RDKit Documentation

Release 2011.09.1

Greg Landrum

October 08, 2011

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED WITH THE RDKIT IN PYTHON

1.1 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 .rst file) or send them to the mailing list: rdkit-devel@lists.sourceforge.net

1.2 Reading and Writing Molecules

1.2.1 Reading single molecules

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

```
>>> from rdkit import Chem
```

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 `rdkit.Chem.rdchem.Mol` object on success:

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

or None on failure:

```
>>> m = Chem.MolFromMolFile('data/invalid.mol')
>>> m is None
True
```

An attempt is made to provide sensible error messages:

```
>>> m1 = Chem.MolFromSmiles('CO(C)C')
```

displays a message like: `[12:18:01] Explicit valence for atom # 1 O greater than permitted` and

```
>>> m2 = Chem.MolFromSmiles('c1cc1')
```

displays something like: `[12:20:41] Can't kekulize mol`. In each case the value `None` is returned:

```
>>> m1 is None
True
>>> m2 is None
True
```

1.2.2 Reading sets of molecules

Groups of molecules are read using a Supplier (for example, an `rdkit.Chem.rdmolfiles.SDMolSupplier` or a `rdkit.Chem.rdmolfiles.SmilesMolSupplier`):

```
>>> 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
```

A good practice is to test each molecule to see if it was correctly read before working with it:

```
>>> suppl = Chem.SDMolSupplier('data/5ht3ligs.sdf')
>>> for mol in suppl:
...   if mol is None: continue
...   print mol.GetNumAtoms()
...
20
24
24
26
```

1.2.3 Writing molecules

Single molecules can be converted to text using several functions present in the `rdkit.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)
<BLANKLINE>
     RDKit
<BLANKLINE>
  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
<BLANKLINE>
```

To include names in the mol blocks, set the molecule’s “_Name” property:

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

It's usually preferable to have a depiction in the Mol block, this can be generated using functionality in the `rdkit.Chem.AllChem` module (see the Chem vs AllChem section for more information).

You can either include 2D coordinates (i.e. a depiction):

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

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
<BLANKLINE>
  4  4  0  0  0  0  0  0  0  0999 V2000
   -0.7931    0.5732   -0.2708 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3802   -0.9196   -0.2340 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7838   -0.5392    0.6548 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.3894    0.8856    0.6202 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
<BLANKLINE>
```

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)
>>>
```

1.2.4 Writing sets of molecules

Multiple molecules can be written to a file using an `rdkit.Chem.rdmolfiles.SDWriter` object:

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

```
...
>>>
```

Other available Writers include the `rdkit.Chem.rdmolfiles.SmilesWriter` and the `rdkit.Chem.rdmolfiles.TDTWriter`.

1.3 Working with Molecules

1.3.1 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()
rdkit.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
```

1.3.2 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 in the section The SSSR Problem.

For more efficient queries about a molecule’s ring systems (avoiding repeated calls to Mol.GetAtomWithIdx), use the `rdkit.Chem.rdchem.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
```

1.3.3 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 `rdkit.Chem.rdmolops.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 `rdkit.Chem.rdmolops.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 `rdkit.Chem.rdmolops.Kekulize` function:

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.AROMATIC
>>> Chem.Kekulize(m)
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.DOUBLE
>>> m.GetBondWithIdx(1).GetBondType()
rdkit.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 `rdkit.Chem.rdmolops.SanitizeMol` function:

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

1.3.4 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 `rdkit.Chem.rdDepictor.Compute2DCoords` function:

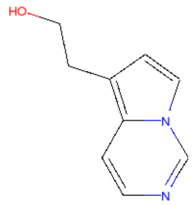
```
>>> m = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(m)
0
```

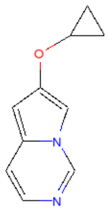
The 2D conformation is constructed in a canonical orientation and is built to minimize intramolecular clashes, i.e. to maximize the clarity of the drawing.

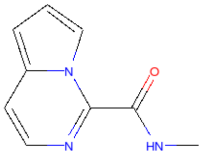
If you have a set of molecules that share a common template and you'd like to align them to that template, you can do so as follows:

```
>>> template = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(template)
0
>>> AllChem.GenerateDepictionMatching2DStructure(m,template)
```

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

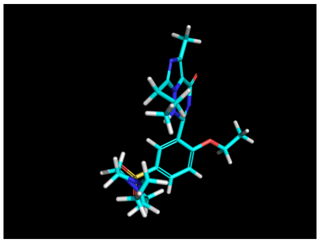


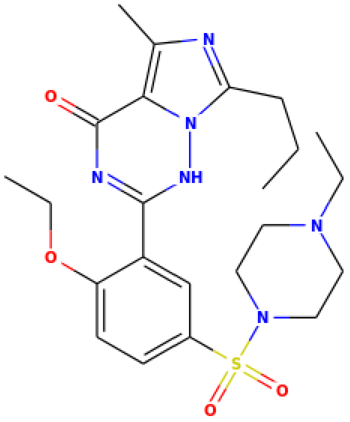




Another option for Compute2DCoords allows you to generate 2D depictions for molecules that closely mimic 3D conformations. This is available using the function `rdkit.Chem.AllChem.GenerateDepictionMatching3DStructure`.

Here is an illustration of the results using the ligand from PDB structure 1XP0:





More fine-grained control can be obtained using the core function `rdkit.Chem.rdDepictor.Compute2DCoordsMimicDistm` 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.

1.3.5 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.
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]

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

[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 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.

1.3.6 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)
123
>>> len(pkl)
475
```

