connection to the AllegroGraph HTTP server.
addRole	Creates a new role.
addRoleAccess	Grant read/write access to a role.
addRolePermission	Grants a role a certain permission.
addRoleSecurityFilter	Add a security filter for the user.
addScript	Create or replace a sitescript.
addUser	Create a new user.
addUserAccess	Grant read/write access to a user.
addUserPermission	Assigns the given permission to this user.
addUserRole	Add a role to a user.
addUserSecurityFilter	Add a security filter for the user.
changeUserPassword	Change the password for the given user.
deleteRole	Deletes a role.
deleteRoleAccess	Revoke read/write access for a role.
deleteRolePermission	Revokes a permission for a role.
deleteRoleSecurityFilter	Add a security filter for the user.
deleteScript	Delete a sitescript.
deleteUser	Delete a user.
deleteUserAccess	Takes the same parameters as PUT on this URL, but revokes the access instead of granting it.
deleteUserData	Remove user data from the server.
deleteUserPermission	Revokes the given permission for this user.
deleteUserRole	Remove a role from a user.
deleteUserSecurityFilter	Add a security filter for the user.
getInitfile	Retrieve the contents of the server initialization file.
getScript	Get the body of a sitescript.
getUserData	Retrieve user data value with given key.
listCatalogs	Get the list of catalogs on this server.
listRoleAccess	Query the access granted to a role.
listRolePermissions	Lists the permission flags granted to a role.
listRoleSecurityFilters	List security filters for user.
listRoles	Returns the names of all defined roles.
listScripts	Return the list of Sitescripts currently on the server.
listUserAccess	Retrieve the read/write access for a user.
listUserData	Get all used keys from the user data store on the server.
listUserEffectiveAccess	Like listUserAccess, but also includes the access granted to roles that this user has.
listUserEffectivePermissions	Retrieve the permission flags assigned to the user, or any of its roles.
listUserPermissions	List the permission flags that have been assigned to a user (any of super, eval, session, replication).
listUserRoles	Retrieves a list of role names for this user name.
listUserSecurityFilters	List security filters for user.
listUsers	Returns a list of names of all the users that have been defined.

Continued on next page

Table 1 – continued from previous page

openCatalog	Open a catalog by name.
openFederated	Open a session that federates several repositories.
openSession	Open a session on a federated, reasoning, or filtered store.
setInitfile	Replace the current initialization file contents with the content string or remove if None.
setUserData	Set user data with given key.

3.2 Catalog class

class franz.openrdf.sail.allegrographserver.**Catalog**(*name*, *server*, *client*)

Container of multiple repositories (triple stores).

Construct catalogs using the server object:

```
catalog = server.openCatalog('scratch')
```

createRepository	Creates a new repository with the given name.
deleteRepository	Delete a repository.
getName	Return the catalog name.
getRepository	Creates or opens a repository.
listRepositories	Return a list of names of repositories (triple stores) managed by this catalog.

3.3 Spec module

Helper functions for creating session specification strings. See openSession()

federate	Create a session spec for connecting to a federated store.
graphFilter	Create a graph-filtered session spec.
local	Create a session spec for connecting to a store on the local server.
map	Return a list of the results of applying the function to the items of the argument sequence(s).
reason	Create a session spec that adds reasoning support to another session.
remote	Create a session spec for connecting to a store on another server.
url	Create a session spec for connecting to a remote store with known URL.

3.4 Repository class

class franz.openrdf.repository.repository.**Repository**(*catalog*, *database_name*, *repository*)

A repository contains RDF data that can be queried and updated. Access to the repository can be acquired

by opening a connection to it. This connection can then be used to query and/or update the contents of the repository.

Please note that a repository needs to be initialized before it can be used and that it should be shut down before it is discarded/garbage collected. Forgetting the latter can result in loss of data (depending on the Repository implementation)!

