rdfox_runner

Release 0.1.0

Rick Lupton

Aug 02, 2023

CONTENTS:

CHAPTER

ONE

INSTALLING RDFOX_RUNNER

Install rdfox_runner using pip:

```
pip install rdfox_runner
```

Of course, you will also need a copy of RDFox.

CHAPTER

TWO

RUNNING BASIC RDFOX SCRIPTS

The simplest way to use rdfox_runner goes like this:

- Set up a temporary directory with the required input files, scripts, rules etc.
- Run RDFox sandbox in that directory
- RDFox produces some output files
- The contents of the output files is captured and returned

For example, if we have some RDF triples in *facts.ttl*, and a query to answer in *query.rq*, we can get the answer to the query like this:

```
input_files {
    "facts.ttl": "path/to/facts.ttl",
    "query.rq": "path/to/query.rq",
}
script = [
    'dstore create default type par-complex-nn',
    'import facts.ttl',
    'set query.answer-format "text/csv"',
    'set output "output.csv"',
    'answer query.rq',
]
with RDFoxRunner(input_files, script) as rdfox:
    result = rdfox.files("output.csv").read_text()
```

Alternatively, you can start RDFox running and then interact with its REST API; see *Running RDFox and interacting with the endpoint*.

CHAPTER

THREE

RUNNING RDFOX AND INTERACTING WITH THE ENDPOINT

If you want to run multiple queries or interact with RDFox while it is running, you can start the RDFox REST endpoint.

For example, if we have some RDF triples in *facts.ttl*, we can answer queries like this:

```
input_files {
    "facts.ttl": "path/to/facts.ttl",
}
script = [
    'dstore create default type par-complex-nn',
    'import facts.ttl',
    'endpoint start',
]
with RDFoxRunner(input_files, script) as rdfox:
    result = rdfox.query(sparql_query)
```

See the *`rdfox_runner.RDFoxEndpoint`* API documentation for details of the query methods.

API REFERENCE

4.1 RDFox endpoint

The *rdfox_runner.RDFoxEndpoint* class helps to interface with a running RDFox endpoint.

class rdfox_runner.**RDFoxEndpoint**(*namespaces: Optional[Mapping] = None*)
Interface to interact with a running RDFox endpoint.

Parameters **namespaces** – dict of RDFlib namespaces to bind

add_triples(*triples*)
Add triples to the RDF data store.

In principle this should work via the rdflib SPARQLUpdateStore, but RDFox does not accept data in that format.

Note: compatible with RDFox version 5.0 and later.

connect(*url: str*)
Connect to RDFox at given base URL.

The SPARQL endpoint is at *{url}/datastores/default/sparql*.

facts(*format='text/turtle'*) → str
Fetch all facts from the server.

Parameters **format** – format for results send in Accept header.

query(*query_object*, **args*, ***kwargs*)
Query the SPARQL endpoint.

This method is a simple wrapper about `rdflib.Graph.query()` which shows more useful error output when there is a problem with the query.

Raises ParsingError

query_dataframe(*query_object*, *n3=True*, **args*, ***kwargs*)
Query the SPARQL endpoint, returning a pandas DataFrame.

Because this is often useful for human-readable output, the default is to serialise results in N3 notation, using defined prefixes.

See *query()*.

Parameters **n3** – whether to return results in N3 notation, defaults to True.

query_one_record(*query_object*, **args*, ***kwargs*) → Dict[str, Any]
Query the SPARQL endpoint, and check that only one result is returned (as a dict).

See *query()*.

query_raw(*query*, *answer_format=None*)
Query the RDFox SPARQL endpoint directly.

Unlike *query*, the result is the raw response from RDFox, not an *rdflib* Result object.

Raises ParsingError

query_records(*query_object*, *n3=False*, **args*, ***kwargs*) → List[Dict[str, Any]]
Query the SPARQL endpoint, returning a list of dicts.

See `query()`.

Parameters **n3** – whether to return results in N3 notation, defaults to False.

4.2 RDFox runner

The `rdfox_runner.RDFoxRunner` class handles starting and stopping an RDFox instance with a specified set of input files and a script to run. It derives from `rdfox_runner.RDFoxEndpoint` so the same query methods can be used once it is running.

class rdfox_runner.**RDFoxRunner**(*input_files: Mapping[str, Union[str, pathlib.Path, TextIO]]*, *script: Union[List[str], str]*, *namespaces: Optional[Mapping] = None*, *wait: Optional[str] = None*, *working_dir: Optional[Union[str, pathlib.Path]] = None*, *rdfox_executable: Optional[Union[str, pathlib.Path]] = None*, *endpoint: Optional[rdfox_runner.rdfox_endpoint.RDFoxEndpoint] = None*)
Bases: `object`

Context manager to run RDFox in a temporary directory.

