razi Documentation

Release 2.0.0b0

Riccardo Vianello

Dec 23, 2017

Contents

Contents:

CHAPTER 1

Introduction

Razi provides extensions to SQLAlchemy to work with chemical databases. Currently, development is mainly focused on the functionalities provided by the RDKit extension for the PostgreSQL database, but support for additional database backends is in the plans.

1.1 Installation

Source code is available on GitHub. Either clone the repository or download a source package, change into the razi directory and type the following command:

```
$ python setup.py install
```

1.2 Documentation

This documentation is available from http://razi.readthedocs.org/.

1.3 License

Razi is released under the MIT License.

CHAPTER 2

Tutorials

2.1 Creating a chemical database

This tutorial is based on a similar document that is part of the RDKit official documentation and it illustrates how to build a chemical database and perform some simple search queries using Razi. The Python bindings for the RDKit libraries will be used in some data pre-processing steps, so you'll need to have them available on your system.

Some basic understanding of SQLAlchemy is assumed. For additional details, please refer to the excellent SQLALchemy ORM tutorial.

2.1.1 Download the ChEMBL compounds data

Download the ChEMBLdb database and decompress it:

```
$ wget ftp://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/releases/chembl_23/chembl_23_
↪chemreps.txt.gz
$ gunzip chembl_23_chemreps.txt.gz
```

2.1.2 Database creation

Then, create a new database and configure it to use the RDKit extension:

```
$ createdb -Udb_user razi_tutorial
$ psql razi_tutorial db_user

razi_tutorial=# create extension rdkit;
CREATE EXTENSION
razi_tutorial=# \q
```

The database username and password, together with the name of the database itself, are most likely to differ in your case. Please replace them with the values appropriate to your work environment.

2.1.3 Connection to the database

Start your python interpreter and configure a database connection:

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://db_user:db_password@host:1234/razi_tutorial')
```

also, define the database session factory object:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

2.1.4 Schema definition

For this tutorial we only need one single database entity, mapped to a python class. Following the SQLAlchemy's *declarative* style, a base class is first defined:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base(bind=engine)
```

then, the definition of the mapped entity follows:

```
from sqlalchemy import Column, Index, Integer, String
from razi.rdkit_postgresql.types import Mol

class Compound(Base):
    __tablename__ = 'compounds'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    structure = Column(Mol)

    __table_args__ = (
        Index('compounds_structure', 'structure',
              postgresql_using='gist'),
        )

    def __init__(self, name, structure):
        self.name = name
        self.structure = structure

    def __repr__(self):
        return '(%s) < %s >' % (self.name, self.structure)
```

to actually create the database schema, a method of the `Base` class `metadata` attribute is called:

```
Base.metadata.create_all()
```

In the present case this last command creates a table named `compounds` with columns `id`, `name` and `structure`, and it also includes the creation of a structural index on the column with type `Mol`.

2.1.5 Inserting data

The data from ChEMBL is formatted as a csv file, tab-separated and beginning with a header line:

```
$ head -n3  chembl_23_chemreps.txt
chembl_id   canonical_smiles        standard_inchi  standard_inchi_key
CHEMBL153534        Cc1cc(cn1C)c2csc(N=C(N)N)n2     InChI=1S/C10H13N5S/c1-6-3-7(4-
↪15(6)2)8-5 <snip>
CHEM1BL440060
↪CC[C@H](C)[C@H](NC(=O)[C@H](CC(C)C)NC(=O)[C@@H](NC(=O)[C@@H](N)CCSC) <snip>
```

To semplify the processing of these data records we define a namedtuple matching the input format:

```
from collections import namedtuple
Record = namedtuple('Record', 'chembl_id, smiles, inchi, inchi_key')
```

During this tutorial only a portion of the whole database is actually imported, and at the same time we want to make sure that troublesome SMILES strings are skipped (SMILES containing errors, compounds that are too big and/or other strings that the RDKit cartridge won't be able to process). File parsing and data filtering can therefore be performed with a function similar to the following:

```
import csv
from rdkit import Chem

def read_chembldb(filepath, limit=0):

    with open(filepath, 'rt') as inputfile:
        reader = csv.reader(inputfile, delimiter='\t', skipinitialspace=True)
        next(reader) # skip header

        for count, record in enumerate(map(Record._make, reader), 1):

            smiles = record.smiles

            # skip problematic smiles
            if len(smiles) > 300: continue
            smiles = smiles.replace('=N#N','=[N+]=[N-]')
            smiles = smiles.replace('N#N=','[N-]=[N+]=')
            if not Chem.MolFromSmiles(smiles):
                continue

            yield count, record.chembl_id, smiles
            if count == limit:
                break
```

The `read_chembldb` function above is a python generator, producing for each valid record a python tuple containing the record counter and the `chembl_id` and `smiles` strings.

