API

Overview

This page describes the API of Indigo library and its rendering plugin. The API allows developers to integrate Indigo into their C/Java/C#/Python projects. Please note that Indigo is under active development, and can always post your comments and suggestions to our team.

Basics

System

Indigo acts like a state machine that consists of:

- Objects
- Configuration settings
- Error handling facility

It is possible to use more than one Indigo instance at a time. In plain C API, the "active" instance can be switched with `indigoSetSession` call, while in Python, Java, and C#, the instance is represented as an object of class `Indigo`.

The objects that belong to the Indigo state machine are represented as integer handles in the C API, while in Python, Java, and C# they are wrapped by the `IndigoObject` class.

Access to configuration settings is done via `indigoSetOption***` functions in the C API, while in Python, Java, and C# a number of `Indigo.setOption` methods can accomplish it.

Error handling in C is done via return codes, `indigoGetLastError`, and `indigoSetErrorHandler`. In Python, Java, and C#, as soon as some Indigo function terminates with an error, an `IndigoException` is thrown.

Several library instances may be created to act simultaneously and independently. However, each instance requires a certain amount of memory, and thus it is recommended to have as few instances as possible.

It is allowable to have multiple Indigo instances within one program and even in different threads. However, using a single Indigo instance across multiple threads is prohibited.

From now on, only the Python, Java, and C# interfaces are explained. For those who are interested in plain C interface, please read the C API page.

Indigo Constructor

Java:

```
import com.epam.indigo.*;
...
Indigo indigo = new Indigo();
```

C#:

```
using com.epam.indigo;
....
Indigo indigo = new Indigo();
```

Python:

```
from indigo import *
...
indigo = Indigo()
```

Python will assume the Indigo binaries stored in the `lib` directory in the directory where `indigo.py` is located. In C# and Java, the binaries are unpacked automatically into the system temporary folder.

Getting the Version String

You can use the `Indigo.version` method to get the string containing the Indigo library version number.

Java:

```
System.out.println("Indigo version " + indigo.version());
```

C#:

```
System.Console.WriteLine("Indigo version " + indigo.version());
```

Python:

```
print "Indigo version " + indigo.version()
```

Molecules

Loading Molecules and Query Molecules

The `Indigo` object provides methods for loading molecules and query molecules from: strings, `byte[]` buffers, and files. The input format is detected automatically, except for SMARTS expressions, for which there are special methods.

Java and C#:

```
IndigoObject mol1 = indigo.loadMolecule("ONc1cccc1");
IndigoObject mol2 = indigo.loadMoleculeFromFile("structure.mol");
IndigoObject qmol1 = indigo.loadQueryMolecule("C1-C-C-C-1");
IndigoObject qmol2 = indigo.loadQueryMoleculeFromFile("query.mol");
IndigoObject qmol3 = indigo.loadSmarts("[N,n,O;!H0]");
IndigoObject qmol4 = indigo.loadSmartsFromFile("query.sma");
```

Python: the same with the `IndigoObject` omitted.

Instrumenting Molecules

You can programmatically add atoms and bonds to a molecule. Atoms can be added with the `addAtom` method of the molecule. This method accepts an atom symbol (a string) — an element from the periodic table, or a pseudoatom. Similarly, bonds can be added by calling `addBond` method of an atom. This method accepts another atom and the order of the new bond. You can also create an empty molecule by calling the `createMolecule` method.

Java and C#:

```
IndigoObject mol = indigo.createMolecule();
IndigoObject atom1 = mol.addAtom("C");
IndigoObject atom2 = mol.addAtom("C");
IndigoObject atom3 = mol.addAtom("C");
IndigoObject atom4 = mol.addAtom("C");
IndigoObject atom5 = mol.addAtom("C");
IndigoObject atom6 = mol.addAtom("N");
IndigoObject bond1 = atom1.addBond(atom2, 2);
IndigoObject bond2 = atom2.addBond(atom3, 2);
IndigoObject bond3 = atom3.addBond(atom4, 1);
IndigoObject bond4 = atom4.addBond(atom5, 2);
IndigoObject bond5 = atom5.addBond(atom6, 1);
IndigoObject bond6 = atom6.addBond(atom1, 2);
```

You can programmatically construct a query molecule via `createQueryMolecule` method.