Parameters

- **input_files** – mapping of files {target path: source file} to set up in temporary working directory.
- **script** – RDFox commands to run, either as a list of strings or a single string.
- **namespaces** – dict of RDFlib namespaces to bind
- **wait** – whether to wait for RDFox to start the endpoint or exit when starting. If None, look for the presence of an "endpoint start" command in *script* and wait for the endpoint if found, wait for exit otherwise.
- **working_dir** – Path to setup command in, defaults to a temporary directory
- **rdfox_executable** – Path RDFox executable (default "RDFox")
- **endpoint** – RDFoxEndpoint instance to use (default None, meaning use the built in class). This can be used to customise the endpoint interface.

When used as a context manager, the *RDFoxRunner* instance returns *endpoint* for running queries etc. For more control a custom *RDFoxEndpoint* can be passed in. When the RDFox endpoint is started, the *connect()* method on the endpoint will be called with the connection string. The endpoint is available at the attribute *endpoint*.

files(*path*) → pathlib.Path
Return path to temporary directory.

Parameters **path** – path relative to the working directory

raise_for_errors()
Raise an exception if RDFox has reported an error.

"Critical" errors are reported. If the error policy is set to "stop", then errors that caused RDFox to stop are also reported.

send_quit()
Send "quit" command to RDFox.

start()
Start RDFox.

Parameters **wait_secs** – how many seconds to wait for RDFox to start.

stop()
Stop RDFox.

4.3 Generic command runner

The *rdfox_runner.CommandRunner* class is the building block for rdfox_runner, which handles setting up a temporary working directory and running a given command within it.

For example:

```
from io import StringIO
import time

input_files = {
    "a.txt": StringIO("hello world"),
}

# The -u is important for unbuffered output
command = ["python", "-u", "-m", "http.server", "8008"]

with CommandRunner(input_files, command):
    time.sleep(0.1)
    response = requests.get("http://localhost:8008/a.txt")

assert response.text == "hello world"
```

class rdfox_runner.**CommandRunner**(*input_files: Optional[Mapping[str, Union[str, pathlib.Path, TextIO]]] = None*, *command: Optional[Union[List, str, Callable]] = None*, *shell: bool = False*, *wait_before_enter: bool = False*, *wait_before_exit: bool = False*, *timeout: Optional[float] = None*, *working_dir: Optional[Union[str, pathlib.Path]] = None*, *output_callback: Optional[Callable] = None*)

Run a command in a temporary directory.

This can be used as a context manager, ensuring the temporary directory is cleaned up and the subprocess is stopped when finished with.

Parameters

- **input_files** – mapping with keys being the target path and value the source path.
- **command** – command to run, as passed to `subprocess.Popen`
- **shell** – whether to run command within shell
- **wait_before_enter** – whether to wait for command to complete before continuing with context manager body.

- **wait_before_exit** – whether to wait for command to complete by itself before terminating it, when leaving context manager body.
- **timeout** – timeout if *wait* is true.
- **working_dir** – Path to setup command in, defaults to a temporary directory
- **output_callback** – Callback on output from command.

The values in *input_files* can be:

- **a pathlib.Path or string – interpreted as a path to a file to** copy
- **a file-like object – read to provide the content for the temporary file.** This can be a io.StringIO object if you would like to provide a constant value

cleanup_files()

Cleanup temporary working directory, if needed.

The directory is only removed if it was newly created, not if it was passed in as `working_dir`.

files(*path*) → pathlib.Path

Return path to temporary directory.

Parameters **path** – path relative to the working directory

setup_files()

Setup the files ready to run the command.

If `working_dir` has been specified, it is created if it does not exist. Otherwise, a new temporary directory is created.

The files listed in `input_files` are copied into the working directory.

start()

Setup files and start the command running.

This is a convenience method to run *setup_files()* and *start_subprocess()* together, as needed.

start_subprocess()

Start the subprocess running.

If *wait* is true, wait for up to `timeout` seconds before continuing.

stop()

Stop the command and clean up files.

This is a convenience method to run *stop_subprocess()* and *cleanup_files()* together, as needed.

Raises **subprocess.CalledProcessError** – if the subprocess returns an error exit code.

stop_subprocess()

Stop the subprocess.

wait()

Wait for subprocess to exit.

Waits up to `timeout` seconds.

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX