Chemo-informatics and computational drug design

Prof. Dr. Hans De Winter

University of Antwerp

Campus Drie Eiken, Building A

Universiteitsplein 1, 2610 Wilrijk, Belgium

Gefinancierd door de Europese Unie

NextGenerationEU

Chapter 3. Chemical informatics with RDKit

RDKit is an open-source programming API (Application Programming Interface) that allows users to write programs (called scripts) for the manipulation of small molecules. In this section, you will learn how to install RDKit on your computer, and how you can start working with molecules. In later sections, we will apply RDKit to illustrate the concepts of clustering, molecular similarity, and concepts of machine learning.

Since RDKit is open-source, this means that many modelling scientist are using this tool, and many more scientists are also contributing to improve the code behind it. The original link to RDKit is https://www.rdkit.org, and all code can be downloaded from the RDKit github repository: https://github.com/rdkit. You can download the RDKit software from this repository, but there are much easier methods to install the software on your computer. This is described in the following section. In that section, you will also learn how to make use of the Jupyter notebook for easy scripting and testing of our code.

1. Installing RDKit

There are many ways to install RDKit on your computer, but by far the easiest ways to install or use RDKit is 1) to use Anaconda for this purpose, or 2) to use Google Colab.

1.1. Installing with Anaconda

Anaconda is an environment and toolkit that equips you with tools to work with many open source packages and libraries. Anaconda is a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS.

Installation of Anaconda is described in section 2.1. Once you have installed Anaconda on your computer, you can start using it and install RDKit. For this, open a terminal, and enter the following command to make sure that Anaconda has installed correctly:

```
$ conda --version
conda 4.8.5
```

If Anaconda installed correctly the version should be displayed; in this example it is version 4.8.5 but it might be different in your case.

It is not obligatory but advisable to create a separate conda environment in which RDKit will be installed. This environment will be using Python 3. For this example we will give this environment the name ‘rdkit’, but other names are also possible:

```
$ conda create --name rdkit python=3
```

Once created, you should activate this environment:

```
$ conda activate rdkit
```

Now install RDKit:

```
$ conda install -c conda-forge rdkit
```

To test the installation, start a Python session and use RDKit from there:

```
$ python
Python 3.9.1 (default, Dec 11 2020, 06:28:49)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles("Cc1ccccc1")
>>> mol
<rdkit.Chem.rdchem.Mol object at 0x7fd50862da60>
```

```
>>> mol.GetNumAtoms()
7
>>> exit()
```

A useful extension to this conda installation is the installation of a Jupyter notebook. A Jupyter notebook is a Python environment that runs in your web browser and that provides you interactive sessions. It is an ideal environment to play around with the Python and RDKit modules, as it allows to test your code interactively and it also allows you to visualize images along your code. To install, just follow the following easy steps.

First make sure that you have activated your rdkit environment (done in the previous steps), so that the Jupyter notebook will be installed in the same conda environment as RDKit:

```
$ conda activate rdkit
```

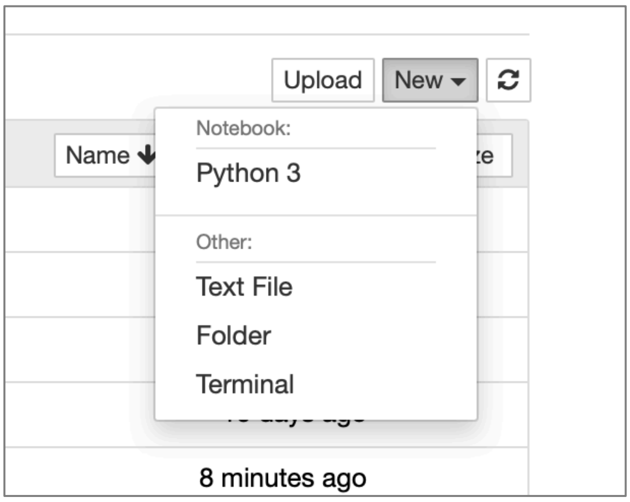
Now install Jupyter:

```
$ conda install jupyter
```

and create your first notebook:

```
$ jupyter notebook
```

This will open your web browser and you can now start a new Python 3 session:

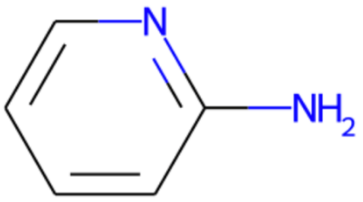


Now you can try your first notebook session. Enter the following:

```
>>> from IPython.display import SVG
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles("c1cccnc1N")
>>> mol
```

Press the Run button and you should see a depiction of the molecule:

```
In [6]:  from IPython.display import SVG
         from rdkit import Chem
         mol = Chem.MolFromSmiles('c1cccnc1N')
         mol
Out[6]:
```