Construct instances using `getRepository()`.

```
with catalog.getRepository("agraph_test", Repository.ACCESS) as repo:
    ...
```

`__init__`	Invoke through `getRepository()`.
`getConnection`	Opens a connection to this store that can be used for querying and updating the contents of the store.
`getDatabaseName`	Return the name of the database (remote triple store) that this repository is interfacing with.
`getSpec`	Return a session spec string for this repository.
`getValueFactory`	Return a ValueFactory for this store.
`initialize`	Initializes this repository.
`isWritable`	Checks whether this store is writable, i.e.
`registerDatatypeMapping`	Register an inlined datatype.
`shutDown`	Shuts the store down, releasing any resources that it keeps hold of.

3.5 Utility connection functions

Manually constructing server, catalog and repository objects is often tedious when only a simple connection to a single repository is needed. In that case the connection may be created directly using `ag_connect()`.

`ag_connect`	Create a connection to an AllegroGraph repository.

3.6 RepositoryConnection class

class franz.openrdf.repository.repositoryconnection.**RepositoryConnection**(*repository*, *close_repo=False*)

The RepositoryConnection class is the main interface for updating data in and performing queries on a `Repository`. By default, a RespositoryConnection is in autoCommit mode, meaning that each operation corresponds to a single transaction on the underlying triple store. autoCommit can be switched off, in which case it is up to the user to handle transaction commit/rollback. Note that care should be taken to always properly close a RepositoryConnection after one is finished with it, to free up resources and avoid unnecessary locks.

Note that concurrent access to the same connection object is explicitly forbidden. The client must perform its own synchronization to ensure non-concurrent access.

Several methods take a *vararg* argument that optionally specifies a set of contexts on which the method should operate. (A context is the URI of a subgraph.) Note that a *vararg* parameter is optional, it can be completely left out of the method call, in which case a method either operates on a provided statement's context (if one of the method parameters is a statement or collection of statements), or operates on the repository as a whole, completely ignoring context. A *vararg* argument may also be `None`, meaning that the method operates on those statements which have no associated context only.

`RepositoryConnection` objects should be constructed using `getConnection()`. To ensure that repository connections are closed, the best practice is to use a `with` statement:

```
with repository.getConnection() as conn:
    ...
```

`__init__`	Call through `getConnection()`.

3.6.1 General connection methods

This section contains the `RepositoryConnection` methods that create, maintain, search, and delete triple stores.

`add`	Calls `addTriple()`, `addStatement()`, or `addFile()`.
`addData`	Adds data from a string to the repository.
`addFile`	Loads a file into the triple store.
`addStatement`	Add the supplied statement to the specified contexts in the repository.
`addTriple`	Add a single triple to the repository.
`addTriples`	Add the supplied triples or quads to this repository.
`clear`	Removes all statements from designated contexts in the repository.
`clearNamespaces`	Delete all namespaces in this repository for the current user.
`close`	Close the connection.
`createBNode`	Create a new blank node.
`createLiteral`	Create a new literal with value `value`.
`createRange`	Create a compound literal representing a range from lowerBound to upperBound.
`createStatement`	Create a new Statement object.
`createURI`	Creates a new URI from the supplied string-representation(s).
`deleteDuplicates`	Delete duplicate triples from the store.
`executeBooleanQuery`	Prepare and immediately evaluate a query that returns a boolean.
`executeGraphQuery`	Prepare and immediately evaluate a query that returns RDF.
`executeTupleQuery`	Prepare and immediately evaluate a query that returns tuples.
`executeUpdate`	Prepare and immediately evaluate a SPARQL update query.
`export`	Export all explicit statements in the specified contexts to a file.
`exportStatements`	Export statements to a file.
`getAddCommitSize`	Get the current value of `add_commit_size`.
`getContextIDs`	Return a list of context resources, one for each context referenced by a quad in the triple store.
`getDuplicateStatements`	Return all duplicates in the store.
`getNamespace`	Get the namespace that is associated with the specified prefix, if any.