With this function importing the compounds into the database reduces to a simple loop *(please note that depending on the available hardware resources importing the whole database may require a few hours; to keep this tutorial short we'll limit the processing to the first 25K compounds, a dataset size the usually corresponds to a few minutes)*:

```
session = Session()
for count, chembl_id, smiles in read_chembldb('chembl_08_chemreps.txt', 25000):
    compound = Compound(chembl_id, smiles)
    session.add(compound)
session.commit()
```

2.1.6 Querying the database

Finally, we can perform some queries. We can for example verify the number of compounds actually loaded into the database:

```
>>> print session.query(Compound).count()
24956
```

or display the first 5 compounds:

```
>>> for compound in session.query(Compound)[:5]:
...     print compound
...
(CHEMBL6582) < NC(=O)c1cc(-c2ccc(Cl)cc2)nnc1Cl >
(CHEMBL6583) < Cc1cnc(NS(c2cccc3c(N(C)C)cccc23)(=O)=O)cn1 >
(CHEMBL6584) < CN(C)/C=N/c1nc(/N=C\N(C)C)c2c(ncc(Sc3cc(Cl)c(Cl)cc3)n2)n1 >
(CHEMBL6585) < CC12C(C[C@@H](I)[C@@H]1O)C1C(c3ccc(O)cc3CC1)CC2 >
(CHEMBL6637) < C/C(=C\Cn1oc(=O)[nH]c1=O)c1ccc(OCCc2nc(-c3ccc(C(F)(F)F)cc3)oc2C)cc1 >
```

Finally (and hopefully more interestingly), here's a first example of a more chemistry-aware query, searching the database for a given substructure:

```
>>> # which compounds contain 'c1cccc2c1nncc2' as a substructure?
...
>>> subset = session.query(Compound)
>>> subset = subset.filter(Compound.structure.contains('c1cccc2c1nncc2'))
>>> print subset.count()
2
>>> for compound in subset: print compound
...
(CHEMBL12112) < CC(C)Sc1ccc(CC2CCN(C3CCN(C(=O)c4cnnc5ccccc54)CC3)CC2)cc1 >
(CHEMBL26025) < Cc1cccc(NC(=O)Nc2ccc3nnccc3c2)c1 >
```

Please notice how the SQLAlchemy's ORM API allows the incremental specification of the filtering clause (or clauses) associated to the main selection query and how the `subset` instance is actually used twice, in two distinct queries (to compute the number of record matching the clause and to retrieve the actual records). In addition to this, the returned records can also serve as the basis for further queries, also using the chemical functions provided by the database backend:

```
>>> for compound in subset:
...     # run a query to compute the molecular weight for this compound
...     print session.scalar(compound.structure.mw)
...
488.701
278.315
```

2.2 Example similarity queries

This tutorial is again based on a similar document available from the RDKit wiki and it illustrates how to use Razi to perform structure similarity queries on a chemical database.

2.2.1 Database creation

Create a new database for this tutorial:

```
$ createdb -Udb_user -Ttemplate_rdkit razi_tutorial
```

The database user name and password, together with the name of the database itself, are most likely to change in your case. Just replace them consistently with the values appropriate to your work environment.

2.2.2 Connection to the database

Start your python interpreter and configure a database connection:

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://db_user:db_password@host:1234/razi_tutorial')
```

also, define the database session factory object:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

2.2.3 Schema definition

The schema is composed of one single database entity, mapped to a python class. As described in the first tutorial, a base class is first defined:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base(bind=engine)
```

then, the definition of the mapped entity follows:

```
from sqlalchemy import Column, Integer, String
from razi.orm import ChemColumn
from razi.chemtypes import Molecule, BitFingerprint

class Compound(Base):
    __tablename__='compounds'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    structure = ChemColumn(Molecule)
    atompair = ChemColumn(BitFingerprint)
    torsion = ChemColumn(BitFingerprint)
    morgan = ChemColumn(BitFingerprint)

    def __init__(self, name, structure):
        self.name = name
        self.structure = structure
        self.atompair = self.structure.atompair_b()
        self.torsion = self.structure.torsion_b()
        self.morgan = self.structure.morgan_b(2)

    def __repr__(self):
        return '(%s) < %s >' % (self.name, self.structure)
```

and the database schema is created:

```
Base.metadata.create_all()
```

In the present case this last command creates the `compounds` table and also implicitly includes the creation of indices on the columns with types `Molecule` and `BitFingerprint`. Please notice how in the constructor the fingerprint fields are initialized by database backend expressions invoked on the `structure` column.

