Getting Started with the RDKit in Python

Original Author: Greg Landrum
Version: Q3 2008

Table of Contents

1 What is this?

This document is intended to provide an overview of how one can use the RDKit functionality from Python. It's not comprehensive and it's not a manual.

If you find mistakes, or have suggestions for improvements, please either fix them yourselves in the source document (the .odt file) or send them to the mailing list: rdkit-devel@lists.sourceforge.net

2 Reading and Writing Molecules

2.1 Reading single molecules

The majority of the basic molecular functionality is found in module Chem:

```
>>> import Chem
>>> Chem.rdBase.AttachFileToLog('rdApp.error','stdout')
```

Individual molecules can be constructed using a variety of approaches:

```
>>> m = Chem.MolFromSmiles('Cc1ccccc1')
>>> m = Chem.MolFromMolFile('data/input.mol')
>>> stringWithMolData=file('data/input.mol','r').read()
>>> m = Chem.MolFromMolBlock(stringWithMolData)
```

All of these functions return a Mol object on success:

```
>>> m
<Chem.rdchem.Mol object at 0x...>
```

or None on failure:

```
>>> m = Chem.MolFromMolFile('data/invalid.mol')
#[12:20:12] Explicit valence for atom # 6 N greater than permitted
>>> m is None
True
```

An attempt is made to provide sensible error messages:

```
>>> m = Chem.MolFromSmiles('CO(C)C')
#[12:18:01] Explicit valence for atom # 1 O greater than permitted
>>  m is None
True
>>> m = Chem.MolFromSmiles('c1cc1')
#[12:20:41] Can't kekulize mol
>>> m is None
True
```

2.2 Reading sets of molecules

Groups of molecules are read using a Supplier:

```
>>> suppl = Chem.SDMolSupplier('data/5ht3ligs.sdf')
>>> for mol in suppl:
...     print mol.GetNumAtoms()
...
20
24
```

```
24
26
```

You can easily produce lists of molecules from a Supplier:

```
>>> mols = [x for x in suppl]
>>> len(mols)
4
```

or just treat the Supplier itself as a random-access object:

```
>>> suppl[0].GetNumAtoms()
20
```

2.3 Writing molecules

Single molecules can be converted to text using several functions present in the Chem module. For example, for SMILES:

```
>>> m = Chem.MolFromMolFile('data/chiral.mol')
>>> Chem.MolToSmiles(m)
'CC(O)c1ccccc1'
>>> Chem.MolToSmiles(m,isomericSmiles=True)
'C[C@H](O)c1ccccc1'
```

Note that the SMILES provided is canonical, so the output should be the same no matter how a particular molecule is input:

```
>>> Chem.MolToSmiles(Chem.MolFromSmiles('C1=CC=CN=C1'))
'c1ccncc1'
>>> Chem.MolToSmiles(Chem.MolFromSmiles('c1cccnc1'))
'c1ccncc1'
>>> Chem.MolToSmiles(Chem.MolFromSmiles('n1ccccc1'))
'c1ccncc1'
```

If you'd like to have the Kekule form of the SMILES, first Kekulize the molecule, then use the "kekuleSmiles" option:

```
>>> Chem.Kekulize(m)
>>> Chem.MolToSmiles(m,kekuleSmiles=True)
'CC(O)C1=CC=CC=C1'
```

Note: as of this writing (Aug 2008), the smiles provided when one requests kekuleSmiles are not canonical. The limitation is not in the SMILES generation, but in the kekulization itself.

MDL Mol blocks are also available:

```
>>> m2 = Chem.MolFromSmiles('C1CCC1')
>>> print Chem.MolToMolBlock(m2)

     RDKit

  4  4  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

To include names in the mol blocks, set the molecule's "_Name" property:

```
>>> m2.SetProp("_Name","cyclobutane")
>>> print Chem.MolToMolBlock(m2)
cyclobutane
```

```
     RDKit

  4  4  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

It's usually preferable to have a depiction in the Mol block, this can be generated using functionality in the AllChem module (see the note on the AllChem module). You can either include 2D coordinates (i.e. a depiction):

```
>>> from Chem import AllChem
>>> AllChem.Compute2DCoords(m2)
0
>>> print Chem.MolToMolBlock(m2)
cyclobutane
     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

Or you can add 3D coordinates by embedding the molecule:

```
>>> AllChem.EmbedMolecule(m2)
0
>>> AllChem.UFFOptimizeMolecule(m2)
0
>>> print Chem.MolToMolBlock(m2)
cyclobutane
     RDKit          3D

  4  4  0  0  0  0  0  0  0  0999 V2000
   -0.7951    0.5727   -0.2680 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3783   -0.9192   -0.2367 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7809   -0.5398    0.6589 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.3924    0.8863    0.6160 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

The optimization step isn't necessary, but it substantially improves the quality of the conformation.

If you'd like to write the molecules to a file, use Python file objects:

```
>>> print >>file('data/foo.mol','w+'),Chem.MolToMolBlock(m2)
>>>
```

2.4 Writing sets of molecules

Multiple molecules can be written to a file using a Writer object:

```
>>> w = Chem.SDWriter('data/foo.sdf')
>>> for m in mols: w.write(m)
...
>>>
```

Other available Writers include the SmilesWriter and the TDTWriter.

3 Working with Molecules

3.1 Looping over Atoms and Bonds

Once you have a molecule, it's easy to loop over its atoms and bonds:

```
>>> m  = Chem.MolFromSmiles('C1OC1')
>>> for atom in m.GetAtoms():
...     print atom.GetAtomicNum()
...
6
8
6
>>> print m.GetBonds()[0].GetBondType()
SINGLE
```

You can also request individual bonds or atoms:

```
>>> m.GetAtomWithIdx(0).GetSymbol()
'C'
>>> m.GetAtomWithIdx(0).GetExplicitValence()
2
>>> m.GetBondWithIdx(0).GetBeginAtomIdx()
0
>>> m.GetBondWithIdx(0).GetEndAtomIdx()
1
>>> m.GetBondBetweenAtoms(0,1).GetBondType()
Chem.rdchem.BondType.SINGLE
```

Atoms keep track of their neighbors:

```
>>> atom = m.GetAtomWithIdx(0)
>>> [x.GetAtomicNum() for x in atom.GetNeighbors()]
[8, 6]
>>> len(x.GetBonds())
2
```

3.2 Ring Information

Atoms and bonds both carry information about the molecule's rings:

```
>>> m = Chem.MolFromSmiles('OC1C2C1CC2')
>>> m.GetAtomWithIdx(0).IsInRing()
False
>>> m.GetAtomWithIdx(1).IsInRing()
True
>>> m.GetAtomWithIdx(2).IsInRingSize(3)
True
>>> m.GetAtomWithIdx(2).IsInRingSize(4)
True
```

```
>>> m.GetAtomWithIdx(2).IsInRingSize(5)
False
>>> m.GetBondWithIdx(1).IsInRingSize(3)
True
>>> m.GetBondWithIdx(1).IsInRing()
True
```
But note that the information is only about the smallest rings:
```
>>> m.GetAtomWithIdx(1).IsInRingSize(5)
False
```
More detail about the smallest set of smallest rings (SSSR) is available:
```
>>> ssr = Chem.GetSymmSSSR(m)
>>> len(ssr)
2
>>> list(ssr[0])
[1, 2, 3]
>>> list(ssr[1])
[4, 5, 2, 3]
```
As the name indicates, this is a symmetrized SSSR; if you are interested in the number of "true" SSSR, use the GetSSSR function.
```
>>> Chem.GetSSSR(m)
2
```
The distinction between symmetrized and non-symmetrized SSSR is discussed in more detail below.
For more efficient queries about a molecule's ring systems (avoiding repeated calls to Mol.GetAtomWithIdx), use the RingInfo class:
```
>>> m = Chem.MolFromSmiles('OC1C2C1CC2')
>>> ri = m.GetRingInfo()
>>> ri.NumAtomRings(0)
0
>>> ri.NumAtomRings(1)
1
>>> ri.NumAtomRings(2)
2
>>> ri.IsAtomInRingOfSize(1,3)
True
>>> ri.IsBondInRingOfSize(1,3)
True
```

3.3 Modifying molecules

Normally molecules are stored in the RDKit with the hydrogen atoms implicit (e.g. not explicitly present in the molecular graph. When it is useful to have the hydrogens explicitly present, for example when generating or optimizing the 3D geometry, the AddHs function can be used:
```
>>> m=Chem.MolFromSmiles('CCO')
>>> m.GetNumAtoms()
3
>>> m2 = Chem.AddHs(m)
>>> m2.GetNumAtoms()
9
```
The Hs can be removed again using the RemoveHs function:
```
>>> m3 = Chem.RemoveHs(m2)
>>> m3.GetNumAtoms()
3
```
RDKit molecules are usually stored with the bonds in aromatic rings having aromatic bond types. This can be changed with the Kekulize function:
```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetBondWithIdx(0).GetBondType()
```

```
Chem.rdchem.BondType.AROMATIC
>>> Chem.Kekulize(m)
>>> m.GetBondWithIdx(0).GetBondType()
Chem.rdchem.BondType.DOUBLE
>>> m.GetBondWithIdx(1).GetBondType()
Chem.rdchem.BondType.SINGLE
```

The bonds are still marked as being aromatic:

```
>>> m.GetBondWithIdx(1).GetIsAromatic()
True
```

and can be restored to the aromatic bond type using the SanitizeMol function:

```
>>> Chem.SanitizeMol(m)
>>> m.GetBondWithIdx(0).GetBondType()
Chem.rdchem.BondType.AROMATIC
```

3.4 Working with 2D molecules: Generating Depictions

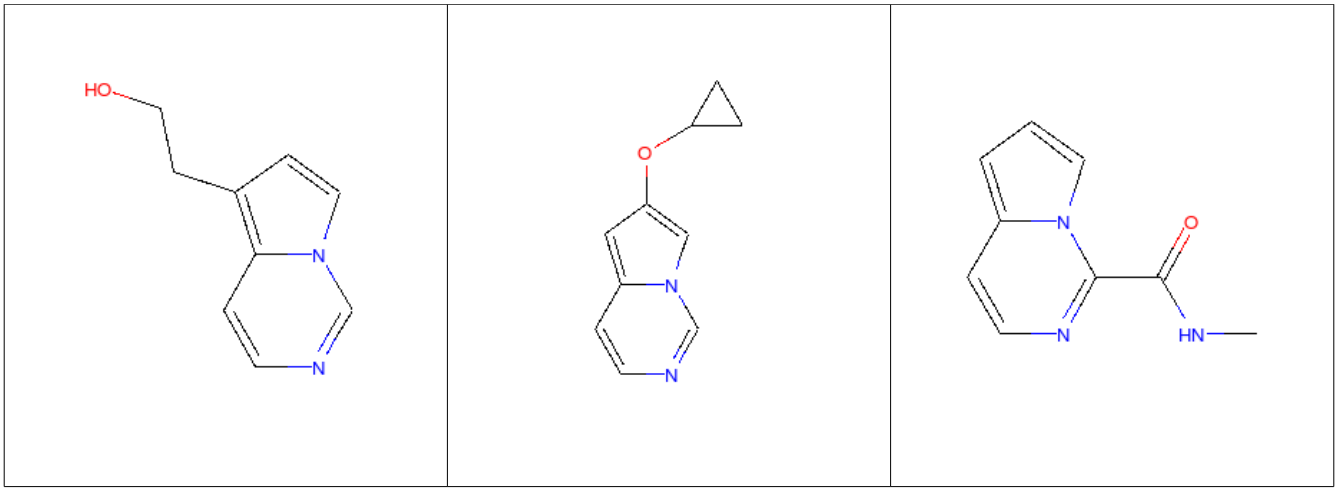
The RDKit has a library for generating depictions (sets of 2D) coordinates for molecules. This library, which is part of the AllChem module, is accessed using the Compute2DCoords function:

```
>>> m = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(m)
0
```

The 2D conformation is constructed in a canonical orientation and is built to minimize intramolecular clashes, i.e. to maximize the clarity of the drawing.
If you have a set of molecules that share a common template and you'd like to align them to that template, you can do so as follows:

```
>>> template = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(template)
0
>>> conf = template.GetConformer(0)
>>> coords = []
>>> for i in range(template.GetNumAtoms()):
...     pos = conf.GetAtomPosition(i)
...     coords.append(pos)
...
>>> import Geometry
>>> coords = [Geometry.Point2D(pt.x,pt.y) for pt in coords]
>>> m = Chem.MolFromSmiles('c1nccc2n1ccc2CCO')
>>> match = m.GetSubstructMatch(template)
>>> coordDict = {}
>>> for i,idx in enumerate(match): coordDict[idx]=coords[i]
...
>>> AllChem.Compute2DCoords(m,coordMap=coordDict)
0
```

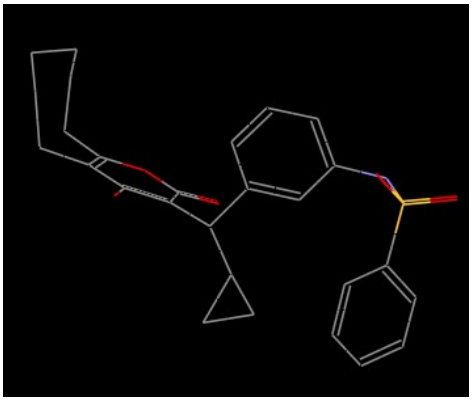
Running this process for a couple of other molecules gives the following depictions:

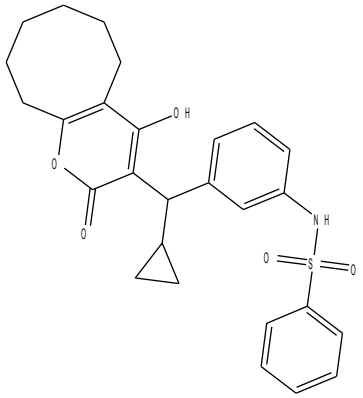


Note that this same functionality is available with more polish and error checking via the function AlignDepict, which is in the Chem.ChemUtils.AlignDepict module.

Another option for Compute2DCoords allows you to generate 2D depictions for molecules that closely mimic 3D conformations. This is available using the function AllChem.GenerateDepictionMatching3DStructure.

Here is an illustration of the results using the ligand from PDB structure 7UPJ:





More fine-grained control can be obtained using the core function AllChem.Compute2DCoordsMimicDistmat, but that is beyond the scope of this document. See the implementation of GenerateDepictionMatching3DStructure in AllChem.py for an example of how it is used.

3.5 Working with 3D Molecules

The RDKit can generate conformations for molecules using distance geometry.[1] The algorithm followed is:

1. The molecule's distance bounds matrix is calculated based on the connection table and a set of rules.
2. The bounds matrix is smoothed using a triangle-bounds smoothing algorithm.
3. A random distance matrix that satisfies the bounds matrix is generated.

1 Blaney, J. M.; Dixon, J. S. "Distance Geometry in Molecular Modeling". *Reviews in Computation Chemistry*; VCH: New York, 1994.

4. This distance matrix is embedded in 3D dimensions (producing coordinates for each atom).
5. The resulting coordinates are cleaned up somewhat using a crude force field and the bounds matrix.

Multiple conformations can be generated by repeating steps 4 and 5 several times, using a different random distance matrix each time.

Note that the conformations that result from this procedure tend to be fairly ugly. They should be cleaned up using a force field. This can be done within the RDKit using its implementation of the Universal Force Field (UFF).[2]

The full process of embedding and optimizing a molecule is easier than all the above verbiage makes it sound:

```
>>> m = Chem.MolFromSmiles('C1CCC1OC')
>>> m2=Chem.AddHs(m)
>>> AllChem.EmbedMolecule(m2)
0
>>> AllChem.UFFOptimizeMolecule(m2)
0
```

Disclaimer/Warning: Conformation generation is a difficult and subtle task. The 2D->3D conversion provided within the RDKit is not intended to be a replacement for a "real" conformational analysis tool; it merely provides quick 3D structures for cases when they are required.

3.6 Preserving Molecules

Molecules can be converted to and from text using Python's pickling machinery:

```
>>> m = Chem.MolFromSmiles('c1ccncc1')
>>> import cPickle
>>> pkl = cPickle.dumps(m)
>>> type(pkl)
<type 'str'>
>>> m2=cPickle.loads(pkl)
>>> Chem.MolToSmiles(m2)
'c1ccncc1'
```

The RDKit pickle format is fairly compact and it is much, much faster to build a molecule from a pickle than from a Mol file or SMILES string, so storing molecules you will be working with repeatedly as pickles can be a good idea.
The raw binary data that is encapsulated in a pickle can also be directly obtained from a molecule:

```
>>> binStr = m.ToBinary()
```

This can be used to reconstruct molecules using the Chem.Mol constructor:

```
>>> m2 = Chem.Mol(binStr)
>>> Chem.MolToSmiles(m2)
'c1ccncc1'
>>> len(binStr)
163
>>> len(pkl)
647
```

Note that this huge difference in text length is because we didn't tell python to use its most efficient representation of the pickle:

```
>>> pkl = cPickle.dumps(m,2)
>>> len(pkl)
191
```

2 Rappé, A. K.; Casewit, C. J.; Colwell, K. S.; Goddard III, W. A.; Skiff, W. M. "UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations". *J. Am. Chem. Soc.* **114:**10024-35 (1992) .

The small overhead associated with python's pickling machinery normally doesn't end up making much of a difference for collections of larger molecules (the extra data associated with the pickle is independent of the size of the molecule, while the binary string increases in length as the molecule gets larger).

Tip: The performance difference associated with storing molecules in a pickled form on disk instead of constantly reparsing an SD file or SMILES table is difficult to overstate. In a test I just ran on my laptop, loading a set of 699 drug-like molecules from an SD file took 10.8 seconds; loading the same molecules from a pickle file took 0.7 seconds. The pickle file is also smaller – 1/3 the size of the SD file – but this difference is not always so dramatic (it's a particularly fat SD file).

4 Substructure Searching

Substructure matching can be done using query molecules built from SMARTS:

```
>>> m = Chem.MolFromSmiles('c1ccccc1O')
>>> patt = Chem.MolFromSmarts('ccO')
>>> m.HasSubstructMatch(patt)
True
>>> m.GetSubstructMatch(patt)
(0, 5, 6)  #<- atom indices in m, ordered as patt's atoms
>>> m.GetSubstructMatches(patt)
((0, 5, 6), (4, 5, 6))  #<- all of m's substructure matches
```

This can be used to easily filter lists of molecules:

```
>>> suppl = Chem.SDMolSupplier('data/actives_5ht3.sdf')
>>> patt = Chem.MolFromSmarts('c[NH1]')
>>> matches = []
>>> for mol in suppl:
...     if mol.HasSubstructMatch(patt):
...             matches.append(mol)
...
>>> len(matches)
22
```

We can write the same thing more compactly using Python's list comprehension syntax:

```
>>> matches = [x for x in suppl if x.HasSubstructMatch(patt)]
>>> len(matches)
22
```

Substructure matching can also be done using molecules built from SMILES instead of SMARTS:

```
>>> m = Chem.MolFromSmiles('C1=CC=CC=C1OC')
>>> m.HasSubstructMatch(Chem.MolFromSmarts('CO'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CO'))
True
```

But don't forget that the semantics of the two languages are not exactly equivalent:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('COC'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmarts('COC'))
False
>>> m.HasSubstructMatch(Chem.MolFromSmarts('COc'))  #<- need an aromatic C
True
```

There's also functionality for using the substructure machinery for doing quick molecular transformations. These transformations include deleting substructures:

```
>>> m = Chem.MolFromSmiles('CC(=O)O')
```

```
>>> patt = Chem.MolFromSmarts('C(=O)[OH]')
>>> rm = AllChem.DeleteSubstructs(m,patt)
>>> Chem.MolToSmiles(rm)
'C'
```

replacing substructures:

```
>>> repl = Chem.MolFromSmiles('OC')
>>> patt = Chem.MolFromSmarts('[$(NC(=O))]')
>>> m = Chem.MolFromSmiles('CC(=O)N')
>>> rms = AllChem.ReplaceSubstructs(m,patt,repl)
>>> rms
(<Chem.rdchem.Mol object at 0x...>,)
>>> Chem.MolToSmiles(rms[0])
'COC(=O)C'
```

as well as simple SAR-table transformations like removing side chains:

```
>>> m1 = Chem.MolFromSmiles('BrCCc1cncnc1C(=O)O')
>>> core = Chem.MolFromSmiles('c1cncnc1')
>>> tmp = Chem.ReplaceSidechains(m1,core)
>>> Chem.MolToSmiles(tmp)
'[*]c1cncnc1[*]'
```

and removing cores:

```
>>> tmp = Chem.ReplaceCore(m1,core)
>>> Chem.MolToSmiles(tmp)
'[*]CCBr.[*]C(=O)O'
```

To get more detail about the sidechains (e.g. sidechain labels), use isomeric smiles:

```
>>> Chem.MolToSmiles(tmp,True)
'[1*]CCBr.[2*]C(=O)O'
```

Note that these transformation functions are intended to provide an easy way to make simple modifications to molecules. For more complex transformations, use the chemical reaction functionality.

5 Fingerprinting and Molecular Similarity

The RDKit has a variety of built-in functionality for generating molecular fingerprints and using them to calculate molecular similarity.

5.1 Topological Fingerprints

```
>>> import DataStructs
>>> from Chem.Fingerprints import FingerprintMols
>>> ms = [Chem.MolFromSmiles('CCOC'), Chem.MolFromSmiles('CCO'),
...  Chem.MolFromSmiles('COC')]
>>> fps = [FingerprintMols.FingerprintMol(x) for x in ms]
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1])
0.625
>>> DataStructs.FingerprintSimilarity(fps[0],fps[2])
0.5
>>> DataStructs.FingerprintSimilarity(fps[1],fps[2])
0.384...
```

The fingerprinting algorithm used is similar to that used in the Daylight fingerprinter: it identifies and hashes topological paths (e.g. along bonds) in the molecule and then uses them to set bits in a fingerprint of user-specified lengths. After all paths have been identified, the fingerprint is typically folded down until a particular density of set bits is obtained. The default set of parameters used by the fingerprinter is:

- minimum path size: 1 bond
- maximum path size: 7 bonds
- fingerprint size: 2048 bits
- number of bits set per hash: 4
- minimum fingerprint size: 64 bits
- target on-bit density 0.3

You can control these by calling Chem.RDKitFingerprint directly; this will return an unfolded fingerprint that you can then fold to the desired density. The function Chem.Fingerprints.FingerprintMols.FingerprintMol (written in python) shows how this is done.

The default similarity metric used by DataStructs.FingerprintSimilarity is the Tanimoto similarity. One can use different similarity metrics:

```
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1],
metric=DataStructs.DiceSimilarity)
0.769...
```

Available similarity metrics include Tanimoto, Dice, Cosine, Sokal, Russel, Kulczynski, McConnaughey, and Tversky.

5.2 Other Fingerprints

5.2.1 MACCS keys

There is a SMARTS-based implementation of the 166 public MACCS keys.