Continued on next page

Table 7 – continued from previous page

`getNamespaces`	Get all declared prefix/namespace pairs.
`getSpec`	Get the session specification string for this repository.
`getStatements`	Get all statements with a specific subject, predicate and/or object from the repository.
`getStatementsById`	Return all statements whose triple ID matches an ID in the list 'ids'.
`getValueFactory`	Get the `ValueFactory` associated with this repository.
`isEmpty`	Return `True` if this repository does not contain any (explicit) statements.
`namespace`	Creates an object that allows for simple creation of URIs in given namespace.
`prepareBooleanQuery`	Parse `query` into a boolean query object which can be executed against the triple store.
`prepareGraphQuery`	Parse `query` into a graph query object which can be executed against the triple store.
`prepareTupleQuery`	Parse `query` into a tuple query object which can be executed against the triple stroe.
`prepareUpdate`	Parse `query` into an update query object which can be executed against the triple store.
`registerDatatypeMapping`	Register an inlined datatype.
`remove`	Call `removeTriples()` or `removeStatement()`.
`removeNamespace`	Remove a namespace declaration by removing the association between a prefix and a namespace name.
`removeQuads`	Remove enumerated quads from this repository.
`removeQuadsByID`	Remove all quads with matching IDs.
`removeStatement`	Remove the supplied statement from the specified contexts in the repository.
`removeTriples`	Remove the statement(s) with the specified subject, predicate and object from the repository, optionally restricted to the specified contexts.
`setAddCommitSize`	Set the value of `add_commit_size`.
`setNamespace`	Define or redefine a namespace mapping in the repository.
`size`	Returns the number of (explicit) statements that are in the specified contexts in this repository.
`add_commit_size`	The threshold for commit size during triple add operations.

3.6.2 Triple Index Methods

These `RepositoryConnection` methods support user-defined triple indices. See AllegroGraph Triple Indices for more information on this topic.

`listIndices`	Return the list of the current set of triple indices.
`listValidIndices`	Return the list of valid index names.
`addIndex`	Add a specific type of index to the current set of triple indices.

Continued on next page

Table 8 – continued from previous page

`dropIndex`	Removes a specific type of index to the current set of triple indices.
`optimizeIndices`	Optimize indices.

3.6.3 Free Text Indexing Methods

The following `RepositoryConnection` methods support free-text indexing in AllegroGraph.

`createFreeTextIndex`	Create a free-text index with the given parameters.
`deleteFreeTextIndex`	Delete a free-text index from the server.
`evalFreeTextSearch`	Return a list of statements for the given free-text pattern search.
`getFreeTextIndexConfiguration`	Get the current settings of a free-text index.
`listFreeTextIndices`	Get the names of currently defined free-text indices.
`modifyFreeTextIndex`	Modify parameters of a free-text index.

Note that text search is implemented through a SPARQL query using a "magic" predicate called `fti:search`. See the *AllegroGraph Python API Tutorial* for an example of how to set up this search.

3.6.4 Prolog Rule Inference Methods

These `RepositoryConnection` methods support the use of Prolog rules in AllegroGraph. Any use of Prolog rules requires that you create a *transaction* to run them in.

`addRules`	Add Prolog functors to the current session.
`loadRules`	Add Prolog rules from file to the current session.

3.6.5 Geospatial Reasoning Methods

These `RepositoryConnection` methods support geospatial reasoning.

`createBox`	Create a rectangular search region (a box) for geospatial search.
`createCircle`	Create a circular search region for geospatial search.
`createCoordinate`	Create an x, y or latitude, longitude coordinate in the current coordinate system.
`createLatLongSystem`	Create a spherical coordinate system and use it as the current coordinate system.
`createPolygon`	Define a polygonal region with the specified vertices.
`createRectangularSystem`	Create a Cartesian coordinate system and use it as the current coordinate system.
`getGeoType`	Get the current geospatial coordinate system.
`setGeoType`	Set the current geospatial coordinate system.

3.6.6 Social Network Analysis Methods

The following `RepositoryConnection` methods support Social Network Analysis in AllegroGraph. The Python API to the Social Network Analysis methods of AllegroGraph requires Prolog queries, and therefore must be run in a *dedicated session*.