2.2.4 Inserting data

To populate the database the same data and code we used in the first tutorial is used again (this time we import a more extended dataset, but still small enough to keep the processing time acceptably short for a tutorial. Feel free to modify the number of compounds imported, just be aware that some results may change with the imported dataset):

```
session = Session()
for count, chembl_id, smiles in read_chembldb('chembl_08_chemreps.txt', 50000):
    compound = Compound(chembl_id, smiles)
    session.add(compound)
session.commit()
```

2.2.5 Querying the database

We are now ready to reproduce the similarity queries executed in the original RDKit tutorial.

Similarity search using Morgan fingerprints

Tanimoto similarity using the current similarity cutoff:

```
>>> # search compounds similar to the following (known to be in the db):
>>> query_cmpnd = 'CN(C)/C=N/c1nc(/N=C\N(C)C)c2c(ncc(Sc3cc(Cl)c(Cl)cc3)n2)n1'
>>> # compute a binary fingerprint for the query compound
>>> # (actually, it's the corresponding SQL expression whose value is to be computed␣
↪by the db)
>>> from razi.functions import functions
>>> from razi.expression import TxtMoleculeElement
>>> query_bfp = functions.morgan_b(TxtMoleculeElement(query_cmpnd), 2)
>>> # determine the number of compunds with Tanimoto similarity above
>>> # the current threshold value:
>>> print session.query(Compound).filter(Compound.morgan.tanimoto_similar(query_bfp)).
↪count()
2
```

Or using the Dice similarity:

```
>>> print session.query(Compound).filter(Compound.morgan.dice_similar(query_bfp)).
↪count()
6
```

Including the similarity values in the search results:

```
>>> constraint = Compound.morgan.dice_similar(query_bfp)
>>> dice_sml = Compound.morgan.dice_similarity(query_bfp).label('dice')
>>> from sqlalchemy import desc
>>> results = session.query(Compound, dice_sml).filter(constraint).order_by(desc(dice_
↪sml))
>>> for row in results: print row.Compound, row.dice
(CHEMBL6584) < CN(C)/C=N/c1nc(/N=C\N(C)C)c2c(ncc(Sc3cc(Cl)c(Cl)cc3)n2)n1 > 1.0
(CHEMBL6544) < Nc1nc(N)c2c(ncc(Sc3cc(Cl)c(Cl)cc3)n2)n1 > 0.666666666667
```

```
(CHEMBL6618) < Nc1nc(N)c2c(ncc(Sc3cc4c(cccc4)cc3)n2)n1 > 0.52380952381
(CHEMBL6465) < Nc1nc(N)c2c(ncc(Sc3cc(Cl)c(Cl)cc3Cl)n2)n1 > 0.506024096386
(CHEMBL6631) < COc1ccc(Sc2cnc3c(c(N)nc(N)n3)n2)cc1 > 0.5
(CHEMBL57035) < CCN(CC)CCCNc1ncc2cc(-c3c(Cl)cccc3Cl)c(/N=C\N(C)C)nc2n1 > 0.5
```

Similarity search using other fingerprints

At this point using the other fingerprint types basically only requires redefining the `query_bfp` fingerprint and the query constraint. For example, Tanimoto similarity between topological torsion fingerprints using the current similarity cutoff:

```
>>> query_bfp = functions.torsion_b(TxtMoleculeElement(query_cmpnd))
>>> constraint = Compound.torsion.tanimoto_similar(query_bfp)
>>> tanimoto_sml = Compound.torsion.tanimoto_similarity(query_bfp).label('tanimoto')
>>> results = session.query(Compound, tanimoto_sml).filter(constraint).order_
↪by(desc(tanimoto_sml))
```

and Tanimoto similarity between atom-pair fingerprints using the current similarity cutoff is almost identical:

```
>>> query_bfp = functions.atompair_b(TxtMoleculeElement(query_cmpnd))
>>> constraint = Compound.atompair.tanimoto_similar(query_bfp)
>>> tanimoto_sml = Compound.atompair.tanimoto_similarity(query_bfp).label('tanimoto')
>>> results = session.query(Compound, tanimoto_sml).filter(constraint).order_
↪by(desc(tanimoto_sml))
```