Java and C#:

```
IndigoObject qmol = indigo.createQueryMolecule()
IndigoObject a1 = qmol.addAtom("C")
IndigoObject a2 = qmol.addAtom("[#6]")
a2.addBond(a1, 1)
```

Python: the same with the `IndigoObject` omitted.

You can reset existing atom keeping its connections and stereo configuration using `IndigoObject.resetAtom` method. It accepts string representation of an atom in the SMILES (or SMARTS for the queries) notation:

```
a.resetAtom("N")
```

To add R-sites or convert existing atom into R-site you can use `addRSite` and `setRSite` methods:

```
atom = mol.addRSite("R3")
atom2.setRSite("R4")
```

Instrumenting Query Atoms

Each atom and bond in the query molecule represents as a logic expression of various properties. Indigo support almost all constraints from the SMARTS specification. To alter existing constraints you can use the following methods:

- `addConstraint(type, value)` — adds a specified constraint using logical `and` operation.
- `addConstraintNot(type, value)` — adds a negation of a constraint
- `addConstraintOr(type, value)` — adds a constraint using logical `or` operation.
- `removeConstraints(type)` — removed all constraints with a specified type.

The following self-explaining integer constraint types are supported:

- “atomic-number”
- “charge”
- “isotope”
- “radical”
- “valence”
- “connectivity”
- “total-bond-order”
- “hydrogens”
- “substituents”
- “ring”

- “smallest-ring-size”
- “ring-bonds”
- “rsite-mask”
- “rsite”

Other constraints:

- “aromaticity” = “aliphatic” or “aromatic”
- “smarts” — any single-atom SMARTS expression

Code example:

```
query = indigo.createQueryMolecule()
atom = q.addAtom("")
atom.addConstraint("substituents", "3")
atom.addConstraintNot("atomic-number", "16")
atom.addConstraint("smarts", "[$([#6]=[N+]=[N-]),$([#6-]-[N+]#[N])]")
```

Merging Molecules

You can merge one molecule into another using the `merge` method of a molecule. This method accepts a molecule that is to be merged into the first molecule, and returns a “mapping” object. You can call the `mapAtom` method of the mapping object to know what is the (new) atom of the first molecule that was transferred from the second molecule.

Java and C#:

```
IndigoObject mol = indigo.loadMolecule("c1ccccc1");
IndigoObject mol2 = indigo.loadMolecule("ON");
IndigoObject mapping = mol.merge(mol2);
mapping.mapAtom(mol2.getAtom(0)).addBond(mol.getAtom(3), 1);
```

Python: the same with the `IndigoObject` omitted.

Removing Atoms and Bonds from Molecules

You can call the `IndigoObject.remove` method on an atom or a bond to remove it from the molecule it belongs to. Also, if you want to remove many atoms at once, you can call `IndigoObject.removeAtoms` method, providing to it an array of indices of atoms that you want to remove.

Submolecules

The `IndigoObject.createSubmolecule` method is applicable to a molecule or a query molecule. It accepts an array of atom indices and returns a new molecule containing the given atoms copied from the molecule, and the bonds between them.

Similarly, the `IndigoObject.createEdgeSubmolecule` method accepts two arrays — atom indices and bond indices — and returns a new molecule containing the given atoms and bonds copied from the molecule.

Indigo allows to create a reference on a submolecule of a molecule with method `IndigoObject.getSubmolecule`. Such molecule can be later used for finding layout of a molecule part.

Accessing Atoms and Bonds

The following methods can be applied to a molecule or query molecule:

- `getAtom` — returns the atom by the given index.
- `getBond` — returns the bond by the given index.
- `iterateAtoms` — returns an iterator over atoms, including pseudoatoms and R-sites.
- `iteratePseudoatoms` — returns an iterator over pseudoatoms.
- `iterateRSites` — returns an iterator over R-sites.

- `iterateBonds` — returns an iterator over bonds.