`registerNeighborMatrix`	Construct a neighbor matrix named `name`.
`registerSNAGenerator`	Create a new SNA generator named `name`.

3.6.7 Transactions

AllegroGraph lets you set up a special `RepositoryConnection` (a "session") that supports transaction semantics. You can add statements to this session until you accumulate all the triples you need for a specific transaction. Then you can commit the triples in a single act. Up to that moment the triples will not be visible to other users of the repository.

If anything interrupts the accumulation of triples building to the transaction, you can roll back the session. This discards all of the uncommitted triples and resynchronizes the session with the repository as a whole.

Closing the session discards all uncommitted triples and all rules, generators, and matrices that were created in the session. Rules, generators, and matrices cannot be committed. They persist as long as the session persists.

`openSession`	Open a session.
`closeSession`	Close a session.
`session`	A session context manager for use with the `with` statement:
`commit`	Commit changes on an open session.
`rollback`	Roll back changes on open session.

3.6.8 Subject Triples Caching

You can enable subject triple caching to speed up queries where the same subject URI appears in multiple patterns. The first time AllegroGraph retrieves triples for a specific resource, it caches the triples in memory. Subsequent query patterns that ask for the same subject URI can retrieve the matching triples very quickly from the cache. The cache has a size limit and automatically discards old entries as that limit is exceeded.

`enableSubjectTriplesCache`	Maintain a cache of size `size` that caches, for each accessed resource, quads where the resource appears in subject position.
`disableSubjectTriplesCache`	Disable the subject triples cache (see `enableSubjectTriplesCache()`).
`getSubjectTriplesCacheSize`	Return the current size of the subject triples cache.

3.7 Query Class (and Subclasses)

Note: The Query class is non-instantiable. It is an abstract class from which the three query subclasses are derived. It is included here because of its methods, which are inherited by the subclasses.

A query on a `Repository` that can be formulated in one of the supported query languages (for example SPARQL).

The query can be parameterized, to allow one to reuse the same query with different parameter bindings.

The best practice is to allow the `RepositoryConnection` object to create an instance of one of the Query subclasses (`TupleQuery`, `GraphQuery`, `BooleanQuery`, `UpdateQuery`). There is no reason for the Python application programmer to create such objects directly.

```
tupleQuery = conn.prepareTupleQuery(QueryLanguage.SPARQL, queryString)
result = tupleQuery.evaluate()
```

`evaluate_generic_query`	Evaluate a SPARQL or PROLOG query, which may be a 'select', 'construct', 'describe' or 'ask' query (in the SPARQL case).
`getBindings`	Retrieve the bindings that have been set on this query.
`getDataset`	Get the dataset against which this query will operate.
`getIncludeInferred`	Check whether this query will return inferred statements.
`removeBinding`	Removes a previously set binding on the supplied variable.
`setBinding`	Binds the specified variable to the supplied value.
`setBindings`	Set multiple variable bindings.
`setCheckVariables`	Determine whether the presence of variables in the select clause not referenced in a triple is flagged.
`setConnection`	Internal call to embed the connection into the query.
`setContexts`	Assert a set of contexts (named graphs) that filter all triples.
`setDataset`	Select the dataset against which to evaluate this query, overriding any dataset that is specified in the query itself.
`setIncludeInferred`	Determine whether evaluation results of this query should include inferred statements.

3.7.1 Subclass TupleQuery

This subclass is used with SELECT queries. Use the `prepareTupleQuery()` method to create a TupleQuery object. The results of the query are returned in a `QueryResult` iterator that yields a sequence of `binding sets`.

TupleQuery uses all the methods of the `Query` class, plus two more:

`evaluate`	Execute the embedded query against the RDF store.
`analyze`	Analyze the query.

3.7.2 Subclass GraphQuery

This subclass is used with CONSTRUCT and DESCRIBE queries. Use the `prepareGraphQuery()` method to create a GraphQuery object. The results of the query are returned in a `GraphQueryResult` iterator that yields a sequence of `statements`.