Changing the similarity cutoff values

The threshold values used by the Tanimoto and Dice filter operators are mapped to two expressions defined in module `razi.postgresql_rdkit`:

```
>>> from razi.postgresql_rdkit import tanimoto_threshold, dice_threshold
>>> session.scalar(tanimoto_threshold), session.scalar(dice_threshold)
(u'0.5', u'0.5')
```

The same expressions provide a mechanism to set a different cutoff:

```
>>> session.execute(tanimoto_threshold.set(0.65))
<sqlalchemy.engine.base.ResultProxy object at 0x1bbc5a10>
>>> session.scalar(tanimoto_threshold), session.scalar(dice_threshold)
(u'0.65', u'0.5')
```

2.3 Example structure queries

This tutorial is based on a similar document available from the RDKit wiki and it illustrates how to use Razi to perform substructure and superstructure queries on a chemical database.

No dedicated database is created for this tutorial. The same database used in the "*Example similarity queries*" tutorial can be used instead. If you are no longer connected to the database, the connection configuration and schema definition must be entered again. If you are still connected just skip to section "*Querying the database*".

2.3.1 Connection to the database

Start your python interpreter and configure a database connection:

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://db_user:db_password@host:1234/razi_tutorial')
```

also, define the database session factory object:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

2.3.2 Schema definition

Then define the mapping to the database table:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base(bind=engine)

from sqlalchemy import Column, Integer, String
from razi.orm import ChemColumn
from razi.chemtypes import Molecule, BitFingerprint

class Compound(Base):
    __tablename__='compounds'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    structure = ChemColumn(Molecule)
    atompair = ChemColumn(BitFingerprint)
    torsion = ChemColumn(BitFingerprint)
    morgan = ChemColumn(BitFingerprint)

    def __init__(self, name, structure):
        self.name = name
        self.structure = structure
        self.atompair = self.structure.atompair_b()
        self.torsion = self.structure.torsion_b()
        self.morgan = self.structure.morgan_b(2)


    def __init__(self, name, structure):
        self.name = name
        self.structure = structure

    def __repr__(self):
        return '(%s) < %s >' % (self.name, self.structure)
```

2.3.3 Querying the database

Substructure queries

Retrieve the number of molecules containing a triazine:

```
>>> constraint = Compound.structure.contains('c1ncncn1')
>>> print session.query(Compound).filter(constraint).count()
56
```

Retrieve the number of molecules containing a coumarin:

```
>>> constraint = Compound.structure.contains('O=C1OC2=CC=CC=C2C=C1')
>>> print session.query(Compound).filter(constraint).count()
166
```

Get the first 10 of those:

```
>>> for c in session.query(Compound).filter(constraint)[:10]: print c
(CHEMBL58793) < OC(=O)CCCCc1cc(=O)oc2c1ccc(O)c2CN1CCCC1 >
(CHEMBL56784) < [Na+].COc1ccc(-c2c3n(c4c(=O)oc5cc(OS([O-
↪])(=O)=O)c(OC)cc5c42)CCc2cc(OC)c(OC)cc2-3)cc1O >
(CHEMBL54909) < COc1cc2ccc(=O)oc2c(O)c1O >
(CHEMBL50150) < COc1ccc(CCn2cc(-c3ccc(OC)c(OC)c3)c3c4c(oc(=O)c23)cc(OC)c(OC)c4)cc1OC >
(CHEMBL50201) < CC(C)CCc1c(O)ccc2c1oc(=O)cc2 >
(CHEMBL59509) < OC(=O)CCCCc1cc(=O)oc2c1ccc(O)c2CNc1ccccc1 >
(CHEMBL57330) < CCCN(C1COc2cccc(OC)c2C1)CCCCNC(=O)c1c2c(oc(=O)c1)c1c3c(c2)CCCN3CCC1 >
(CHEMBL57173) < C/C(CC/C=C(\C)C1=CC(=O)C(C)(C)O1)=C\COc1cc2oc(=O)ccc2cc1 >
(CHEMBL57138) < COc1ccc(-c2c3n(c4c(=O)oc5cc(O)c(OC)cc5c42)CCc2c(OC)c(OC)c(OC)cc2-
↪3)cc1O >
(CHEMBL56918) < C/C(=C\COc1ccc2c(oc(=O)cc2)c1)C1=CC(=O)C(C)(C)O1 >
```