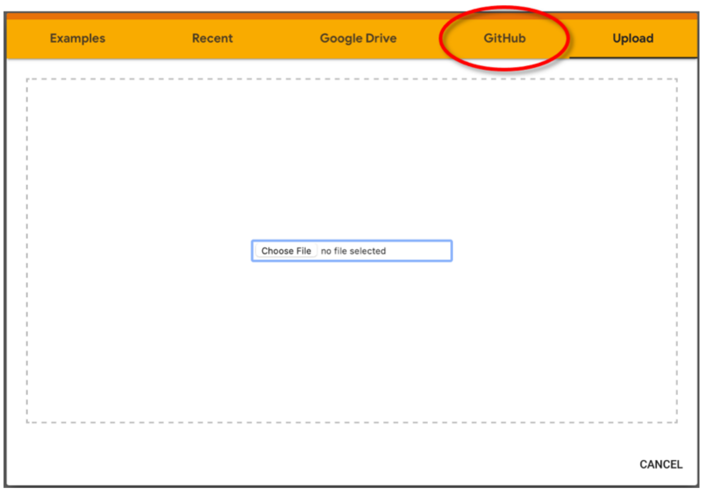


1.2. Installing in Google Colab

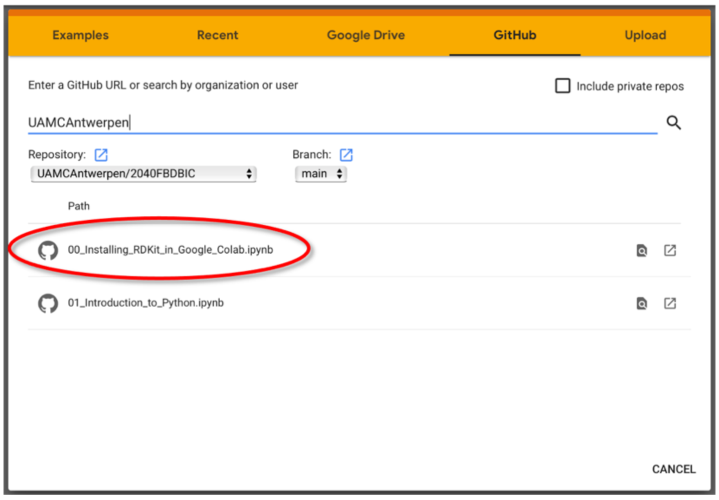
Google Colab can be considered as an online Jupyter notebook, running on the servers of Google and using your Google drive as a harddisk in the cloud. It has the advantage that one does not need to install Anaconda and Jupyter locally, making it easy to work with RDKit if you are not allowed to install software on your local machine. However, the disadvantage of Google Colab is that RDKit is not installed on Google Colab by default. This is something that you will have to take care off each time you will use Google Colab to run RDKit, and installaton of this might take a couple of minutes each time when starting a new workbook. In the following paragraphs we will explain how to do this.

How to start with Google Colab is explained in section 2.2. Please follow these steps and then you are ready to start with the installation of RDKit. This requires some code typing, but as all this code has been saved at the GitHub page of UAMCAntwerpen (the GitHub repository that contains many of the codes for this course), you don't need to type it.

First start a new workbook within Google Colab (`File > New notebook`). This brings you an empy notebook in which you can start typing Python code. However, we need to install RDKit first. For this, select `File > Upload notebook`, and select the GitHub tab from the menu:



Next, enter `UAMCAntwerpen` as the GitHub organization, select `UAMCAntwerpen/2040FBDBIC` as reporsitory (`main` branch), and select `00_Installing_RDKit_in_Google_Colab.ipynb` as the file to download:



Enter a GitHub URL or search by organization or user

Include private repos

UAMCAntwerpen

Repository: UAMCAntwerpen/2040FBDBIC

Branch: main

Path

00_Installing_RDKit_in_Google_Colab.ipynb

01_Introduction_to_Python.ipynb

This will download the RDKit installation code into your newly created notebook,[1] and now you need to press the arrows to run all the code cells and install RDKit (might take some minutes). Test the installation by creating a new code cell and entering:

```
>>> mol = Chem.MolFromSmiles(“C”)
>>> mol
```

This should visualize a depiction of a methane molecule.

2. Your first RDKit lines

After installing RDKit either in an Anaconda environment or on Google Colab, you can now start working with it. RDKit contains a number of modules, of which the `Chem` module being the core module. This one is required for almost all RDKit manipulations:

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

Molecules are stored within RDKit as special `molecule` objects which can be created and modified in a wide variety of ways. An easy way of creating a new molecule is by converting a SMILES representation into a molecule object using the `MolFromSmiles()` method:

```
>>> mol = Chem.MolFromSmiles("CCCC")
>>> print(mol)
<rdkit.Chem.rdchem.Mol object at 0x7fb66742ea60>
```

With RDKit, one can count the number of atoms and bonds very quickly:

```
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 4
>>> nBonds = mol.GetNumBonds()
>>> print("number of bonds:", nBonds)
number of bonds: 3
```

Converting a molecule object back to its SMILES representation is done with the `MolToSmiles()` method:

```
>>> print(Chem.MolToSmiles(mol))
CCCC
```

Molecules can be valid (a correct processed molecule) or invalid (when something went wrong with processing, such as an invalid SMILES:

```
>>> for smiles in ["CCCC", "c"]:
...     print("Smiles:", smiles)
```

[1] In fact, the only code needed to install RDKit in Google Colab is `!pip install rdkit-pypi`

```
...     mol = Chem.MolFromSmiles(smiles)
...     print(mol)
...     print(mol is None)
...     print("")
...
Smiles: CCCC
<rdkit.Chem.rdchem.Mol object at 0x7fb66742eac0>
False

Smiles: c
[11:29:02] non-ring atom 0 marked aromatic
None
True
```