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)
157
```

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).

1.4 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)
```

Those are the atom indices in m, ordered as patt's atoms. To get all of the matches:

```
>>> m.GetSubstructMatches(patt)
((0, 5, 6), (4, 5, 6))
```

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
(<rdkit.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'
```

By default the sidechains are labeled based on the order they are found. They can also be labeled according by the number of that core-atom they're attached to:

```
>>> m1 = Chem.MolFromSmiles('c1c(CCO)ncnc1C(=O)O')
>>> tmp=Chem.ReplaceCore(m1,core,labelByIndex=True)
>>> Chem.MolToSmiles(tmp,True)
'[1*]CCO.[5*]C(=O)O'
```

`rdkit.Chem.rdmolops.ReplaceCore` returns the sidechains in a single molecule. This can be split into separate molecules using `rdkit.Chem.rdmolops.GetMolFrags` :

```
>>> rs = Chem.GetMolFrags(tmp,asMols=True)
>>> len(rs)
2
>>> Chem.MolToSmiles(rs[0],True)
'[1*]CCO'
>>> Chem.MolToSmiles(rs[1],True)
'[5*]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 Reactions functionality.

1.5 Fingerprinting and Molecular Similarity

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

1.5.1 Topological Fingerprints

```
>>> from rdkit import DataStructs
>>> from rdkit.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])
0.666...
>>> DataStructs.FingerprintSimilarity(fps[0],fps[2])
0.444...
>>> DataStructs.FingerprintSimilarity(fps[1],fps[2])
0.25
```

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: 2 - minimum fingerprint size: 64 bits - target on-bit density 0.3

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

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

```
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1], metric=DataStructs.DiceSimilarity)
0.800...
```

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

1.5.2 MACCS Keys

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

```
>>> from rdkit.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.

1.5.3 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 rdkit.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
```

Descriptions of the bits are also available:

```
>>> Pairs.ExplainPairScore(558115)
(('C', 1, 0), 3, ('C', 2, 0))
```

The above means: C with 1 neighbor and 0 pi electrons which is 3 bonds from a C with 2 neighbors and 0 pi electrons

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

```
>>> from rdkit 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 `rdkit.DataStructs` module can be used for similarity:

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

[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).

[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).

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.

1.5.4 Morgan Fingerprints (Circular Fingerprints)

This family of fingerprints, better known as circular fingerprints [5], is built by applying the Morgan algorithm to a set of user-supplied atom invariants. When generating Morgan fingerprints, the radius of the fingerprint must also be provided :

```
>>> from rdkit.Chem import AllChem
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2)
>>> fp1
<rdkit.DataStructs.cDataStructs.UIntSparseIntVect object at 0x...>
>>> m2 = Chem.MolFromSmiles('Cc1ncccc1')
>>> fp2 = AllChem.GetMorganFingerprint(m2,2)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.55...
```

Morgan fingerprints, like atom pairs and topological torsions, use counts by default, but it's also possible to calculate them as bit vectors:

```
>>> fp1 = AllChem.GetMorganFingerprintAsBitVect(m1,2,nBits=1024)
>>> fp1
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> fp2 = AllChem.GetMorganFingerprintAsBitVect(m2,2,nBits=1024)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.51...
```

The default atom invariants use connectivity information similar to those used for the well known ECFP family of fingerprints. Feature-based invariants, similar to those used for the FCFP fingerprints, can also be used. The feature definitions used are defined in the section Feature Definitions Used in the Morgan Fingerprints. At times this can lead to quite different similarity scores:

```
>>> m1 = Chem.MolFromSmiles('c1ccccn1')
>>> m2 = Chem.MolFromSmiles('c1ccco1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2)
>>> fp2 = AllChem.GetMorganFingerprint(m2,2)
>>> ffp1 = AllChem.GetMorganFingerprint(m1,2,useFeatures=True)
>>> ffp2 = AllChem.GetMorganFingerprint(m2,2,useFeatures=True)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.36...
>>> DataStructs.DiceSimilarity(ffp1,ffp2)
0.90...
```

When comparing the ECFP/FCFP fingerprints and the Morgan fingerprints generated by the RDKit, remember that the 4 in ECFP4 corresponds to the diameter of the atom environments considered, while the Morgan fingerprints take a radius parameter. So the examples above, with radius=2, are roughly equivalent to ECFP4 and FCFP4.

The user can also provide their own atom invariants using the optional invariants argument to `rdkit.Chem.rdMolDescriptors.GetMorganFingerprint`. Here's a simple example that uses a constant for the invariant; the resulting fingerprints compare the topology of molecules:

```
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> m2 = Chem.MolFromSmiles('Cc1ncncn1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2,invariants=[1]*m1.GetNumAtoms())
```

[5] Rogers, D.; Hahn, M. "Extended-Connectivity Fingerprints." *J. Chem. Inf. and Model.* **50**:742-54 (2010).