`GraphQuery` implements all the methods of the `Query` class, plus one more:

`evaluate`	Execute the embedded query against the RDF store.

3.7.3 Subclass BooleanQuery

This subclass is used with ASK queries. Use the `prepareBooleanQuery()` method to create a `BooleanQuery` object. The results of the query are `True` or `False`.

BooleanQuery implements all the methods of the `Query` class, plus one more:

`evaluate`	Execute the embedded query against the RDF store.

3.7.4 Subclass UpdateQuery

This subclass is used for `DELETE` and `INSERT` queries. The result returned when the query is evaluated is a boolean that can be used to tell if the store has been modified by the operation. Use the `prepareUpdate()` method to create an `UpdateQuery` object.

`UpdateQuery` implements all the methods of the `Query` class, plus one more:

`evaluate`	Execute the embedded update against the RDF store.

3.8 QueryResult Class

A QueryResult object is simply an iterator that also has a `close()` method that must be called to free resources. Such objects are returned as a result of SPARQL and PROLOG query evaluation and should not be constructed directly. Result objects are context managers and can be used in the `with` statement. The recommended usage looks like this:

```
tupleQuery = conn.prepareTupleQuery(QueryLanguage.SPARQL, queryString)
with tupleQuery.evaluate() as results:
   for result in results:
       print result
```

It is also possible to use one of the `execute*Query` methods (e.g. `executeTupleQuery()`) to prepare and execute a query in a single call:

```
with conn.executeTupleQuery(queryString) as results:
   for result in results:
       print result
```

`close`	Release resources used by this query result.
`__next__`	Return the next result item if there is one.

3.8.1 Subclass TupleQueryResult

A QueryResult subclass used for queries that return tuples.

`getBindingNames`	Get the names of the bindings, in order of projection.

3.8.2 Subclass GraphQueryResult

A QueryResult subclass used for queries that return statements. Objects of this class are also `RepositoryResult` instances.

3.9 RepositoryResult class

A `RepositoryResult` object is a result collection of `statements` that can be iterated over. It keeps an open connection to the backend for lazy retrieval of individual results. Additionally it has some utility methods to fetch all results and add them to a collection.

By default, a `RepositoryResult` is not necessarily a (mathematical) set: it may contain duplicate objects. Duplicate filtering can be switched on, but this should not be used lightly as the filtering mechanism is potentially memory-intensive.

A `RepositoryResult` must be closed after use to free up any resources (open connections, read locks, etc.) it has on the underlying repository. To make this easier it is a context manager and can be used in the `with` statement.

```
graphQuery = conn.prepareGraphQuery(QueryLanguage.SPARQL, queryString)
with graphQuery.evaluate() as results:
    for result in results:
        print result
```

`close`	Shut down the iterator to be sure the resources are freed up.
`__next__`	Return the next Statement in the answer, if there is one.
`enableDuplicateFilter`	Switch on duplicate filtering while iterating over objects.
`asList`	Returns a list containing all objects of this RepositoryResult in order of iteration.
`addTo`	Add all objects of this RepositoryResult to the supplied collection.
`rowCount`	Get the number of statements in this result object.

3.10 Statement Class

A `Statement` is a client-side triple. It encapsulates the subject, predicate, object and context (subgraph) values of a single triple and makes them available.

Best practice is to allow the `RepositoryConnection.createStatement()` method to create and return the Statement object. There is no reason for the Python application programmer to create a Statement object directly.

```
stmt1 = conn.createStatement(alice, age, fortyTwo)
```

`getContext`	Get the graph (the fourth, optional element of the statement).
`getObject`	Get the object (the third element of the statement).
`getPredicate`	Get the predicate (the second element of the statement).
`getSubject`	Get the subject (the first element of the statement).

3.11 ValueFactory Class

A ValueFactory is a factory for creating URIs, blank nodes, literals and `Statement` objects. In the AllegroGraph Python interface, the `ValueFactory` class is regarded as obsolete. Its functions have been subsumed by the expanded capability of the `RepositoryConnection` class. It is documented here for the convenience of users porting an application from Eclipse RDF4J.