Hydrogen atoms are by default handled as being implicitly there, meaning that hydrogen atoms are not part of the general topology but can be generated upon request. When these hydrogen atoms have been generated, they are called being 'explicit'. The AddHs() and RemoveHs() functions within the AllChem module can be used to add (make explicit) or remove (convert back to implicit) hydrogens:

```
>>> from rdkit.Chem import AllChem
>>> mol = Chem.MolFromSmiles("C")
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 1
>>> print("number of bonds:", mol.GetNumBonds())
number of bonds: 0
>>> mol = AllChem.AddHs(mol)
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 5
>>> print("number of bonds:", mol.GetNumBonds())
number of bonds: 4
>>> mol = AllChem.RemoveHs(mol)
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 1
>>> print("number of bonds:", mol.GetNumBonds())
number of bonds: 0
```

Molecules can contain user-definable properties. Each property is defined by a name and a value. Properties are set with the `SetProperty()` method. One of the predefined properties is the molecular name. This name is defined with the `_Name` property value:

```
>>> mol = Chem.MolFromSmiles("c1ccccc1")
>>> mol.SetProp("_Name", "benzene")
>>> print(Chem.MolToMolBlock(mol))
benzene
     RDKit          2D

  6  6  0  0  0  0  0  0  0  0999 V2000
    1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  2  0
  2  3  1  0
  3  4  2  0
  4  5  1  0
  5  6  2  0
  6  1  1  0
M  END
```

3. Looping over atoms and bonds

Looping over the different atoms in a molecule is done with the `GetAtoms()` method:

```
>>> mol = Chem.MolFromSmiles('C1OC1')
>>> for atom in mol.GetAtoms():
```

```
...     print(atom.GetAtomicNum(), atom.GetIdx(), atom.GetSymbol(), atom.GetExplicitValence())
...
6 0 C 2
8 1 O 2
6 2 C 2
```

One can also access individual atoms by means of their index:

```
>>> for i in range(0, mol.GetNumAtoms()):
...     print(i, mol.GetAtomWithIdx(i).GetSymbol())
...
0 C
1 O
2 C
```

Information about each of the atoms neighbours is retrieved using the `GetNeighbors()` method:

```
>>> for atom in mol.GetAtoms():
...     neighbors = atom.GetNeighbors()
...     print(neighbors)
...     print(atom.GetIdx(), end = ": ")
...     for neighbor in neighbors: print(neighbor.GetIdx(), end="-")
...     print("")
...
(<rdkit.Chem.rdchem.Atom object at 0x7fb667482fa0>, <rdkit.Chem.rdchem.Atom object at
0x7fb6674480a0>)
0: 1-2-
(<rdkit.Chem.rdchem.Atom object at 0x7fb667448220>, <rdkit.Chem.rdchem.Atom object at
0x7fb667448280>)
1: 0-2-
(<rdkit.Chem.rdchem.Atom object at 0x7fb667448040>, <rdkit.Chem.rdchem.Atom object at
0x7fb6674480a0>)
2: 1-0-
```

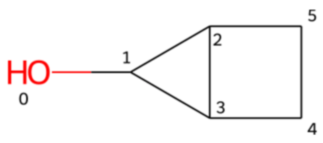
In a similar way, bonds can be looped over with the `GetBonds()` method:

```
>>> for bond in mol.GetBonds():
...     bt = bond.GetBondType()
...     bbi = bond.GetBeginAtomIdx()
...     bei = bond.GetEndAtomIdx()
...     print(bt, bbi, "-", bei)
...
SINGLE 0 - 1
SINGLE 1 - 2
SINGLE 2 - 0
```

4. Rings

RDKit has a wide variety of functions and methods to work with rings. Consider the following molecule:

```
>>> mol = Chem.MolFromSmiles('OC1C2C1CC2')
```



To prompt whether an atom is part of a ring or not, use the `IsInRing()` method:

```
>>> for atom in mol.GetAtoms():
        idx = atom.GetIdx()
        ring = atom.IsInRing()
        r3 = atom.IsInRingSize(3)
        r4 = atom.IsInRingSize(4)
        r6 = atom.IsInRingSize(6)
        print(idx, ring, r3, r4, r6)
```

```
0 False False False False
1 True True False False
2 True True True False
3 True True True False
4 True False True False
5 True False True False
```

The ‘smallest set of smallest rings’ (SSSR) can be obtained with the `GetSymmSSSR()` function within the `Chem` module. In a multi-ring system there are many cyclic paths; for example a naphthalene system has two paths of length six around the two obvious rings plus a path of length ten around the perimeter. Any two of these three rings would completely describe the ring system, but the shortest cyclic paths are what one normally calls the SSSR (Figure 16):