```
>>> fp2 = AllChem.GetMorganFingerprint(m2,2,invariants=[1]*m2.GetNumAtoms())
>>> fp1==fp2
True
```

Note that bond order is by default still considered:

```
>>> m3 = Chem.MolFromSmiles(’CC1CCCCC1’)
>>> fp3 = AllChem.GetMorganFingerprint(m3,2,invariants=[1]*m3.GetNumAtoms())
>>> fp1==fp3
False
```

But this can also be turned off:

```
>>> fp1 = AllChem.GetMorganFingerprint(m1,2,invariants=[1]*m1.GetNumAtoms(),
... useBondTypes=False)
>>> fp3 = AllChem.GetMorganFingerprint(m3,2,invariants=[1]*m3.GetNumAtoms(),
... useBondTypes=False)
>>> fp1==fp3
True
```

Explaining bits from Morgan Fingerprints

Information is available about the atoms that contribute to particular bits in the Morgan fingerprint via the bitInfo argument. The dictionary provided is populated with one entry per bit set in the fingerprint, the keys are the bit ids, the values are lists of (atom index, radius) tuples.

```
>>> m = Chem.MolFromSmiles(’c1cccnc1C’)
>>> info={}
>>> fp = AllChem.GetMorganFingerprint(m,2,bitInfo=info)
>>> len(fp.GetNonzeroElements())
16
>>> len(info)
16
>>> info[98513984]
((1, 1), (2, 1))
>>> info[4048591891]
((5, 2),)
```

Interpreting the above: bit 98513984 is set twice: once by atom 1 and once by atom 2, each at radius 1. Bit 4048591891 is set once by atom 5 at radius 2.

Focusing on bit 4048591891, we can extract the submolecule consisting of all atoms within a radius of 2 of atom 5:

```
>>> env = Chem.FindAtomEnvironmentOfRadiusN(m,2,5)
>>> amap={}
>>> submol=Chem.PathToSubmol(m,env,atomMap=amap)
>>> submol.GetNumAtoms()
6
>>> amap
{0: 3, 1: 5, 3: 4, 4: 0, 5: 1, 6: 2}
```

And then “explain” the bit by generating SMILES for that submolecule:

```
>>> Chem.MolToSmiles(submol)
’ccc(C)nc’
```

This is more useful when the SMILES is rooted at the central atom:

```
>>> Chem.MolToSmiles(submol,rootedAtAtom=amap[5],canonical=False)
'c(nc)(C)cc'
```

1.5.5 Picking Diverse Molecules Using Fingerprints

A common task is to pick a small subset of diverse molecules from a larger set. The RDKit provides a number of approaches for doing this in the `rdkit.SimDivFilters` module. The most efficient of these uses the MaxMin algorithm. [6] Here's an example:

Start by reading in a set of molecules and generating Morgan fingerprints:

```
>>> from rdkit import Chem
>>> from rdkit.Chem.rdMolDescriptors import GetMorganFingerprint
>>> from rdkit import DataStructs
>>> from rdkit.SimDivFilters.rdSimDivPickers import MaxMinPicker
>>> ms = [x for x in Chem.SDMolSupplier('data/actives_5ht3.sdf')]
>>> while ms.count(None): ms.remove(None)
>>> fps = [GetMorganFingerprint(x,3) for x in ms]
>>> nfps = len(fps)
```

The algorithm requires a function to calculate distances between objects, we'll do that using DiceSimilarity:

```
>>> def distij(i,j,fps=fps):
...   return 1-DataStructs.DiceSimilarity(fps[i],fps[j])
```

Now create a picker and grab a set of 10 diverse molecules:

```
>>> picker = MaxMinPicker()
>>> pickIndices = picker.LazyPick(distij,nfps,10,seed=23)
>>> list(pickIndices)
[93, 109, 154, 6, 95, 135, 151, 61, 137, 139]
```

Note that the picker just returns indices of the fingerprints; we can get the molecules themselves as follows:

```
>>> picks = [ms[x] for x in pickIndices]
```

1.6 Descriptor Calculation

A variety of descriptors are available within the RDKit. The complete list is provided in List of Available Descriptors.

Most of the descriptors are straightforward to use from Python via the centralized `rdkit.Chem.Descriptors` module :

```
>>> from rdkit.Chem import Descriptors
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> Descriptors.TPSA(m)
37.299...
>>> Descriptors.MolLogP(m)
1.3848
```

Partial charges are handled a bit differently:

[6] Ashton, M. et al. "Identification of Diverse Database Subsets using Property-Based and Fragment-Based Molecular Descriptions." *Quantitative Structure-Activity Relationships* **21**:598-604 (2002).

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

1.7 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 [7]:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1](=[O:2])-[OD1].[N!H0:3]>>[C:1](=[O:2])[N:3]')
>>> rxn
<rdkit.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
>>> 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
```

[7] A more detailed description of reaction smarts, as defined by the rdkit, is in the *The RDKit Book*.