`createBNode`	See `RepositoryConnection.createBNode()`.
`createLiteral`	See `RepositoryConnection.createLiteral()`.
`createStatement`	See `RepositoryConnection.createStatement()`.
`createURI`	See `RepositoryConnection.createURI()`.

CHAPTER 4

AllegroGraph Python client release history

4.1 Release 100.3.0

4.1.1 User data access

AllegroGraph allows each user to store arbitrary key-value data on the server. This storage can now be accessed from Python by using new ‘‘AllegroGraphServer‘ methods:

- `listUserData()`
- `getUserData()`
- `setUserData()`
- `deleteUserData()`

4.2 Release 100.2.0

4.2.1 Pandas support

It is now possible to turn a query result into a Pandas DataFrame by calling the `toPandas()` method of the result object. Note that Pandas must be installed separately for this to work.

4.3 Release 100.1.2

4.3.1 bug25281: Proxy settings are ignored

Proxy settings used to be ignored when the requests backend was used. This has been corrected.

Thanks to Iván Darío Ramos Vacca for reporting the bug and providing a fix.

4.4 Release 100.1.1

A bugfix release that adds some missing dependencies that are needed when using Python < 3.5.

4.5 Release 100.1.0

4.5.1 Triple attributes

Added support for triple attributes (requires AG >= 6.1). Specifically it is now possible to:

- Set and retrieve the static attribute filter using `conn.setAttributeFilter()` and `conn.getAttributeFilter()`
- Set and retrieve user attributes (that will be sent with each request) using `conn.setUserAttributes()` and `conn.getUserAttributes()`.
- Manage attribute definitions using various methods in the connection class..
- Add triples with attributes - a new keyword parameter named 'attributes' has been added to methods that add triples, such as `addData()`. It is also possible to pass five-element tuples to `addTriples()`, where the fifth element is a dictionary of attribute values.

4.5.2 Distributed transaction settings

It is now possible to configure distributed transaction parameters in multiple ways:

- By passing arguments to the `commit()` method
- By calling `setTransactionSettings()` on the connection object.
- By using a context manager returned by the `temporaryTransactionSettings()` method.

In all cases the settings can be passed either in a single `TransactionSettings` object or as individual keyword arguments.

4.5.3 Enhanced namespace objects

Namespace objects can now create URIs when indexed or called like a function. This makes it easier to create URIs where the local name is not a valid attribute name:

```
>>> from franz.openrdf.connect import ag_connect
>>> conn = ag_connect('repo')
>>> ex = conn.namespace('http://franz.com/example/')
>>> ex('is')
<http://franz.com/example/is>
>>> ex['def']
<http://franz.com/example/def>
```

4.6 Release 100.0.4

4.6.1 Jupyter-friendly stdout

The `output_to` context manager (used internally when writing output to stdout) has been modified to work better in environments that hijack the `sys.stdout` value, such as Jupyter notebooks or IDLE.

4.7 Release 100.0.3

4.7.1 Resolved issues with running unit tests from a wheel

Some unit tests used to fail when the module was installed from a binary wheel. This has been corrected.

4.7.2 bug25081: The 'context' argument to addTriples() is broken

Using the `addTriples()` method with the `context` parameter set to a non-default value used to produce errors:

```
>>> conn.addTriples([(s, p, o)], context=g)
400 MALFORMED DATA: Invalid graph name: (<ex://g>)
```

This has been corrected. Context can now be set to a single URI or a list of URIs. Both URI objects and strings are supported.

4.7.3 bug25079: Statement objects not created from strings are broken

Statement objects that were created in user code were not fully functional. In particular attempts to convert such statements to strings or to pass them to addTriples() would fail.

This has been corrected.