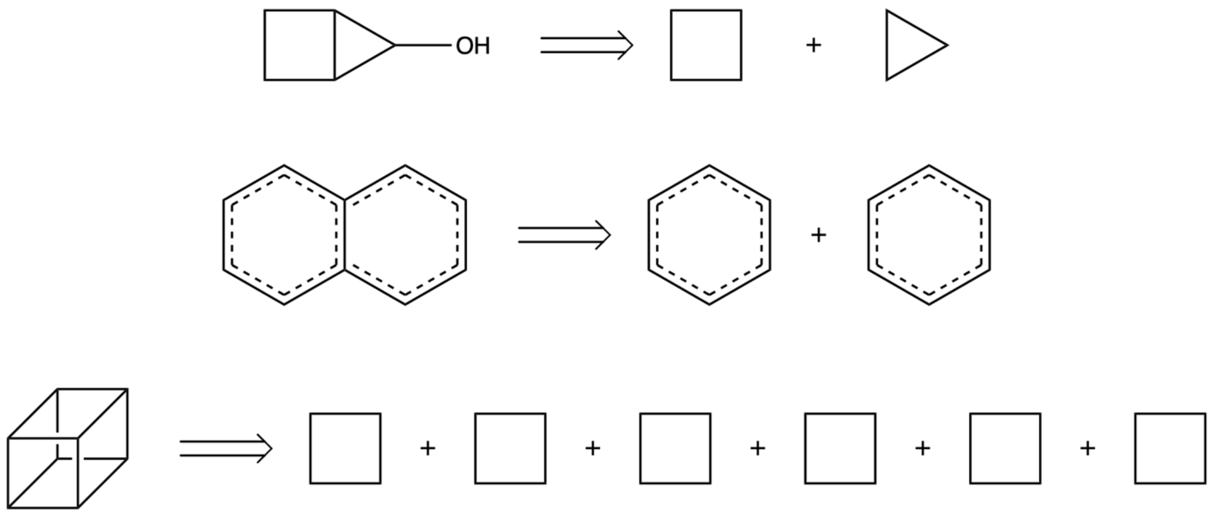


Figure 16. The smallest set of smallest rings (SSSR) depicted for a number of example molecules.

```
>>> mol = Chem.MolFromSmiles("OC1C2C1CC2")    # bicyclo-propane-butane
>>> smallestSetOfSmallestRings = Chem.GetSymmSSSR(mol)
>>> n_sssr = len(smallestSetOfSmallestRings)
>>> print(n_sssr)
2
>>> for i in range(n_sssr): print(list(smallestSetOfSmallestRings[i]))

[1, 2, 3]
[4, 5, 2, 3]
```

```
>>> mol = Chem.MolFromSmiles("c12ccccc1cccc2")    # napthalene
>>> smallestSetOfSmallestRings = Chem.GetSymmSSSR(mol)
>>> n_sssr = len(smallestSetOfSmallestRings)
>>> print(n_sssr)
2
>>> for i in range(n_sssr): print(list(smallestSetOfSmallestRings[i]))

[1, 2, 3, 4, 5, 0]
[6, 7, 8, 9, 0, 5]
```

```
>>> mol = Chem.MolFromSmiles("[C@H]12[C@@H]3[C@@H]4[C@H]1[C@H]5[C@@H]4[C@H]3[C@@H]25")
>>> smallestSetOfSmallestRings = Chem.GetSymmSSSR(mol)
>>> n_sssr = len(smallestSetOfSmallestRings)
>>> print(n_sssr)
6
>>> for i in range(n_sssr): print(list(smallestSetOfSmallestRings[i]))

[0, 3, 2, 1]
[0, 7, 6, 1]
[0, 7, 4, 3]
[1, 6, 5, 2]
[3, 4, 5, 2]
[7, 4, 5, 6]
```

5. Reading and writing molecules

5.1. Single molecules

Molecules can be constructed from a wide variety of molecular formats as decribed in Chapter 2.1. Many examples on how to construct a molecule from a SMILES string with the `Chem.MolFromSmiles()` function have been given in the previous sections, and with the `Chem.MolFromMolFile()` function one can read directly from a SDF file to create a new molecule:

```
>>> mol = Chem.MolFromMolFile(“input.sdf”)
```

It is good practice to check the validity of the generated molecule, since it may happen that the input file (or SMILES string) contains errors which may lead to errors in the molecule construction phase. This can be done by checking whether the generated molecule is `None` or not:

```
>>> mol = Chem.MolFromSmiles("c1ccccc1")
>>> mol is None
False
>>> mol = Chem.MolFromSmiles("c1cCc1")
[12:52:32] Can't kekulize mol.  Unkekulized atoms: 0 1 3

>>> mol is None
True
>>> smiles = ['c1ccccc1', 'c1cCc1']
>>> mols = []
>>> for s in smiles:
...     mol = Chem.MolFromSmiles(s)
...     if mol is None:
...             continue
...     else:
...             mols.append(mol)
...
[12:55:20] Can't kekulize mol.  Unkekulized atoms: 0 1 3

>>> print(len(mols))
1
```

Molecules can also be created from an InChi string using the `MolFromInchi()` method, which is located in the `Chem.inchi` module:

```
>>> from rdkit.Chem import inchi
>>> mol = Chem.MolFromSmiles("C1CCNCC1")
>>> inchistring = inchi.MolToInchi(mol)
>>> print(inchistring)
InChI=1S/C5H11N/c1-2-4-6-5-3-1/h6H,1-5H2
>>> mol = inchi.MolFromInchi(inchistring)
>>> print(Chem.MolToSmiles(mol))
C1CCNCC1
```

MOL blocks (Chapter 2.1.4.) are also possible to write or read:

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

     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
```

Property data can be added with the `SetProp()` method of the molecule. In order to specify the name of the molecule, set the `_Name` property to the actual name

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

This name, or any other property that has been set, can be retrieved with the `GetProp()` method:

```
>>> mol.GetProp("_Name")
'cyclobutane'
```

5.2. Sets of molecules

Multiple molecules can be written to a SDF file (Chapter 2.1.4.) using the `SDWriter()` function:

```
>>> smiles = ["C", "CC", "CO", "CCO", "C1OC1"]
>>> mols = []
>>> for s in smiles: mols.append(Chem.MolFromSmiles(s))
>>> writer = Chem.SDWriter("filename.sdf")
>>> for mol in mols: writer.write(mol)
>>> writer.close()
```

These molecules can be read in again with the corresponding `SDMolSupplier()` function:

```
>>> reader = Chem.SDMolSupplier("filename.sdf")
>>> for mol in reader:
...     if mol is None: continue
...     print(Chem.MolToSmiles(mol))
...
C
CC
CO
CCO
C1CO1
```

To generate a file containing the SMILES representations of molecules, you can use the standard Python `write()` function:

```
>>> f = open("file.smi", "w")
>>> for mol in mols: f.write(Chem.MolToSmiles(mol) + "\n")
...
2
3
3
4
6
>>> f.close()
```

6. Working with conformations

When molecules are created from input like SMILES, there is no information present that describes the position of each atom in three-dimensional space (conformations). RDKit contains a number of functions that allow the user to generate the 3D-conformation(s) of molecules, and one of these is the `EmbedMolecule()` function within the `AllChem` module. Consider this example on aspirine:

```
>>> mol = Chem.MolFromSmiles("CC(=O)Oc1ccccc1C(=O)O")
>>> mol.SetProp("_Name", "aspirine")
>>> print(Chem.MolToMolBlock(mol))
aspirine
     RDKit          2D

 13 13  0  0  0  0  0  0  0  0999 V2000
    5.2500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000   -2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    2.5981    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    3.8971    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  2  0
  2  4  1  0
  4  5  1  0
  5  6  2  0
  6  7  1  0
  7  8  2  0
  8  9  1  0
  9 10  2  0
 10 11  1  0
 11 12  2  0
 11 13  1  0
 10  5  1  0
M  END
```

A number of observations can be made. First, the molecule does not contains hydrogens atoms (these are implicitely present but are not showing up in the connectivities). Second, the z-coordinates of the atoms (third column) are all set to 0, meaning that the molecule contains 2D- rather than 3D-information. To generate a conformation of aspirine, first the hydrogen atoms need to be made explicit:

```
>>> mol = Chem.AddHs(mol)
>>> print(Chem.MolToMolBlock(mol))
aspirine
     RDKit          2D

 21 21  0  0  0  0  0  0  0  0999 V2000
    5.2500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000   -2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    2.5981    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    3.8971    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    6.7500   -1.2990    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    5.2500   -2.7990    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    5.2500    0.2010    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000   -2.5981    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000   -2.5981    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
   -3.0000    0.0000    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    2.5981    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    5.1962    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  2  0
  2  4  1  0
  4  5  1  0
  5  6  2  0
  6  7  1  0
  7  8  2  0
  8  9  1  0
  9 10  2  0
 10 11  1  0
 11 12  2  0
 11 13  1  0
 10  5  1  0
  1 14  1  0
  1 15  1  0
  1 16  1  0
  6 17  1  0
  7 18  1  0
  8 19  1  0
  9 20  1  0
 13 21  1  0
M  END
```

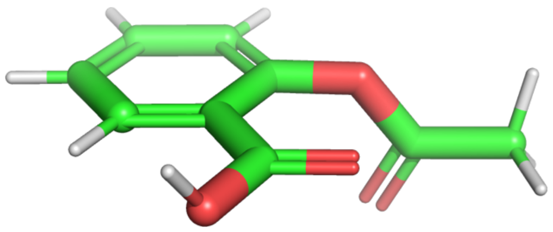
Second, a conformation needs to be generated with the `EmbedMolecule()` function:

```
>>> AllChem.EmbedMolecule(mol)
0
>>> print(Chem.MolToMolBlock(mol))
aspirine
     RDKit          3D

 21 21  0  0  0  0  0  0  0  0999 V2000
   -2.8028   -2.2971    0.2528 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.9196   -1.1193    0.3821 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.9380   -0.4309    1.4053 O   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0329   -0.6957   -0.5799 O   0  0  0  0  0  0  0  0  0  0  0  0
   -0.2038    0.3759   -0.5076 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.5008    1.6592   -0.9114 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.4068    2.6946   -0.8011 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.6647    2.4889   -0.2762 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.9781    1.2093    0.1330 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.0695    0.1765    0.0217 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.4778   -1.1541    0.4786 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.6542   -2.1230    0.3851 O   0  0  0  0  0  0  0  0  0  0  0  0
    2.7241   -1.3647    1.0002 O   0  0  0  0  0  0  0  0  0  0  0  0
   -2.9017   -2.5297   -0.8210 H   0  0  0  0  0  0  0  0  0  0  0  0
   -3.7860   -2.0195    0.6858 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.4405   -3.1762    0.7986 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.4956    1.7914   -1.3196 H   0  0  0  0  0  0  0  0  0  0  0  0
    0.1087    3.6835   -1.1386 H   0  0  0  0  0  0  0  0  0  0  0  0
    2.3837    3.2960   -0.1858 H   0  0  0  0  0  0  0  0  0  0  0  0
    2.9838    1.0330    0.5554 H   0  0  0  0  0  0  0  0  0  0  0  0
    3.5700   -1.4979    0.4428 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  2  0
  2  4  1  0
  4  5  1  0
  5  6  2  0
  6  7  1  0
  7  8  2  0
  8  9  1  0
  9 10  2  0
 10 11  1  0
 11 12  2  0
 11 13  1  0
 10  5  1  0
  1 14  1  0
  1 15  1  0
  1 16  1  0
  6 17  1  0
  7 18  1  0
  8 19  1  0
  9 20  1  0
 13 21  1  0
```