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
>>> 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'
```

1.7.1 Recap Implementation

Associated with the chemical reaction functionality is an implementation of the Recap algorithm. [8] 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 `rdkit.Chem.Recap` implementation keeps track of the hierarchy of transformations that were applied:

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

[8] 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).

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'
```

1.8 Chemical Features and Pharmacophores

1.8.1 Chemical Features

Chemical features in the RDKit are defined using a SMARTS-based feature definition language (described in detail in the RDKit book). To identify chemical features in molecules, you first must build a feature factory:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import ChemicalFeatures
>>> from rdkit 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 rdkit.Chem import AllChem
>>> AllChem.Compute2DCoords(m)
0
>>> feats[0].GetPos()
<rdkit.Geometry.rdGeometry.Point3D object at 0x...>
>>> list(feats[0].GetPos())
[-2.99..., -1.558..., 0.0]
```

1.8.2 2D Pharmacophore Fingerprints

Combining a set of chemical features with the 2D (topological) distances between them gives a 2D pharmacophore. When the distances are binned, unique integer ids can be assigned to each of these pharmacophores and they can be stored in a fingerprint. Details of the encoding are in the *The RDKit Book*.

Generating pharmacophore fingerprints requires chemical features generated via the usual RDKit feature-typing mechanism:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import ChemicalFeatures
>>> fdefName = 'data/MinimalFeatures.fdef'
>>> featFactory = ChemicalFeatures.BuildFeatureFactory(fdefName)
```

The fingerprints themselves are calculated using a signature (fingerprint) factory, which keeps track of all the parameters required to generate the pharmacophore:

```
>>> from rdkit.Chem.Pharm2D.SigFactory import SigFactory
>>> sigFactory = SigFactory(featFactory,minPointCount=2,maxPointCount=3)
>>> sigFactory.SetBins([(0,2),(2,5),(5,8)])
>>> sigFactory.Init()
>>> sigFactory.GetSigSize()
885
```

The signature factory is now ready to be used to generate fingerprints, a task which is done using the `rdkit.Chem.Pharm2D.Generate` module:

```
>>> from rdkit.Chem.Pharm2D import Generate
>>> mol = Chem.MolFromSmiles('OCC(=O)CCCN')
>>> fp = Generate.Gen2DFingerprint(mol,sigFactory)
>>> fp
<rdkit.DataStructs.cDataStructs.SparseBitVect object at 0x...>
>>> len(fp)
885
>>> fp.GetNumOnBits()
57
```

Details about the bits themselves, including the features that are involved and the binned distance matrix between the features, can be obtained from the signature factory:

```
>>> list(fp.GetOnBits())[:5]
[1, 2, 6, 7, 8]
>>> sigFactory.GetBitDescription(1)
'Acceptor Acceptor |0 1|1 0|'
>>> sigFactory.GetBitDescription(2)
'Acceptor Acceptor |0 2|2 0|'
>>> sigFactory.GetBitDescription(8)
'Acceptor Donor |0 2|2 0|'
>>> list(fp.GetOnBits())[-5:]
[704, 706, 707, 708, 714]
>>> sigFactory.GetBitDescription(707)
'Donor Donor PosIonizable |0 1 2|1 0 1|2 1 0|'
>>> sigFactory.GetBitDescription(714)
'Donor Donor PosIonizable |0 2 2|2 0 0|2 0 0|'
```

For the sake of convenience (to save you from having to edit the fdef file every time) it is possible to disable particular feature types within the SigFactory:

```
>>> sigFactory.skipFeats=['PosIonizable']
>>> sigFactory.Init()
>>> sigFactory.GetSigSize()
510
>>> fp2 = Generate.Gen2DFingerprint(mol,sigFactory)
>>> fp2.GetNumOnBits()
36
```

Another possible set of feature definitions for 2D pharmacophore fingerprints in the RDKit are those published by Gobbi and Poppinger. [9] The module `rdkit.Chem.Pharm2D.Gobbi_Pharm2D` has a pre-configured signature factory for these fingerprint types. Here's an example of using it:

```
>>> from rdkit import Chem
>>> from rdkit.Chem.Pharm2D import Gobbi_Pharm2D,Generate
>>> m = Chem.MolFromSmiles('OCC=CC(=O)O')
>>> fp = Generate.Gen2DFingerprint(m,Gobbi_Pharm2D.factory)
>>> fp
<rdkit.DataStructs.cDataStructs.SparseBitVect object at 0x...>
>>> fp.GetNumOnBits()
8
>>> list(fp.GetOnBits())
[23, 30, 150, 154, 157, 185, 28878, 30184]
>>> Gobbi_Pharm2D.factory.GetBitDescription(157)
'HA HD |0 3|3 0|'
>>> Gobbi_Pharm2D.factory.GetBitDescription(30184)
'HA HD HD |0 3 0|3 0 3|0 3 0|'
```

1.9 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 rdkit.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>'
```

The fragments are stored as entries in a `rdkit.Chem.rdfragcatalog.FragCatalog`. 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

[9] Gobbi, A. & Poppinger, D. "Genetic optimization of combinatorial libraries." *Biotechnology and Bioengineering* **61**:47-54 (1998).

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 `rdkit.Chem.rdfragcatalog.FragCatParams` object:

```
>>> list(fcat.GetEntryFuncGroupIds(2))
[34, 1]
>>> fparams.GetFuncGroup(1)
<rdkit.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 `rdkit.Chem.rdfragcatalog.FragCatalog.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()
1169
>>> 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
>>> fcat.GetNumEntries()
1169
```

Once a `rdkit.Chem.rdfragcatalog.FragCatalog` has been generated, it can be used to fingerprint molecules:

```
>>> fpgen = FragmentCatalog.FragFPGenerator()
>>> fp = fpgen.GetFPForMol(ms[8],fcat)
>>> fp
<rdkit.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 `rdkit.ML.InfoTheory.rdInfoTheory.InfoBitRanker` class 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 rdkit.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)
...
702 0.081 20 17
329 0.073 23 25
160 0.073 30 43
315 0.073 30 43
1034 0.069 5 53
```

The columns above are: bitId, infoGain, nInactive, nActive. Note that this approach isn’t particularly effective for this artificial example.

1.10 Non-Chemical Functionality

1.10.1 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.

1.10.2 Discrete value vectors

1.10.3 3D grids

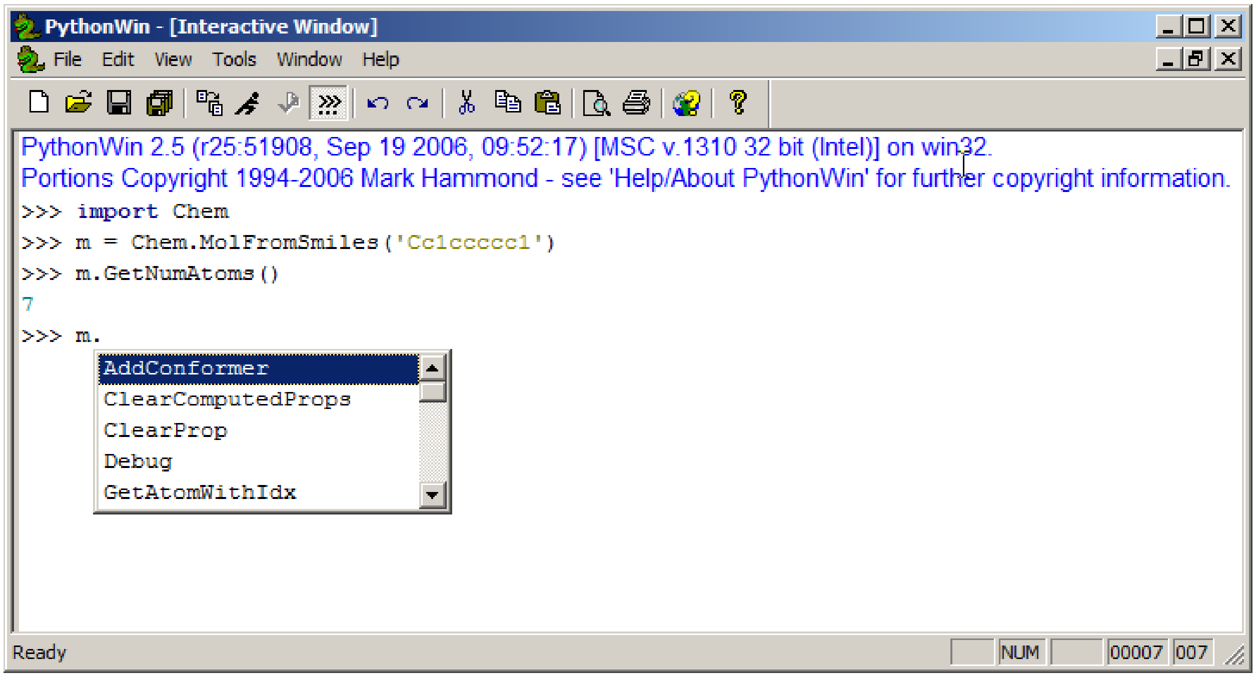
1.10.4 Points

1.11 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:
<BLANKLINE>
GetNumAtoms(...) method of rdkit.Chem.rdchem.Mol instance
    GetNumAtoms( (Mol)arg1 [, (bool)onlyHeavy=True]) -> int :
        Returns the number of Atoms in the molecule.
<BLANKLINE>
          ARGUMENTS:
            - onlyHeavy: (optional) include only heavy atoms (not Hs)
                         defaults to 1.
<BLANKLINE>
<BLANKLINE>
        C++ signature :
            unsigned int GetNumAtoms(RDKit::ROMol {lvalue} [,bool=True])
<BLANKLINE>
>>> 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:



```
PythonWin 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32.
Portions Copyright 1994-2006 Mark Hammond - see 'Help/About PythonWin' for further copyright information.
>>> import Chem
>>> m = Chem.MolFromSmiles('Cc1ccccc1')
>>> m.GetNumAtoms()
7
>>> m.
      AddConformer
      ClearComputedProps
      ClearProp
      Debug
      GetAtomWithIdx
```

1.12 Advanced Topics/Warnings

1.12.1 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 "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[0]>", line 1, in <module>
    Chem.SanitizeMol(m)
ValueError: Sanitization error: Can't kekulize mol
<BLANKLINE>
```