Including property filters

Differently from the original RDKit tutorial, chemical descriptor were not introduced into the current database schema. Filtering based on chemical properties can still be introduced, with the difference that these properties are in this case computed while processing the query:

```
>>> mw = Compound.structure.mw.label('mw')
>>> logp = Compound.structure.logp.label('logp')
>>> # compounds containing coumarin as substructure, with molecular weight
>>> # not above 200, ordered by ascending estimated logp
>>> subset = session.query(Compound, mw, logp).filter(constraint).filter(mw <= 200).
↪order_by(logp)
>>> for row in subset: print row.Compound.name, row.mw, row.logp
CHEMBL32810 178.143 1.2042
CHEMBL51628 162.144 1.4986
CHEMBL12252 192.17 1.51262
CHEMBL6466 146.145 1.793
CHEMBL49732 176.171 1.8016
CHEMBL12626 176.171 1.80702
CHEMBL12208 176.171 1.80702
CHEMBL12279 160.172 2.10142
CHEMBL12636 190.198 2.11002
CHEMBL19240 190.198 2.11544
CHEMBL53569 186.166 2.5392
CHEMBL6355 196.205 2.9462
```

Other kinds of structural searches

Superstructure queries

Look for molecules in the database that are substructures of a query (i.e. where the query is a superstructure of the database molecule):

```
>>> constraint = Compound.structure.contained_in(
↪'c1ccc(C(COC(c2c(=O)oc3c(ccc(O)c3)c2)=O)=O)cc1')
>>> for c in session.query(Compound).filter(constraint)[:10]: print c
(CHEMBL51628) < O=c1oc2cc(O)ccc2cc1 >
(CHEMBL44857) < CCCOC(=O)C >
(CHEMBL44215) < CCOC=O >
(CHEMBL545) < CCO >
(CHEMBL14688) < CO >
(CHEMBL17564) < C >
(CHEMBL15972) < O=Cc1ccccc1 >
(CHEMBL14687) < CCCO >
(CHEMBL16264) < CCOCC >
(CHEMBL14079) < COC(=O)C >
```

SMARTS-based Queries

`contains` substructure queries are by default executed using SMILES semantics. In order to do SMARTS-based queries, one may use `match`, as this example shows:

```
>>> constraint = Compound.structure.match('cc(c)NC(=O)N')
>>> for c in session.query(Compound).filter(constraint)[:10]: print c
(CHEMBL6997) <␣
↪CSCC[C@H](NC(Nc1cc(C)ccc1)=O)C(=O)N[C@@H](CC(C)C)C(N[C@@H](Cc1ccccc1)C(O)=O)=O >
(CHEMBL6500) < CCOC(c1ccc(NC(=O)Nc2c(C)cc3c(c2)C(C)(C)CC(C)(C)S3)cc1)=O >
(CHEMBL6218) < COc1cc2c(c(N)nc(N3CCN(C(=O)Nc4ccccc4)CC3)n2)cc1OC >
(CHEMBL7610) <␣
↪COc1ccc(C[C@H](NC(Nc2cc3n(Cc4c(Cl)cccc4Cl)cc(CN4CCCC4)c3cc2)=O)C(N[C@@H](CCCNC(=N)N)C(NCc2ccccc2)=O
↪>
(CHEMBL7667) < CCCCNS(=NC(=O)Nc1ccc(Cl)cc1)(=O)c1ccc(C)cc1 >
(CHEMBL7955) < CCNS(=NC(=O)Nc1ccc(Cl)cc1)(=O)c1ccc(C)cc1 >
(CHEMBL7851) < Cc1c(Cl)c(C)cc(S(N)(=NC(=O)Nc2ccc(Cl)cc2)=O)c1 >
(CHEMBL7627) <␣
↪COc1ccc(C[C@H](NC(Nc2cc3n(Cc4ccc(F)cc4)cc(CNC4CCCC4)c3cc2)=O)C(N[C@@H](CCCN=C(N)N)C(NCc2ccccc2)=O)=
↪>
(CHEMBL7346) < CCOC(c1ccc(NC(=O)Nc2cc3c(cc2)N(C)C(C)(C)C=C3C)cc1)=O >
(CHEMBL7520) <␣
↪CSCC[C@H](NC(Nc1ccccc1)=O)C(N[C@@H](CC(C)C)C(N[C@@H](Cc1ccccc1)C(O)=O)=O)=O >
```

Exact match queries

Matching full structures is supported by using `equals`:

```
>>> print session.query(Compound).filter(Compound.structure.equals('c1ncncn1')).
↪count()
```

or by just using the equality operator ==:

```
>>> print session.query(Compound).filter(Compound.structure == 'c1ncncn1').count()
```

CHAPTER 3

API Reference

3.1 Column and Data Types

3.2 Functions

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

R