Getting the Properties of Atoms and Bonds

The following methods of a molecule's atom can be called to obtain information:

- `atomicNumber` — returns zero if the atomic number is undefined or ambiguous. (happens only on queries). This method can not be applied to R-sites or pseudoatoms.
- `isotope` — returns the isotope value or zero if the atomic number is undefined or ambiguous.
- `degree` — returns explicit atom degree.
- `charge` — returns the charge value or `null` if the charge is undefined (can happens only on queries).
- `explicitValence` — returns the explicit valence or `null` if there is no explicit valence.
- `radicalElectrons` — returns the number of radical electrons or `null` if the radical is undefined (can happen only on queries).
- `countHydrogens` — returns the total number of hydrogens connected to the atom (explicit+implicit). Can return `null` on query atoms where the number of hydrogens is not definitely known.
- `countImplicitHydrogens` — returns the number of implicit hydrogens connected to the atom. Not applicable to query atoms.
- `valence` — returns the valence of the atom. Not applicable to query atoms.
- `isPseudoatom` — returns `true` if the atom is a pseudoatom, `false` otherwise.
- `isRSite` — returns `true` if the atom is a pseudoatom, `false` otherwise.
- `symbol` — returns a string containing the atom symbol. It is either a symbol the periodic table ("C", "Na"), or a pseudoatom label ("Res"), or an R-site mark ("R1").
- `xyz` —returns an array of three `float` numbers, which define the position of the atom.
- `singleAllowedRGroup` — R-Group index allowed on R-Site (usually there is a single allowed index). This method can be applied exclusively to R-sites.

The following `IndigoObject` methods can be applied to molecule's bonds:

- `bondOrder` — returns 1/2/3 if the bond is a single/double/triple bond. Returns 4 if the bond is an aromatic bond. Returns zero if the bond is ambiguous (query bond).
- `source` — the atom from which the bond is going
- `destination` — the atom to which the bond is going
- `topology` — returns Indigo.RING or Indigo.CHAIN, depending on whether the bond is a ring bond or not. Returns zero if the bond is ambiguous (query bond).

Modifying Atoms and Bonds

The following methods of a molecule's atom can be called to modify the atom:

- `resetCharge`
- `resetExplicitValence`
- `resetIsotope`
- `resetRadical`
- `setCharge` — accepts an integer charge value
- `setIsotope` — accepts an integer isotope value
- `setXYZ` — accepts three float numbers (X, Y, Z)
- `setAttachmentPoint` — accepts an integer index of the attachment point (usually 1 or 2).
- `setBondOrder` — accepts an integer value (1/2/3/4 for single/double/triple/aromatic)

Accessing Neighbor Atoms

With `iterateNeighbors` method you can access the neighbors atoms of an atom. Also, these "neighbor" objects respond to the `bond` method, which returns the bond connecting the atom with the neighbor.

Java:

```
for (IndigoObject atom : mol.iterateAtoms())
{
  System.out.printf("atom %d: %d neighbors\n", atom.index(), atom.degree());
  for (IndigoObject nei : atom.iterateNeighbors())
    System.out.printf("neighbor atom %d is connected by bond %d\n", nei.index(), nei.bond().index());
}
```

C#:

```
foreach (IndigoObject atom in mol.iterateAtoms())
{
  System.Console.WriteLine("atom {0}: {1} neighbors\n", atom.index(), atom.degree());
  foreach (IndigoObject nei in atom.iterateNeighbors())
    System.Console.WriteLine("neighbor atom {0} is connected by bond {1}\n", nei.index(), nei.bond().index());
}
```

Python:

```
for atom in mol.iterateAtoms():
  print "atom %d: %d neighbors" % (atom.index(), atom.degree())
  for nei in atom.iterateNeighbors():
    print "neighbor atom %d is connected by bond %d\n" % (nei.index(), nei.bond().index())
```

Accessing R-Groups

Note: This section applies exclusively to query molecules.

The `iterateRGroups` method iterates over a query molecule's R-groups. Each of the R-groups has a collection of possible "R-group fragments", which in turn can be accessed via the `iterateRGroupFragments` method.

Java:

```
for (IndigoObject rg : mol.iterateRGroups())
{
  System.out.println("RGROUP #" + rg.index());
  for (IndigoObject frag : rg.iterateRGroupFragments())
  {
    System.out.println("  FRAGMENT #" + rg.index());
    System.out.println(frag.molfile());
  }
}
```

C#:

```
foreach (IndigoObject rg in mol.iterateRGroups())
{
  System.Console.WriteLine("RGROUP #" + rg.index());
  foreach (IndigoObject frag in rg.iterateRGroupFragments())
  {
    System.Console.WriteLine("  FRAGMENT #" + rg.index());
    System.Console.WriteLine(frag.molfile());
  }
}
```

Python:

```
for rg in mol.iterateRGroups():
  print "RGROUP #" + rg.index()
  for frag in rg.iterateRGroupFragments():
    print "  FRAGMENT #" + rg.index()
    print frag.molfile()
```

Expand abbreviations

The `expandAbbreviations` function converts input structure using the abbreviations **dictionary** into the expanded form. By default, Indigo uses the following structures list:

Name	Expansion
CO2	[*:2]OC([*:1])=O
Ph	*C1=CC=CC=C1
CO	[*:1]C([*:2])=O
SO2	[*:1]S(=O)(=O)[*:2]
Me	*C
Et	*CC
Boc	CC(C)(C)OC([*])=O
Bz	[*]C(=O)C1=CC=CC=C1
Cbz	[*]C(=O)OCC1=CC=CC=C1
Ac	CC(=O)[*]
NO2	[O-][N+]([*])=O
NO	*N=O
CN	[*]C#N
CHO	[*]C=O
N3	[*]N=[N+]=[N-]
C2H5O	[*]OCC
C6H11	[*]C1CCCCC1
PMB	COC1=CC=C(C[*])C=C1
Bn	[*]CC1=CC=CC=C1
Ms	CS([*])(=O)=O
Cys	SC[C@H](N[*:1])C([*:2])=O
Pr	*CCC
Ts	CC1=CC=C(C=C1)S([*])(=O)=O
t-Bu	CC(C)(C)[*]
Bu	CCCC[*]
Tf	FC(F)(F)S([*])(=O)=O
Tos	CC1=CC=C(C=C1)S([*])(=O)=O
FMOC	[*]C(=O)OCC1C2=CC=CC=C2C2=C1C=CC=C2
SO3-	[O-]S([*])(=O)=O
SO3H	[OH]S([*])(=O)=O
O-	*[O-]
NH3+	*[NH3+]
SiPr	CC(C)S[*]
iPr	CC(C)[*]
CHOH	OC([*:1])[*:2]

Python:

```
mol = indigo.loadMolecule("molfile_with_alias.mol")
mol.expandAbbreviations()
```

Saving Molecules

`IndigoObject.smiles`, when applied to a molecule, returns a SMILES string. Similarly, `IndigoObject.molfile` returns a string with a Molfile, while `IndigoObject.cml` returns a string with CML representation. `IndigoObject.saveMolfile` and `IndigoObject.saveCml` methods save Molfile and CML to disk.

Java:

```
System.out.println(mol1.molfile());
System.out.println(mol2.smiles());
qmol1.saveMolfile("query.mol");
```

C#:

```
System.Console.WriteLine(mol1.molfile());
System.Console.WriteLine(mol2.smiles());
qmol1.saveMolfile("query.mol");
```

Python:

```
print mol1.molfile()
print mol2.smiles()
qmol1.saveMolfile("query.mol")
```

Reactions

Loading Reactions and Query Reactions

Java and C#:

```
IndigoObject rxn1 = indigo.loadReaction("[I-].[Na+].C=CCBr>>[Na+].[Br-].C=CCI");
IndigoObject rxn2 = indigo.loadReactionFromFile("reaction.rxn");
IndigoObject qrxn1 = indigo.loadQueryReaction("CBr>>CCI");
IndigoObject qrxn2 = indigo.loadQueryReactionFromFile("query.rxn");
IndigoObject rs    = indigo.loadReactionSmarts("[C$(CO)]>>[C$(CN)]");
IndigoObject rs2   = indigo.loadReactionSmartsFromFile("query.sma");
```

Python: the same with the `IndigoObject` omitted.

Instrumenting Reactions

The `Indigo.createReaction` method returns an empty reaction. The `Indigo.createQueryReaction` method returns an empty query reaction. The `IndigoObject.addReactant`, `IndigoObject.addProduct`, and `IndigoObject.addCatalyst` methods can then be used to fill it up.

Java and C#:

```
IndigoObject rxn = indigo.createReaction();
rxn.addReactant(mol1);
rxn.addReactant(mol2);
rxn.addProduct(indigo.loadMolecule("ClC1CCCCC1));
```

Python: the same with the `IndigoObject` omitted.

Accessing Reactions

Reactions respond to the following `IndigoObject` methods:

- `IndigoObject.iterateReactants` — enumerates reactants
- `IndigoObject.iterateProducts` — enumerates products
- `IndigoObject.iterateCatalysts` — enumerates catalysts
- `IndigoObject.iterateMolecules` — enumerates reactants, products, and catalysts, in no particular order
- `IndigoObject.countReactants` — returns the number of reactants
- `IndigoObject.countProducts` — returns the number of products
- `IndigoObject.countCatalysts` — returns the number of catalysts

- `IndigoObject.countMolecules` — returns the total number of molecules in the reaction

You can also call the `IndigoObject.remove` method of the reaction molecule to remove it from the reaction.

Java:

```
System.out.println(rxn.countMolecules());
for (IndigoObject item : rxn.iterateReactants())
  System.out.println(item.molfile());

for (IndigoObject item : rxn.iterateCatalysts())
  item.remove();
```

C#:

```
System.Console.WriteLine(rxn.countMolecules());
foreach (IndigoObject item in rxn.iterateReactants())
  System.Console.WriteLine(item.molfile());

foreach (IndigoObject item in rxn.iterateCatalysts())
  item.remove();
```

Python:

```
print rxn.countMolecules()
for item in rxn.iterateReactants():
  print item.molfile()

for item in rxn.iterateCatalysts():
  item.remove();
```

Saving Reactions

The `IndigoObject.smiles`, when applied to a reaction, returns a reaction SMILES string. Similarly, the `IndigoObject.rxnfile` returns a string with an Rxnfile. The `IndigoObject.saveRxnfile` method saves the Rxnfile to disk.

Java:

```
System.out.println(rxn.smiles());
System.out.println(rxn.rxnfile());
rxn.saveRxnfile("reaction.rxn");
```

C#:

```
System.Console.WriteLine(rxn.smiles());
System.Console.WriteLine(rxn.rxnfile());
rxn.saveRxnfile("reaction.rxn");
```

Python:

```
print rxn.smiles()
print rxn.rxnfile()
rxn.saveRxnfile("reaction.rxn")
```

Reacting Centers

Reacting centers include bonds that are involved in the reaction. Indigo supports the following types of reacting centers:

- `Indigo.RC_NOT_CENTER`
- `Indigo.RC_UNMARKED`
- `Indigo.RC_CENTER`

- `Indigo.RC_UNCHANGED`
- `Indigo.RC_MADE_OR_BROKEN`
- `Indigo.RC_ORDER_CHANGED`

These values are bit flags, and can be combined. `IndigoObject.reactingCenter` and `IndigoObject.setReactingCenter` are the getter and setter of the bond reacting center property.

Python:

```
print("reacting centers:")
for m in rxn.iterateMolecules():
    for b in m.iterateBonds():
        print(rxn.reactingCenter(b))
for m in rxn.iterateMolecules():
    for b in m.iterateBonds():
        rxn.setReactingCenter(b, Indigo.RC_CENTER | Indigo.RC_UNCHANGED)
```

The `IndigoObject.correctReactingCenters` method highlights bond reacting centers according to AAM.

Java, C#, and Python:

```
rxn.automap("discard");
rxn.correctReactingCenters();
```

Reaction Atom-to-Atom Mapping

The `IndigoObject.automap(mode [ignore_option])` method is purposed for generating reaction atom-to-atom mapping (AAM). The method accepts a string parameter called `mode`. The following modes are available:

- `discard` : discards the existing mapping entirely and considers only the existing reaction centers (the default)
- `keep` : keeps the existing mapping and maps unmapped atoms
- `alter` : alters the existing mapping, and maps the rest of the reaction but may change the existing mapping
- `clear` : removes the mapping from the reaction

Java, C#, and Python:

```
rxn.automap("discard");
rxn.saveRxnfile("rxn_aam.rxn");
rxn.automap("clear");
rxn.saveSmiles("rxn_noaam.smi");
```

The following options can be added after the `discard`, `keep` or `alter` modes (separated by a space):