More complex transformations can be carried out using the `rdkit.Chem.rdchem.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 `rdkit.Chem.rdchem.EditableMol` must be converted back into a standard `rdkit.Chem.rdchem.Mol` before much else can be done with it:

```
>>> em.GetNumAtoms()
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[0]>", line 1, in <module>
    em.GetNumAtoms()
AttributeError: 'EditableMol' object has no attribute 'GetNumAtoms'
>>> Chem.MolToSmiles(em)
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[1]>", line 1, in <module>
    Chem.MolToSmiles(em)
ArgumentError: Python argument types in
    rdkit.Chem.rdmolfiles.MolToSmiles(EditableMol)
did not match C++ signature:
    MolToSmiles(RDKit::ROMol {lvalue} mol, bool isomericSmiles=False, bool kekuleSmiles=False, int ro
>>> 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.

1.13 Miscellaneous Tips and Hints

1.13.1 Chem vs AllChem

The majority of "basic" chemical functionality (e.g. reading/writing molecules, substructure searching, molecular cleanup, etc.) is in the `rdkit.Chem` module. More advanced, or less frequently used, functionality is in `rdkit.Chem.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 rdkit.Chem import AllChem as Chem
>>> m = Chem.MolFromSmiles('CCC')
```

1.13.2 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 `rdkit.Chem.rdmolops.GetSSSR` function, but this only returns the SSSR count, not the potentially non-unique set of rings.

1.14 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 contributionshttp://www.chemcomp.
SMR_VSA1 - SMR_VSA10	MOE-type descriptors using MR contributions and surface area contributionshttp://www.chemcom
SlogP_VSA1 - SlogP_VSA12	MOE-type descriptors using LogP contributions and surface area contributionshttp://www.chemc
EState_VSA1 - EState_VSA11	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not c
VSA_EState1 - VSA_EState10	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not c
Topliss fragments	implemented using a set of SMARTS definitions in $(RDBASE)/Data/FragmentDescriptors.csv

1.15 List of Available Fingerprints

Fingerprint Type	Notes
Topological	a Daylight-like fingerprint based on hashing molecular subgraphs
Atom Pairs	*JCICS* **25**:64-73 (1985)
Topological Torsions	*JCICS* **27**:82-5 (1987)
MACCS keys	Using the 166 public keys implemented as SMARTS
Morgan/Circular	Fingerprints based on the Morgan algorithm, similar to the ECFP fingerprint*JCIM* **50**:742-54 (2010).
2D Pharmacophore	Uses topological distances between pharmacophoric points.

1.16 Feature Definitions Used in the Morgan Fingerprints

These are adapted from the definitions in Gobbi, A. & Poppinger, D. "Genetic optimization of combinatorial libraries." *Biotechnology and Bioengineering* **61**, 47-54 (1998).

Feature	SMARTS
Donor	`[$([N;!H0;v3,v4&+1]),$([O,S;H1;+0]),n&H1&+0]`
Acceptor	`[$([O,S;H1;v2;!$(*-*=[O,N,P,S])]),$([O,S;H0;v2]),$([O,S;-]),$([N;v3;!$(N-*=[O,N,P,S`
Aromatic	`[a]`
Halogen	`[F,Cl,Br,I]`
Basic	`[#7;+,$([N;H2&+0][$([C,a]);!$([C,a](=O))]),$([N;H1&+0]([$([C,a]);!$([C,a](=O))])[$(`
Acidic	`[$([C,S](=[O,S,P])-[O;H1,-1])]`

1.17 License

This document is copyright (C) 2007-2011 by Greg Landrum

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative 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."

CHAPTER

TWO

THE RDKIT BOOK

2.1 Misc Cheminformatics Topics

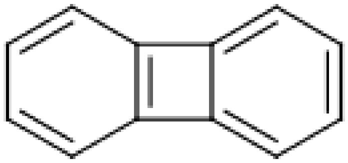
2.1.1 Aromaticity

Aromaticity is one of those unpleasant topics that is simultaneously simple and impossibly complicated. Since neither experimental nor theoretical chemists can agree with each other about a definition, it’s necessary to pick something arbitrary and stick to it. This is the approach taken in the RDKit.

Instead of using patterns to match known aromatic systems, the aromaticity perception code in the RDKit uses a set of rules. The rules are relatively straightforward.

Aromaticity is a property of atoms and bonds in rings. An aromatic bond must be between aromatic atoms, but a bond between aromatic atoms does not need to be aromatic.

For example the fusing bonds here are not considered to be aromatic by the RDKit:



```
>>> from rdkit import Chem
>>> m = Chem.MolFromSmiles('C1=CC2=C(C=C1)C1=CC=CC=C21')
>>> m.GetAtomWithIdx(3).GetIsAromatic()
True
>>> m.GetAtomWithIdx(6).GetIsAromatic()
True
>>> m.GetBondBetweenAtoms(3,6).GetIsAromatic()
False
```

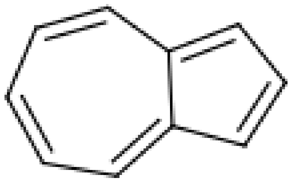
A ring, or fused ring system, is considered to be aromatic if it obeys the 4N+2 rule. Contributions to the electron count are determined by atom type and environment. Some examples:

Fragment	Number of pi electrons
c(a)a	1
n(a)a	1
An(a)a	2
o(a)a	2
s(a)a	2
se(a)a	2
te(a)a	2
O=c(a)a	0
N=c(a)a	0
*(a)a	0, 1, or 2

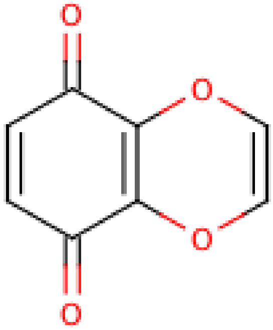
Notation a: any aromatic atom; A: any atom, include H; *: a dummy atom

Notice that exocyclic bonds to electronegative atoms "steal" the valence electron from the ring atom and that dummy atoms contribute whatever count is necessary to make the ring aromatic.