```
M  END
```

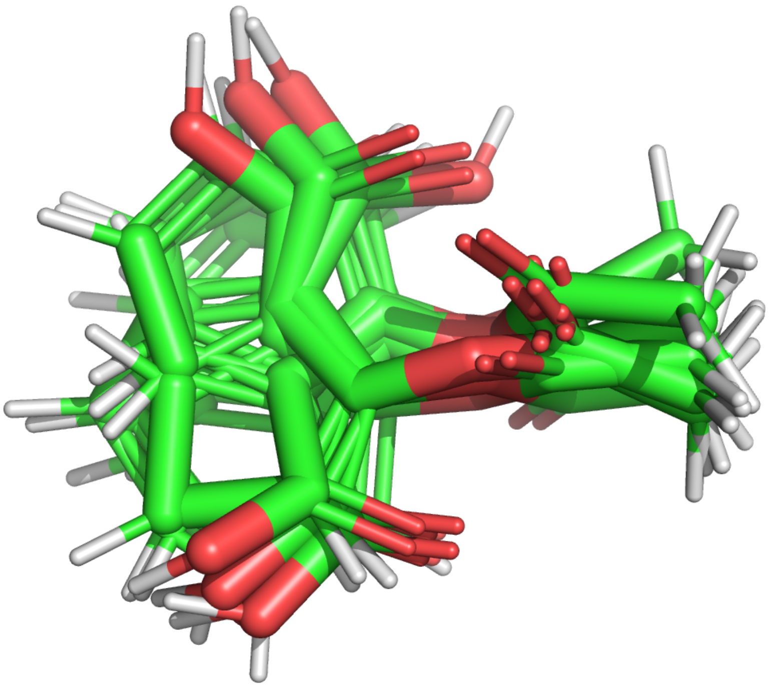
This is better visualised using a molecular graphics visualisation (section 1.2):



Often a single conformation is not enough, as most drug-like molecules contain a significant number of flexible bonds around which the molecule can rotate. Therefore, it is more realistic to generate multiple conformations of the same molecule. This can be achieved with the `EmbedMultipleConfs()` function within `AllChem`:

```
>>> mol = Chem.MolFromSmiles("CC(=O)Oc1ccccc1C(=O)O")
>>> mol = Chem.AddHs(mol)
>>> conformationIds = AllChem.EmbedMultipleConfs(mol, numConfs=10)
>>> print(len(conformationIds))
10
>>> w = Chem.SDWriter("aspirin.sdf")
>>> for cid in conformationIds: w.write(mol, confId = cid)
>>> w.close()
```

This script tries to generate 10 different aspirin conformations and writes these to a file called “`aspirin.sdf`”. The 10 conformations in this file can be visualised with a graphical program like PyMol:

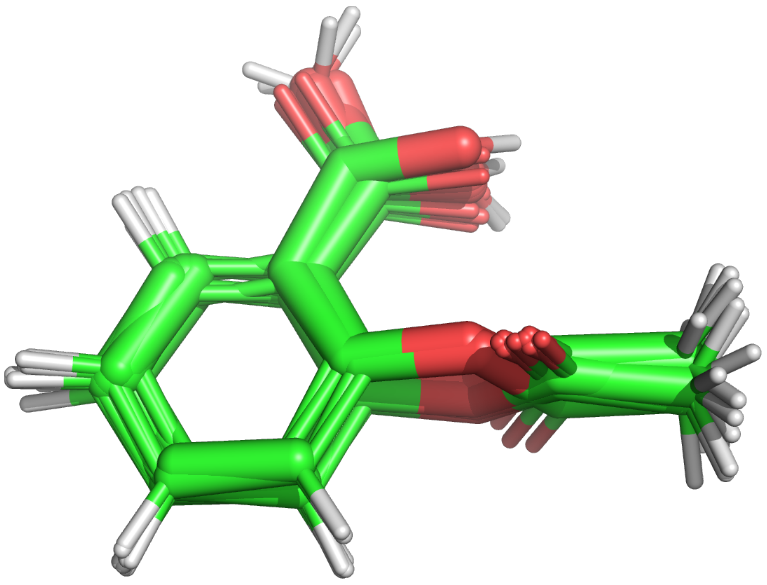


As can be seen, the different conformations are positioned randomly in space, and it would be better to align the structures onto each other for easier visualisation:

```
>>> mol = Chem.MolFromSmiles("CC(=O)Oc1ccccc1C(=O)O")
>>> mol = Chem.AddHs(mol)
>>> conformationIds = AllChem.EmbedMultipleConfs(mol, numConfs=10)
>>> rmslist = []
>>> AllChem.AlignMolConformers(mol, RMSlist = rmslist)
>>> for rms in rmslist: print(rms)