4.7.4 Namespace objects

Namespace objects can be used to create URIs, as in the following example:

```
>>> from franz.openrdf.connect import ag_connect
>>> conn = ag_connect('repo')
>>> ex = conn.namespace('http://franz.com/example/')
>>> ex.foo
<http://franz.com/example/foo>
```

4.8 Release 100.0.2

4.8.1 New query methods

Four new methods have been added to the RepositoryConnection class:

- executeTupleQuery()

- executeGraphQuery()
- executeBooleanQuery()
- executeUpdate()

These can be used to prepare and evaluate a SPARQL query in a single call.

4.8.2 New tutorial

The tutorial has been updated and restyled using Sphinx.

4.8.3 Finalizers for query results

All result objects are now closed automatically when garbage collected. This makes it possible to write simple loops like the one below:

```
for stmt in conn.executeTupleQuery('...'):
    ...
```

without having to use the `with` statement, since reference counting will ensure that the query result is closed at the right time. Note that this should not be relied upon in more complex scenarios, where circular references might occur and prevent the result object from being closed.

4.8.4 Connection parameters can now be passed in environment variables

The following environment variables are now used when connecting to the server:

- `AGRAPH_HOST` - server address, the default is '127.0.0.1'
- **`AGRAPH_PORT` - port number (default: 10035 for HTTP connections,** 10036 for HTTPS).
- `AGRAPH_USER` - Username, no default.
- `AGRAPH_PASSWORD` - Password, no default.

Note that parameters passed to `ag_connect()` or `AllegroGraphServer()` will override these variables.

4.8.5 Various fixes related to data export

Specifically the following adjustments have been done:

- Changed the default RDF export format to N-Quads.
- Fixed a bug where errors returned during export caused an encoding error.
- Provided a default format (CSV) for tuple queries.
- Value of the output parameter can now be True (stdout) or a file descriptor.

4.9 Release 100.0.1

4.9.1 bug24892: Time parsing fixes

The Python client used to fail when trying to retrieve a datetimeValue() of a literal that contained time zone information. This has been corrected.

All datetime objects created by the Python API are now timezone-aware.

4.9.2 rfe15005: duplicate suppression control API

It is now possible to set and query the duplicate suppression policy of a repository from Python, using three new methods of the connection object:

- getDuplicateSuppressionPolicy()
- setDuplicateSuppressionPolicy()
- disableDuplicateSuppression()

4.9.3 New export methods

A new mechanism for exporting data has been added. It utilizes a new `output` parameter that has been added to the following methods:

- RepositoryConnection.getStatements()
- RepositoryConnection.getStatementsById()
- TupleQuery.evaluate()
- GraphQuery.evaluate()

Setting the new parameter to a file name or a file-like object will cause the data that would normally be returned by the call to be saved to the specified file instead. Serialization format can be controlled by setting another new parameter, `output_format`.

4.10 Release 100.0.0

4.10.1 New versioning scheme

Client versions no longer match the server version. Major version number has been bumped to 100 to avoid confusion.

4.10.2 bug24819: Circular import

Importing com.franz.openrdf.query.query failed due to a circular import. Thanks to Maximilien de Bayser for reporting this.

4.10.3 bug24826: removeStatement uses context instead of object

The removeStatement method of RepositoryConnection was broken. Patch by Maximilien de Bayser.

4.11 Release 6.2.2.0.4

4.11.1 bug24728: Incorrect conversion between boolean literals and Python values

The booleanValue() method of the Literal class used to work incorrectly. It would return True for any literal that is not empty, including the "false"^^xsd:boolean literal. This has been corrected - the function will now return expected values for literals of type xsd:boolean. Result for other types remains undefined.

4.12 Release 6.2.2.0.1

4.12.1 bug24680: to_native_string is broken on Python 2

The Python client sometimes failed while processing values with non-ascii characters, showing the following error message:

UnicodeEncodeError: 'ascii' codec can't encode characters in position ??: ordinal not in range(128)

This has been corrected.

4.13 Release 6.2.2.0.0

Released with AllegroGraph 6.2.2. Change log for this and all previous Python client releases can be found in AllegroGraph release notes: https://franz.com/agraph/support/documentation/current/release-notes.html

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

franz.openrdf.sail.spec, 83

Index

F

franz.openrdf.sail.spec (module), 83