The use of fused rings for aromaticity can lead to situations where individual rings are not aromatic, but the fused system is. An example of this is azulene:



An extreme example, demonstrating both fused rings and the influence of exocyclic double bonds:



```
>>> m=Chem.MolFromSmiles('O=C1C=CC(=O)C2=C1OC=CO2')
>>> m.GetAtomWithIdx(6).GetIsAromatic()
True
>>> m.GetAtomWithIdx(7).GetIsAromatic()
True
>>> m.GetBondBetweenAtoms(6,7).GetIsAromatic()
False
```

2.1.2 Ring Finding and SSSR

[Section taken from “Getting Started” document]

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.

2.2 Chemical Reaction Handling

2.2.1 Reaction SMARTS

Not SMIRKS [1] , not reaction SMILES [2], derived from SMARTS [3].

The general grammar for a reaction SMARTS is :

```
reaction    ::=   reactants “>>” products
reactants   ::=   molecules
products    ::=   molecules
molecules   ::=   molecule
                  molecules ”.”  molecule
molecule    ::=   a valid SMARTS string without ”.”  characters
```

Some features

Mapped dummy atoms in the product template are replaced by the corresponding atom in the reactant:

```
>>> from rdkit.Chem import AllChem
>>> rxn = AllChem.ReactionFromSmarts(’[C:1]=[O,N:2]>>[C:1][*:2]’)
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles(’CC=O’),))[0]]
[’CCO’]
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles(’CC=N’),))[0]]
[’CCN’]
```

but unmapped dummy atoms are left as dummies:

```
>>> rxn = AllChem.ReactionFromSmarts(’[C:1]=[O,N:2]>>[*][C:1][*:2]’)
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles(’CC=O’),))[0]]
[’[*]C(C)O’]
```

“Any” bonds in the products are replaced by the corresponding bond in the reactant:

```
>>> rxn = AllChem.ReactionFromSmarts(’[C:1]~[O,N:2]>>[*][C:1]~[*:2]’)
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles(’C=O’),))[0]]
[’[*]C=O’]
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles(’CO’),))[0]]
```

[1] http://www.daylight.com/dayhtml/doc/theory/theory.smirks.html
[2] http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html
[3] http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html

```
['[*]CO']
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('C#N'),))[0]]
['[*]C#N']
```

Rules and caveats

1. Include atom map information at the end of an atom query. So do [C,N,O:1] or [C;R:1].
2. Don't forget that unspecified bonds in SMARTS are either single or aromatic. Bond orders in product templates are assigned when the product template itself is constructed and it's not always possible to tell if the bond should be single or aromatic:

```
>>> rxn = AllChem.ReactionFromSmarts('[#6:1][#7,#8:2]>>[#6:1][#6:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('C1NCCCC1'),))[0]]
['C1CCCCC1']
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('c1ncccc1'),))[0]]
['c1cccc-c1']
```

So if you want to copy the bond order from the reactant, use an "Any" bond:

```
>>> rxn = AllChem.ReactionFromSmarts('[#6:1][#7,#8:2]>>[#6:1]~[#6:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('c1ncccc1'),))[0]]
['c1ccccc1']
```

2.3 The Feature Definition File Format

An FDef file contains all the information needed to define a set of chemical features. It contains definitions of feature types that are defined from queries built up using Daylight's SMARTS language. [3] The FDef file can optionally also include definitions of atom types that are used to make feature definitions more readable.

2.3.1 Chemical Features

Chemical features are defined by a Feature Type and a Feature Family. The Feature Family is a general classification of the feature (such as "Hydrogen-bond Donor" or "Aromatic") while the Feature Type provides additional, higher-resolution, information about features. Pharmacophore matching is done using Feature Family's. Each feature type contains the following pieces of information:

- A SMARTS pattern that describes atoms (one or more) matching the feature type.
- Weights used to determine the feature's position based on the positions of its defining atoms.

2.3.2 Syntax of the FDef file

AtomType definitions

An AtomType definition allows you to assign a shorthand name to be used in place of a SMARTS string defining an atom query. This allows FDef files to be made much more readable. For example, defining a non-polar carbon atom like this:

```
AtomType Carbon_NonPolar [C&!$(C=[O,N,P,S])&!$(C#N)]
```

creates a new name that can be used anywhere else in the FDef file that it would be useful to use this SMARTS. To reference an AtomType, just include its name in curly brackets. For example, this excerpt from an FDef file defines another atom type - Hphobe - which references the Carbon_NonPolar definition:

```
AtomType Carbon_NonPolar [C&!$(C=[O,N,P,S])&!$(C#N)]
AtomType Hphobe [{Carbon_NonPolar},c,s,S&H0&v2,F,Cl,Br,I]
```

Note that `{Carbon_NonPolar}` is used in the new AtomType definition without any additional decoration (no square brackes or recursive SMARTS markers are required).