```
>>> from Chem import MACCSkeys
>>> fps = [MACCSkeys.GenMACCSKeys(x) for x in ms]
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1])
0.5
>>> DataStructs.FingerprintSimilarity(fps[0],fps[2])
0.538...
>>> DataStructs.FingerprintSimilarity(fps[1],fps[2])
0.214...
```

The MACCS keys were critically evaluated and compared to other MACCS implementations in Q3 2008. In cases where the public keys are fully defined, things looked pretty good.

5.3 Atom Pairs and Topological Torsions

Atom-pair descriptors[3] are available in several different forms. The standard form is as fingerprint including counts for each bit instead of just zeros and ones::

```
>>> from Chem.AtomPairs import Pairs
>>> ms = [Chem.MolFromSmiles('C1CCC1OCC'),
... Chem.MolFromSmiles('CC(C)OCC'),
... Chem.MolFromSmiles('CCOCC')]
>>> pairFps = [Pairs.GetAtomPairFingerprint(x) for x in ms]
```

Because the space of bits that can be included in atom-pair fingerprints is huge, they are stored in a sparse manner. We can get the list of bits and their counts for each fingerprint as a dictionary:

```
>>> d = pairFps[-1].GetNonzeroElements()
>>> d[541732]
1
>>> d[1606690]
2
```

3 Carhart, R.E.; Smith, D.H.; Venkataraghavan R. “Atom Pairs as Molecular Features in Structure-Activity Studies: Definition and Applications” *J. Chem. Inf. Comp. Sci.* **25:**64-73 (1985).

Descriptions of the bits are also available:

```
>>> Pairs.ExplainPairScore(558115)
(('C', 1, 0), 3, ('C', 2, 0))  #<- C with 1 neighbors and 0 pi electrons
                               #3 bonds from a C with 2 neighbors and
                               #0 pi electrons
```

The usual metric for similarity between atom-pair fingerprints is Dice similarity:

```
>>> import DataStructs
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.333...
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.258...
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.560...
```

It's also possible to get atom-pair descriptors encoded as a standard bit vector fingerprint (ignoring the count information):

```
>>> pairFps = [Pairs.GetAtomPairFingerprintAsBitVect(x) for x in ms]
```

Since these are standard bit vectors, the DataStructs module can be used for similarity:

```
>>> import DataStructs
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.479...
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.380...
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.625
```

Topological torsion descriptors[4] are calculated in essentially the same way:

```
>>> from Chem.AtomPairs import Torsions
>>> tts = [Torsions.GetTopologicalTorsionFingerprintAsIntVect(x) for x in ms]
>>> DataStructs.DiceSimilarity(tts[0],tts[1])
0.166...
```

At the time of this writing, topological torsion fingerprints have too many bits to be encodeable using the BitVector machinery, so there is no GetTopologicalTorsionFingerprintAsBitVect function.

6 Descriptor Calcuation

A variety of descriptors are available within the RDKit. The complete list is provided in Section 14.

Most of the descriptors are straightforward to use from Python via the centralized AvailDescriptors module :

```
>>> from Chem import AvailDescriptors
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> AvailDescriptors.descDict['TPSA'](m)
37.299...
>>> AvailDescriptors.descDict['MolLogP'](m)
1.3848
```

Partial charges are handled a bit differently:

```
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> AllChem.ComputeGasteigerCharges(m)
>>> float(m.GetAtomWithIdx(0).GetProp('_GasteigerCharge'))
-0.047...
```

4 Nilakantan, R.; Bauman N.; Dixon J.S.; Venkataraghavan R. “Topological Torsion: A New Molecular Descriptor for SAR Applications. Comparison with Other Desciptors.” *J. Chem.Inf. Comp. Sci.* **27:**82-5 (1987).

7 Chemical Reactions

The RDKit also supports applying chemical reactions to sets of molecules. One way of constructing chemical reactions is to use a SMARTS-based language similar to Daylight's Reaction SMILES:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1](=[O:2])-[OD1].[N:3!H0]>>[C:1]
(=[O:2])[N:3]')
>>> rxn
<Chem.rdChemReactions.ChemicalReaction object at 0x...>
>>> rxn.GetNumProductTemplates()
1
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('CC(=O)O'),
...    Chem.MolFromSmiles('NC')))
>>> len(ps)  # one entry for each possible set of products
1
>>> len(ps[0]) # each entry contains one molecule for each product
1
>>> Chem.MolToSmiles(ps[0][0])
'CNC(=O)C'
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('C(COC(=O)O)C(=O)O'),
...    Chem.MolFromSmiles('NC')))
>>> len(ps)
2
>>> Chem.MolToSmiles(ps[0][0])
'CNC(OCCC(O)=O)=O'
>>> Chem.MolToSmiles(ps[1][0])
'CNC(CCOC(O)=O)=O'
```

Reactions can also be built from MDL rxn files:

```
>>> rxn = AllChem.ReactionFromRxnFile('data/AmideBond.rxn')
>>> rxn.GetNumReactantTemplates()
2
>>> rxn.GetNumProductTemplates()
1
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('CC(=O)O'),
Chem.MolFromSmiles('NC')))
>>> len(ps)
1
>>> Chem.MolToSmiles(ps[0][0])
'CNC(=O)C'
```

It is, of course, possible to do reactions more complex than amide bond formation:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1]=[C:2].[C:3]=[*:4]
[*:5]=[C:6]>>[C:1]1[C:2][C:3][*:4]=[*:5][C:6]1')
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('OC=C'),
Chem.MolFromSmiles('C=CC(N)=C')))
>>> Chem.MolToSmiles(ps[0][0])
'NC1=CCCC(O)C1'
```

Note in this case that there are multiple mappings of the reactants onto the templates, so we have multiple product sets:

```
>>> len(ps)
4
```

You can use canonical smiles and a python dictionary to get the unique products:

```
>>> uniqps = {}
>>> for p in ps:
...     smi = Chem.MolToSmiles(p[0])
...     uniqps[smi] = p[0]
...
>>> uniqps.keys()
['NC1=CCC(O)CC1', 'NC1=CCCC(O)C1']
```

Note that the molecules that are produced by the chemical reaction processing code are not sanitized, as

this artificial reaction demonstrates:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1]=[C:2][C:3]=[C:4].
[C:5]=[C:6]>>[C:1]1=[C:2][C:3]=[C:4][C:5]=[C:6]1')
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('C=CC=C'),
Chem.MolFromSmiles('C=C')))
>>> Chem.MolToSmiles(ps[0][0])
'C1=CC=CC=C1'
>>> p0 = ps[0][0]
>>> Chem.SanitizeMol(p0)
>>> Chem.MolToSmiles(p0)
'c1ccccc1'
```

7.1 Recap Implementation

Associated with the chemical reaction functionality is an implementation of the Recap algorithm.[5] Recap uses a set of chemical transformations mimicking common reactions carried out in the lab in order to decompose a molecule into a series of reasonable fragments.

The RDKit Recap implementation keeps track of the hierarchy of transformations that were applied:

```
>>> import Chem
>>> from Chem import Recap
>>> m = Chem.MolFromSmiles('c1ccccc1OCCOC(=O)CC')
>>> hierarch = Recap.RecapDecompose(m)
>>> type(hierarch)
<class 'Chem.Recap.RecapHierarchyNode'>
```

The hierarchy is rooted at the original molecule:

```
>>> hierarch.smiles
'CCC(=O)OCCOc1ccccc1'
```

and each node tracks its children using a dictionary keyed by SMILES:

```
>>> ks=hierarch.children.keys()
>>> ks.sort()
>>> ks
['[*]C(=O)CC', '[*]CCOC(=O)CC', '[*]CCOc1ccccc1', '[*]OCCOc1ccccc1',
'[*]c1ccccc1']
```

The nodes at the bottom of the hierarchy (the leaf nodes) are easily accessible, also as a dictionary keyed by SMILES:

```
>>> ks=hierarch.GetLeaves().keys()
>>> ks.sort()
>>> ks
['[*]C(=O)CC', '[*]CCO[*]', '[*]CCOc1ccccc1', '[*]c1ccccc1']
```

Notice that dummy atoms are used to mark points where the molecule was fragmented.
The nodes themselves have associated molecules:

```
>>> leaf = hierarch.GetLeaves()[ks[0]]
>>> Chem.MolToSmiles(leaf.mol)
'[*]C(=O)CC'
```

8 Chemical Features and Pharmacophores

Chemical features in the RDKit are defined using a SMARTS-based feature definition language (described in detail in the file `$RDBASE/Docs/Programs/RDPharm3D/FDefFile.htm`). To identify chemical features in molecules, you first must build a feature factory:

5 Lewell, X.Q.; Judd, D.B.; Watson, S.P.; Hann, M.M. "RECAP-Retrosynthetic Combinatorial Analysis Procedure: A Powerful New Technique for Identifying Privileged Molecular Fragments with Useful Applications in Combinatorial Chemistry" *J. Chem. Inf. Comp. Sci.* **38:**511-22 (1998).

```
>>> import Chem
>>> from Chem import ChemicalFeatures
>>> import RDConfig
>>> import os
>>> fdefName = os.path.join(RDConfig.RDDataDir,'BaseFeatures.fdef')
>>> factory = ChemicalFeatures.BuildFeatureFactory(fdefName)
```
and then use the factory to search for features:
```
>>> m = Chem.MolFromSmiles('OCc1ccccc1CN')
>>> feats = factory.GetFeaturesForMol(m)
>>> len(feats)
8
```
The individual features carry information about their family (e.g. donor, acceptor, etc.), type (a more detailed description), and the atom(s) that is/are associated with the feature:
```
>>> feats[0].GetFamily()
'Donor'
>>> feats[0].GetType()
'SingleAtomDonor'
>>> feats[0].GetAtomIds()
(0,)
>>> feats[4].GetFamily()
'Aromatic'
>>> feats[4].GetAtomIds()
(2, 3, 4, 5, 6, 7)
```
If the molecule has coordinates, then the features will also have reasonable locations:
```
>>> from Chem import AllChem
>>> AllChem.Compute2DCoords(m)
0
>>> feats[0].GetPos()
<Geometry.rdGeometry.Point3D object at 0x...>
>>> list(feats[0].GetPos())
[3.75, -1.299..., 0.0]
```

9 Molecular Fragments

The RDKit contains a collection of tools for fragmenting molecules and working with those fragments. Fragments are defined to be made up of a set of connected atoms that may have associated functional groups. This is more easily demonstrated than explained:
```
>>> fName=os.path.join(RDConfig.RDDataDir,'FunctionalGroups.txt')
>>> from Chem import FragmentCatalog
>>> fparams = FragmentCatalog.FragCatParams(1,6,fName)
>>> fparams.GetNumFuncGroups()
39
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> fcgen=FragmentCatalog.FragCatGenerator()
>>> m = Chem.MolFromSmiles('OCC=CC(=O)O')
>>> fcgen.AddFragsFromMol(m,fcat)
3
>>> fcat.GetEntryDescription(0)
'CC<-O>'
>>> fcat.GetEntryDescription(1)
'C<-C(=O)O>=C'
>>> fcat.GetEntryDescription(2)
'C<-C(=O)O>=CC<-O>'
```
Notice that the entry descriptions include pieces in angular brackets (e.g. between '<' and '>'). These describe the functional groups attached to the fragment. For example, in the above example, the catalog

entry 0 corresponds to an ethyl fragment with an alcohol attached to one of the carbons and entry 1 is an ethylene with a carboxylic acid on one carbon. Detailed information about the functional groups can be obtained by asking the fragment for the ids of the functional groups it contains and then looking those ids up in the FragCatParams object:

```
>>> list(fcat.GetEntryFuncGroupIds(2))
[34, 1]
>>> fparams.GetFuncGroup(1)
<Chem.rdchem.Mol object at 0x...>
>>> Chem.MolToSmarts(fparams.GetFuncGroup(1))
'*-C(=O)-,:[O&D1]'
>>> Chem.MolToSmarts(fparams.GetFuncGroup(34))
'*-[O&D1]'
>>> fparams.GetFuncGroup(1).GetProp('_Name')
'-C(=O)O'
>>> fparams.GetFuncGroup(34).GetProp('_Name')
'-O'
```

The catalog is hierarchical: smaller fragments are combined to form larger ones. From a small fragment, one can find the larger fragments to which it contributes using the GetEntryDownIds method:

```
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> m = Chem.MolFromSmiles('OCC(NC1CC1)CCC')
>>> fcgen.AddFragsFromMol(m,fcat)
15
>>> fcat.GetEntryDescription(0)
'CC<-O>'
>>> fcat.GetEntryDescription(1)
'CN<-cPropyl>'
>>> list(fcat.GetEntryDownIds(0))
[3, 4]
>>> fcat.GetEntryDescription(3)
'CCC<-O>'
>>> fcat.GetEntryDescription(4)
'C<-O>CN<-cPropyl>'
```

The fragments from multiple molecules can be added to a catalog:

```
>>> suppl = Chem.SmilesMolSupplier('data/bzr.smi')
>>> ms = [x for x in suppl]
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> for m in ms: nAdded=fcgen.AddFragsFromMol(m,fcat)
>>> fcat.GetNumEntries()
1135
>>> fcat.GetEntryDescription(0)
'cC'
>>> fcat.GetEntryDescription(100)
'cc-nc(C)n'
```

The fragments in a catalog are unique, so adding a molecule a second time doesn't add any new entries:

```
>>> fcgen.AddFragsFromMol(ms[0],fcat)
0  #<- that's the number of new entries added
>>> fcat.GetNumEntries()
1135
```

Once a FragmentCatalog has been generated, it can be used to fingerprint molecules:

```
>>> fpgen = FragmentCatalog.FragFPGenerator()
>>> fp  = fpgen.GetFPForMol(ms[8],fcat)
>>> fp
<DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> fp.GetNumOnBits()
```

```
189
```

The rest of the machinery associated with fingerprints can now be applied to these fragment fingerprints. For example, it's easy to find the fragments that two molecules have in common by taking the intersection of their fingerprints:

```
>>> fp2  = fpgen.GetFPForMol(ms[7],fcat)
>>> andfp = fp&fp2
>>> obl = list(andfp.GetOnBits())
>>> fcat.GetEntryDescription(obl[-1])
'ccc(NC<=O>)cc'
>>> fcat.GetEntryDescription(obl[-5])
'c<-X>ccc(N)cc'
```

or we can find the fragments that distinguish one molecule from another:

```
>>> combinedFp=fp&(fp^fp2) # can be more efficent than fp&(!fp2)
>>> obl = list(combinedFp.GetOnBits())
>>> fcat.GetEntryDescription(obl[-1])
'cccc(N)cc'
```

Or we can use the bit ranking functionality from the ML.InfoTheory module to identify fragments that distinguish actives from inactives:

```
>>> suppl = Chem.SDMolSupplier('data/bzr.sdf')
>>> sdms = [x for x in suppl]
>>> fps = [fpgen.GetFPForMol(x,fcat) for x in sdms]
>>> from ML.InfoTheory import InfoBitRanker
>>> ranker = InfoBitRanker(len(fps[0]),2)
>>> acts = [float(x.GetProp('ACTIVITY')) for x in sdms]
>>> for i,fp in enumerate(fps):
...     act = int(acts[i]>7)
...     ranker.AccumulateVotes(fp,act)
...
>>> top5 = ranker.GetTopN(5)
>>> for id,gain,n0,n1 in top5:
...     print int(id),'%.3f'%gain,int(n0),int(n1)
...
160 0.073 30 43
315 0.073 30 43
1013 0.069 5 53
947 0.066 8 63
17 0.064 35 120  #<- columns are: bitId, infoGain, nInactive, nActive
```

Note that this approach isn't particularly effective for this artificial example.

10 Non-Chemical Functionality

10.1 Bit vectors

Bit vectors are containers for efficiently storing a set number of binary values, e.g. for fingerprints. The RDKit includes two types of fingerprints differing in how they store the values internally; the two types are easily interconverted but are best used for different purpose:

- SparseBitVects store only the list of bits set in the vector; they are well suited for storing very large, very sparsely occupied vectors like pharmacophore fingerprints. Some operations, such as retrieving the list of on bits, are quite fast. Others, such as negating the vector, are very, very slow.
- ExplicitBitVects keep track of both on and off bits. They are generally faster than SparseBitVects, but require more memory to store.

10.2 Discrete value vectors

10.3 3D grids

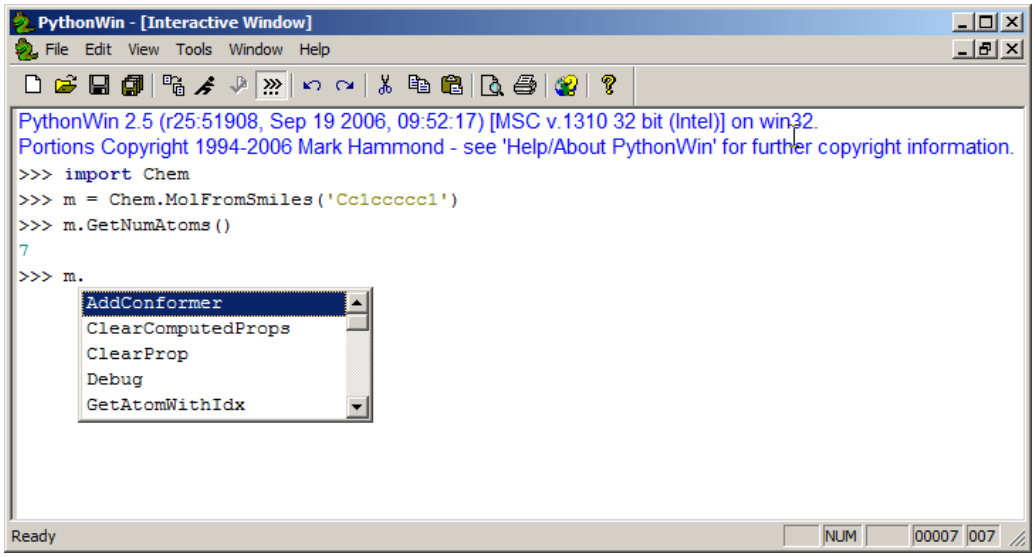
10.4 Points

11 Getting Help

There is a reasonable amount of documentation available within from the RDKit's docstrings. These are accessible using Python's help command:

```
>>> m = Chem.MolFromSmiles('Cc1ccccc1')
>>> m.GetNumAtoms()
7
>>> help(m.GetNumAtoms)
Help on method GetNumAtoms:

GetNumAtoms(...) method of Chem.rdchem.Mol instance
    GetNumAtoms( (Mol)arg1 [, (bool)onlyHeavy=True]) -> int :
        Returns the number of Atoms in the molecule.

          ARGUMENTS:
            - onlyHeavy: (optional) include only heavy atoms (not Hs)
                         defaults to 1.


        C++ signature :
            unsigned int GetNumAtoms(RDKit::ROMol {lvalue} [,bool=True])

>>> m.GetNumAtoms(onlyHeavy=False)
15
```

When working in an environment that does command completion or tooltips, one can see the available methods quite easily. Here's a sample screenshot from within Mark Hammond's PythonWin environment:



12 Advanced Topics/Warnings

12.1 Editing Molecules

Some of the functionality provided allows molecules to be edited “in place”:

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetAtomWithIdx(0).SetAtomicNum(7)
>>> Chem.SanitizeMol(m)
>>> Chem.MolToSmiles(m)
'c1ccncc1'
```

Do not forget the sanitization step, without it one can end up with results that look ok (so long as you don't think):

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetAtomWithIdx(0).SetAtomicNum(8)
>>> Chem.MolToSmiles(m)
'c1ccocc1'
```

but that are, of course, complete nonsense, as sanitization will indicate:

```
>>> Chem.SanitizeMol(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Sanitization error: Can't kekulize mol
```

More complex transformations can be carried out using the EditableMol class:

```
>>> m = Chem.MolFromSmiles('CC(=O)O')
>>> em = Chem.EditableMol(m)
>>> em.ReplaceAtom(3,Chem.Atom(7))
>>> em.AddAtom(Chem.Atom(6))
>>> em.AddAtom(Chem.Atom(6))
>>> em.AddBond(3,4,Chem.BondType.SINGLE)
>>> em.AddBond(4,5,Chem.BondType.DOUBLE)
>>> em.RemoveAtom(0)
```

Note that the EditableMol must be converted back into a standard Mol before much else can be done with it:

```
>>> em.GetNumAtoms()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'EditableMol' object has no attribute 'GetNumAtoms'
>>> Chem.MolToSmiles(em)
```

```
Traceback (most recent call last):
  File "/usr/lib/python2.5/doctest.py", line 1228, in __run
    compileflags, 1) in test.globs
  File "<doctest GettingStartedInPython.odt[319]>", line 1, in <module>
    Chem.MolToSmiles(em)
ArgumentError: Python argument types in
    Chem.rdmolfiles.MolToSmiles(EditableMol)
did not match C++ signature:
    MolToSmiles(RDKit::ROMol {lvalue} mol, bool isomericSmiles=False, bool
kekuleSmiles=False, int rootedAtAtom=-1)
>>> m2 = em.GetMol()
>>> Chem.SanitizeMol(m2)
>>> Chem.MolToSmiles(m2)
'C=CNC=O'
```

It is even easier to generate nonsense using the EditableMol than it is with standard molecules. If you need chemically reasonable results, be certain to sanitize the results.

13 Miscellaneous Tips and Hints

13.1 Chem vs AllChem

The majority of "basic" chemical functionality (e.g. reading/writing molecules, substructure searching, molecular cleanup, etc.) is in the Chem module. More advanced, or less frequently used, functionality is in AllChem. The distinction has been made to speed startup and lower import times; there's no sense in loading the 2D->3D library and force field implementation if one is only interested in reading and writing a couple of molecules. If you find the Chem/AllChem thing annoying or confusing, you can use python's "import ... as ..." syntax to remove the irritation:

```
>>> from Chem import AllChem as Chem
>>> m = Chem.MolFromSmiles('CCC')
```

13.2 The SSSR Problem

As others have ranted about with more energy and eloquence than I intend to, the definition of a molecule's smallest set of smallest rings is not unique. In some high symmetry molecules, a "true" SSSR will give results that are unappealing. For example, the SSSR for cubane only contains 5 rings, even though there are "obviously" 6. This problem can be fixed by implementing a *small* (instead of *smallest*) set of smallest rings algorithm that returns symmetric results. This is the approach that we took with the RDKit.

Because it is sometimes useful to be able to count how many SSSR rings are present in the molecule, there is a GetSSSR function, but this only returns the SSSR count, not the potentially non-unique set of rings.

14 List of Available Descriptors

Descriptor/Descriptor Family	**Notes**
Gasteiger/Marsili Partial Charges	*Tetrahedron* **36**:3219-28 (1980)
BalabanJ	*Chem. Phys. Lett.* **89**:399-404 (1982)
BertzCT	*J. Am. Chem. Soc.* **103**:3599-601 (1981)

Ipc	*J. Chem. Phys.* **67**:4517-33 (1977)
HallKierAlpha	*Rev. Comput. Chem.* **2**:367-422 (1991)
Kappa1 - Kappa3	*Rev. Comput. Chem.* **2**:367-422 (1991)
Chi0, Chi1	*Rev. Comput. Chem.* **2**:367-422 (1991)
Chi0n - Chi4n	*Rev. Comput. Chem.* **2**:367-422 (1991)
Chi0v - Chi4v	*Rev. Comput. Chem.* **2**:367-422 (1991)
MolLogP	Wildman and Crippen *JCICS* **39**:868-73 (1999)
MolMR	Wildman and Crippen *JCICS* **39**:868-73 (1999)
MolWt	
HeavyAtomCount	
HeavyAtomMolWt	
NHOHCount	
NOCount	
NumHAcceptors	
NumHDonors	
NumHeteroatoms	
NumRotatableBonds	
NumValenceElectrons	
RingCount	
TPSA	*J. Med. Chem.* **43**:3714-7, (2000)
LabuteASA	*J. Mol. Graph. Mod.* **18**:464-77 (2000)
PEOE_VSA1 - PEOE_VSA14	MOE-type descriptors using partial charges and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm
SMR_VSA1 - SMR_VSA10	MOE-type descriptors using MR contributions and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm
SlogP_VSA1 - SlogP_VSA12	MOE-type descriptors using LogP contributions and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm
EState_VSA1 - EState_VSA11	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not described in the CCG paper)
VSA_EState1 - VSA_EState10	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not described in the CCG paper)
Topliss fragments	implemented using a set of SMARTS definitions in `$(RDBASE)/Data/FragmentDescriptors.csv`

15 License

Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

The intent of this license is similar to that of the RDKit itself. In simple words: “Do whatever you want with it, but please give us some credit.”