- `ignore_charges` : do not consider atom charges while searching
- `ignore_isotopes` : do not consider atom isotopes while searching
- `ignore_valence` : do not consider atom valence while searching
- `ignore_radicals` : do not consider atom radicals while searching

Python:

```
rxn.automap("alter ignore_charges")
for in rxn.iterateMolecules():
    for atom in mol.iterateAtoms():
        print("Atom %d %d" % atom.index(), atom.atomMappingNumber())
rxn.automap("alter ignore_charges ignore_valence")
...
```

The `IndigoObject.clearAAM` method resets current atom-to-atom mapping. Reaction atom has method `IndigoObject.atomMappingNumber` and `IndigoObject.setAtomMappingNumber` to get and set atom-to-atom mapping manually.

Java, C#, and Python:

```
rxn.clearAAM();
rxn.saveSmiles("rxn_noaam.smi");
```

The `aam-timeout` indigo integer parameter (time in milliseconds) corresponds for the AAM algorithm working time. The AAM method returns a current state solution for a reaction when time is over.

Java, C#, and Python:

```
indigo.setOption("aam-timeout", 500);
rxn.automap("discard");
rxn.saveSmiles("rxn_time.smi");
```

Calculating Properties

The following `IndigoObject` methods can be applied to a molecule or query molecule:

- `countAtoms` — returns the number of atoms, including pseudoatoms and R-sites.
- `countPseudoatoms` — returns the number of pseudoatoms.
- `countRSites` — returns the number of R-sites.
- `coundBonds` — returns the number of bonds.
- `grossFormula` — returns a string with the gross formula.
- `molecularWeight` — returns the molecular weight (a floating-point number).
- `mostAbundantMass` — returns the “most abundant isotopes mass” (a floating-point number).
- `monoisotopicMass` — returns the monoisotopic mass (a floating-point number).
- `hasCoord` — returns `true` if the given molecule has coordinates, `false` otherwise.
- `hasZCoord` — returns `true` if the given molecule has 3D coordinates, `false` otherwise.
- `isChiral` — returns `true` if the molecule was loaded from a Molfile, and if it had the ‘Chiral’ flag set.
- `countHeavyAtoms` — returns the number of atoms in the molecule, excluding hydrogen atoms. Hydrogen isotopes are excluded too.
- `countImplicitHydrogens` — returns the total number of implicit hydrogens in the molecule.
- `countHydrogens` — returns the total number of hydrogens in the molecule (implicit hydrogens included, hydrogen isotopes included).
- `countSSSR` — returns the total number of cycles in the Smallest Set of Smallest Rings (SSSR).

Molecule Validation

Molecule validation can be done using the following methods:

- `checkBadValence`
- `checkAmbiguousH`

These functions returns non-empty string description of found issues, or empty string if molecule is correct.

Working with Connected Components

You can use the `countComponents` method to calculate the number of connected components in a structure. Via the `componentIndex` method, you can obtain the number of the component to which the given atom belongs. Also, you can obtain the whole “component” object via the `component` method. You can also iterate over the components using the `iterateComponents` method.

The “component” objects respond to the `countAtoms`, `countBonds`, `iterateAtoms`, and `iterateBonds` calls. However, they can not be used as molecules. If you want to have a separate molecule representing a connected component, you should `clone` it.

Note: The numbering of the components in zero-based.

Java:

```
System.out.printf("%d components\n", mol.countComponents());

for (IndigoObject comp : mol.iterateComponents())
{
   System.out.println(comp.clone().smiles());
   System.out.printf("component %d: %d atoms, %d bonds\n", comp.index(), comp.countAtoms(), comp.countBonds());

   for (IndigoObject atom : comp.iterateAtoms())
      System.out.println(atom.index());

}

for (IndigoObject atom : mol.iterateAtoms())
   System.out.println(atom.componentIndex());

for (IndigoObject atom : mol.component(0).iterateAtoms())
   System.out.println(atom.index());
```