Repeating an AtomType results in the two definitions being combined using the SMARTS ”,” (or) operator. Here’s an example:

```
AtomType d1 [N&!H0]
AtomType d1 [O&!H0]
```

This is equivalent to:

```
AtomType d1 [N&!H0,O&!H0]
```

Which is equivalent to the more efficient:

```
AtomType d1 [N,O;!H0]
```

Note that these examples tend to use SMARTS’s high-precendence and operator “&” and not the low-precedence and ”;”. This can be important when AtomTypes are combined or when they are repeated. The SMARTS ”,” operator is higher precedence than ”;”, so definitions that use ”;” can lead to unexpected results.

It is also possible to define negative AtomType queries:

```
AtomType d1 [N,O,S]
AtomType !d1 [H0]
```

The negative query gets combined with the first to produce a definition identical to this:

```
AtomType d1 [!H0;N,O,S]
```

Note that the negative AtomType is added to the beginning of the query.

Feature definitions

A feature definition is more complex than an AtomType definition and stretches across multiple lines:

```
DefineFeature HDonor1 [N,O;!H0]
Family HBondDonor
Weights 1.0
EndFeature
```

The first line of the feature definition includes the feature type and the SMARTS string defining the feature. The next two lines (order not important) define the feature’s family and its atom weights (a comma-delimited list that is the same length as the number of atoms defining the feature). The atom weights are used to calculate the feature’s locations based on a weighted average of the positions of the atom defining the feature. More detail on this is provided below. The final line of a feature definition must be EndFeature. It is perfectly legal to mix AtomType definitions with feature definitions in the FDef file. The one rule is that AtomTypes must be defined before they are referenced.

Additional syntax notes:

- Any line that begins with a # symbol is considered a comment and will be ignored.
- A backslash character, , at the end of a line is a continuation character, it indicates that the data from that line is continued on the next line of the file. Blank space at the beginning of these additional lines is ignored. For example, this AtomType definition:

```
AtomType tButylAtom [$([C;!R](-[CH3])(-[CH3])(-[CH3])),\
$([CH3](-[C;!R](-[CH3])(-[CH3])))]
```

is exactly equivalent to this one:

```
AtomType tButylAtom [$([C;!R](-[CH3])(-[CH3])(-[CH3])),$([CH3](-[C;!R](-[CH3])(-[CH3])))]
```

(though the first form is much easier to read!)

Atom weights and feature locations

2.3.3 Frequently Asked Question(s)

- What happens if a Feature Type is repeated in the file? Here's an example:

  ```
  DefineFeature HDonor1 [O&!H0]
  Family HBondDonor
  Weights 1.0
  EndFeature

  DefineFeature HDonor1 [N&!H0]
  Family HBondDonor
  Weights 1.0
  EndFeature
  ```

  In this case both definitions of the HDonor1 feature type will be active. This is functionally identical to:

  ```
  DefineFeature HDonor1 [O,N;!H0]
  Family HBondDonor
  Weights 1.0
  EndFeature
  ```

  However the formulation of this feature definition with a duplicated feature type is considerably less efficient and more confusing than the simpler combined definition.

2.4 Representation of Pharmacophore Fingerprints

In the RDKit scheme the bit ids in pharmacophore fingerprints are not hashed: each bit corresponds to a particular combination of features and distances. A given bit id can be converted back to the corresponding feature types and distances to allow interpretation. An illustration for 2D pharmacophores is shown in *Figure 1: Bit numbering in pharmacophore fingerprints*.

2.5 License

This document is copyright (C) 2007-2011 by Greg Landrum

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

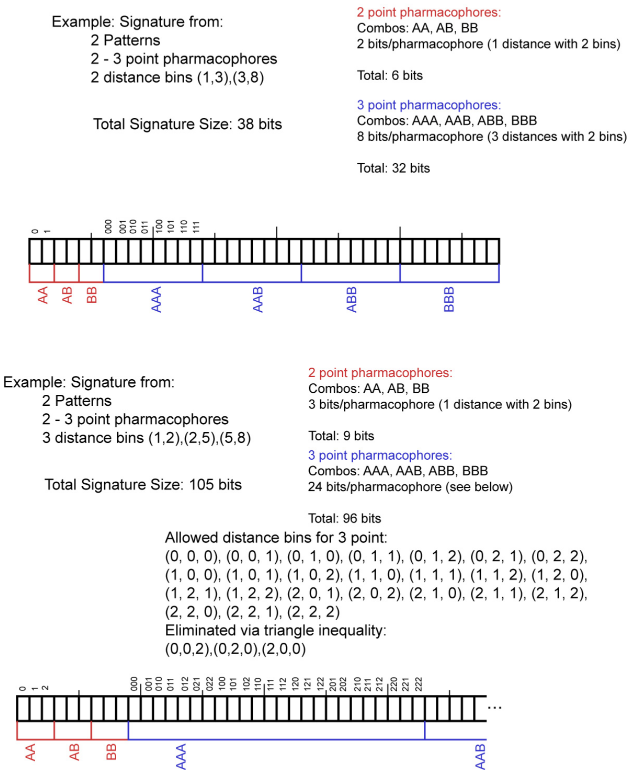


Figure 2.1: Figure 1: Bit numbering in pharmacophore fingerprints

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."