1.017382225703262
1.3937357171422475
1.0824690989074286
1.2752681581629701
```

```
1.0768188563130232
1.4597310140042876
1.3692269740893361
1.4493429134502234
1.5222762797711171
>>> w = Chem.SDWriter("aspirin.sdf")
>>> for cid in conformationIds: w.write(mol, confId = cid)
>>> w.close()
```



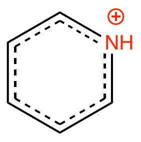
7. Substructure searching: SMARTS

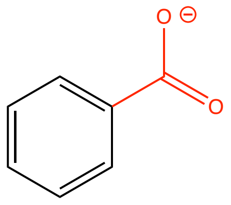
Substructure searching is a very useful application of chemo-informatics. Substructure searches can be used to identify user-defined fragments (substructures) in query molecules. In order to define the substructure that one wants to use, the SMARTS line notation is used (SMARTS stands for **SM**ILES **ar**bitrary **t**arget **s**pecification). SMARTS is related to the SMILES line notation. It was originally developed by David Weininger and colleagues at Daylight Chemical Information Systems. The most comprehensive descriptions of the SMARTS language can be found in Daylight's SMARTS theory manual, tutorial and examples. A useful website to validate your SMARTS is SMARTS-PLUS.

7.1. SMARTS syntax

Atoms

Atoms are specified by their symbol or by their atomic number. An aliphatic carbon is matched by the SMARTS pattern `[C]`, aromatic carbon by `[c]` and any carbon by `[#6]` or `[C,c]` (the comma denotes the `or` operator). The wild card symbols `*`, `A` and `a` match any atom, any aliphatic atom and any aromatic atom respectively (the `*` can also be written as `[A,a]`). Implicit hydrogens are considered to be a characteristic of atoms but these hydrogens can be specified if desired or required. For example, the SMARTS for an amino group can be written as `[NH2]`, but `[N]` can also match an amino group if the nitrogen contains two hydrogens. Atomic formal charge is specified by the descriptors `+` and `-` as exemplified by the SMARTS `[nH+]` of a protonated aromatic nitrogen atom and `[O-]C(=O)c` (deprotonated aromatic carboxylic acid):

`[nH+]` matches 

`[O-]C(=O)c` matches 

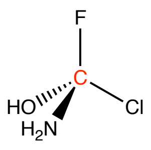
Bonds

Bonds types can be specified with the `-` (single), = (double), # (triple), `:` (aromatic) and ~ (any) tokens. Similar to SMILES, when a bond is not specified, then a single (in the case of aliphatic flanking atoms) or aromatic bond type (in the case of aromatic flanking atoms) is assumed.

Connectivity

The `X` and `D` descriptors are used to specify the total numbers of connections (including implicit hydrogen atoms) and connections to explicit atoms, respectively. As an example, `[CX4]` matches carbon atoms with bonds to any four other atoms while `[CD4]` only matches quaternary carbons:

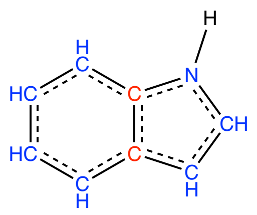
CH_4 is matched by `[CX4]`



is matched by `[CX4]` and `[CD4]`

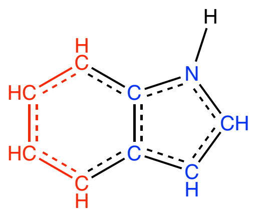
Cyclicity

As originally defined by Daylight, the `R` descriptor is used to specify ring membership. In the Daylight model for cyclic systems, the smallest set of smallest rings (SSSR) is used as a basis for ring membership. For example, indole is perceived as a 5-membered ring fused with a 6-membered ring rather than a 9-membered ring. The two carbon atoms that make up the ring fusion would match `[cR2]` and the other carbon atoms would match `[cR1]`:



match `[cR1]` `[cR2]`

Lower case `r` specifies the size of the smallest ring of which the atom is a member. The carbon atoms of the ring atoms in indole would both match `[cr5]`:



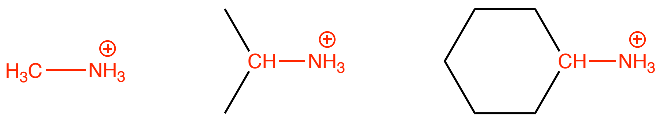
match `[cr5]` `[cr6]`

Bonds can be specified as cyclic, for example `C@C` matches directly bonded atoms in a ring.

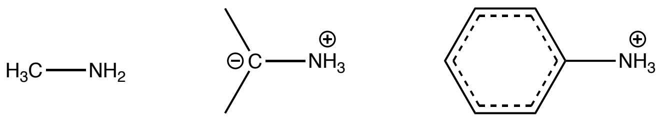
Four logical operators allow atom and bond descriptors to be combined:

Operator	Symbol
and	;
or	,
and (high priority)	&
not	!

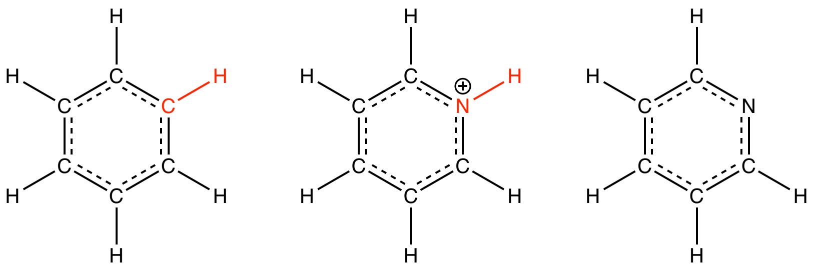
The *and* operator (`;`) can be used to define a protonated primary amine as `[N;H3;+][C;X4]`. This reads as a nitrogen atom (`N`) with (`;`) three hydrogens (`H3`) and (`;`) a positive charge (`+`), bonded with a single bond to an aliphatic carbon atom (`C`) that has (`;`) four implicit or explicit connections (`X4`). The following examples all match this SMARTS pattern:



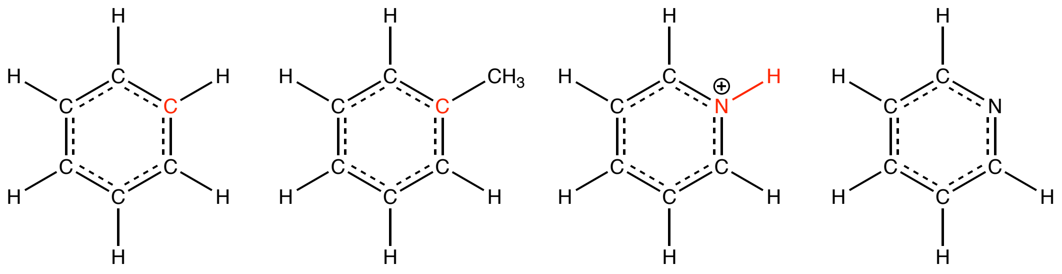
but these examples do not:



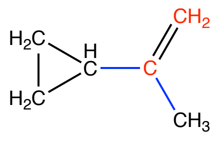
The *or* operator (`,`) has a higher priority than the normal *and* (`;`), so `[c,n;H]` defines an aromatic carbon or an aromatic nitrogen, each possibility bonded to a hydrogen atom:



The *and* operator (&) has higher priority than *or* (`,`) so `[c,n&H]` defines an aromatic carbon, or an aromatic nitrogen with hydrogen atom connected to:



The *not* operator (`!`) can be used to define unsaturated aliphatic carbon as `[C;!X4]` and acyclic single bonds as `*-!@*`:

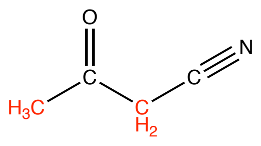


Recursive SMARTS

Recursive SMARTS allow detailed specification of an atom's environment. It is indicated with a dollar sign ($). This SMARTS feature allows you to define an "atomic enviroment" that is required for matching. The "enviroment" atoms will not be included into result match. Any SMARTS expression may be used to define an atomic environment by writing a SMARTS starting with the atom of interest in this form:

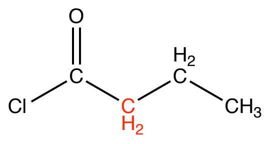
`$(SMARTS)`

For example `[C&!$(C=O)&!$(C#N)]` will match any aliphatic carbon not double bonded to an oxygen and not triple bonded to a nitrogen:



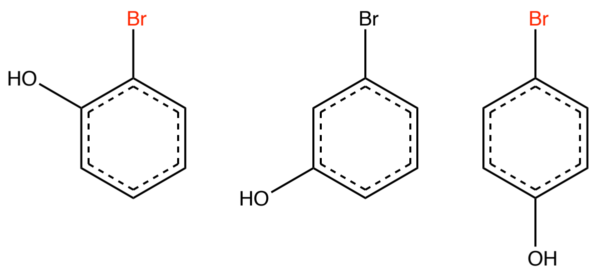
match

while `C[$(C=O)]` will match any aliphatic carbon that is on the α-position of an carbonyl function:



match

As another example, the pattern `Br[$(c1c([OH])cccc1),$(c1ccc([OH])cc1)]` will match any bromo atom that is located on a phenol ring in ortho- or para-position:

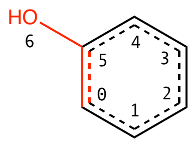


match no match match

7.2. Working with SMARTS

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

```
>>> m = Chem.MolFromSmiles('c1ccccc1O')
>>> smartsMol = Chem.MolFromSmarts('ccO')
>>> m.HasSubstructMatch(smartsMol)
True
>>> m.GetSubstructMatch(smartsMol)
(0, 5, 6)
```



What is returned are the indices of the atoms in the target molecule (phenol), ordered according the atoms in the SMARTS pattern. To get all matches, use the `GetSubstructMatches()` method:

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

To illustrate the bromophenol example above, we can write the following code:

```
>>> bromophenols = ["Oc1ccccc1Br", "Oc1cccc(Br)c1", "Oc1ccc(Br)cc1"]
>>> mols = []
>>> for bp in bromophenols:
...     mols.append(Chem.MolFromSmiles(bp))
...
>>> p = Chem.MolFromSmarts("Br[$(c1c([OH])cccc1),$(c1ccc([OH])cc1)]")
>>> for mol in mols:
...     if mol.HasSubstructMatch(p):
...             print(Chem.MolToSmiles(mol), "True")
...     else:
...             print(Chem.MolToSmiles(mol), "False")
...
Oc1ccccc1Br True
Oc1cccc(Br)c1 False
Oc1ccc(Br)cc1 True
```