C#:

```
System.Console.WriteLine("{0} components", mol.countComponents());

foreach (IndigoObject comp in mol.iterateComponents())
{
   System.Console.WriteLine(comp.clone().smiles());
   System.Console.WriteLine("component {0}: {0} atoms, {0} bonds\n", comp.index(), comp.countAtoms(), comp.countBon
ds());
   foreach (IndigoObject atom in comp.iterateAtoms())
      System.Console.WriteLine(atom.index());

}

foreach (IndigoObject atom in mol.iterateAtoms())
   System.Console.WriteLine(atom.componentIndex());

foreach (IndigoObject atom in mol.component(0).iterateAtoms())
   System.Console.WriteLine(atom.index());
```

Python:

```
print mol.countComponents(), 'components'

for comp in mol.iterateComponents():
  print comp.clone().smiles()
  print "component %d: %d atoms, %d bonds\n", (comp.index(), comp.countAtoms(), comp.countBonds())
  for atom in comp.iterateAtoms():
    print atom.index()

for atom in mol.iterateAtoms():
  print atom.compomentIndex()

for atom in mol.component(0).iterateAtoms():
  print atom.index()
```

Canonical SMILES

`IndigoObject.canonicalSmiles` method computes the canonical SMILES (also known as absolute SMILES) string for either molecule or reaction.

Java:

```
System.out.println(mol.canonicalSmiles());
System.out.println(rxn.canonicalSmiles());
```

C#:

```
System.Console.WriteLine(mol.canonicalSmiles());
System.Console.WriteLine(rxn.canonicalSmiles());
```

Python:

```
print mol.canonicalSmiles()
print rxn.canonicalSmiles()
```

Please see the Canonical Smiles for detailed examples

Attachment points

Every molecule can have many attachment points. They are grouped by order - the number of connections. For example, a molecule can have 2 attachment points with order 1, and 3 attachment points with order 2. The following methods of `IndigoObject` for a molecule are available for working with attachment points:

- `clearAttachmentPoints` resets all the attachment points.
- `countAttachmentPoints` returns maximal order of attachment points.
- `iterateAttachmentPoints(order)` iterates atoms corresponding to the attachment points with the same specified order.

Python:

```
count = mol.countAttachmentPoints()
print("%s Number of attachment points: %s" % (offset, count))
for order in range(1, count + 1):
    for a in mol.iterateAttachmentPoints(order):
        print("%s   Index: %d. Order %d" % (offset, a.index(), order))
 mol.clearAttachmentPoints()
```

Layout (2D coordinates)

The `IndigoObject.layout` method performs the cleanup of the object it is applied to by computing atoms 2D coordinates.

Java, C#, and Python:

```
mol.layout();
rxn.layout();
```

Aromaticity

The `IndigoObject.aromatize` and `IndigoObject.dearomatize` methods convert molecules/reactions to aromatic and Kekule forms respectively.

Java, C#, and Python:

```
mol1.dearomatize();
rxn.aromatize();
```

Implicit and Explicit Hydrogens

Indigo does not change the representation of the hydrogens automatically. If the hydrogens in the input structure or reaction are implicit (this is usually the case), Indigo does not add the missing hydrogens to the structure or reaction. If some (or all) of hydrogens in the input are explicitly drawn, Indigo does not remove them.

You can force folding (i.e. removal) or unfolding (i.e. addition) the hydrogens of a molecule or reaction by calling

`IndigoObject.foldHydrogens` and `IndigoObject.unfoldHydrogens` methods.

Java, C#, and Python:

```
mol.unfoldHydrogens();
mol.foldHydrogens();
rxn.unfoldHydrogens();
rxn.foldHydrogens();
```

Stereochemistry

The following methods of `IndigoObject` are available for accessing molecule's stereo configuration:

- `countStereocenters` returns the number of the chiral atoms in a molecule
- `iterateStereocenters` returns an iterator for molecule's atoms that are stereocenters
- `countAlleneCenters` returns the number of allene-like stereo fragments
- `iterateAlleneCenters` returns an iterator for molecule's atoms that are centers of allene fragments (the middle 'C' in 'C=C=C')
- `bondStereo` returns one of the following constants:
  - `Indigo.UP` — stereo "up" bond
  - `Indigo.DOWN` — stereo "down" bond
  - `Indigo.EITHER` — stereo "either" bond
  - `Indigo.CIS` — "Cis" double bond
  - `Indigo.TRANS` — "Trans" double bond
  - zero — not a stereo bond of any kind
- `stereocenterType` returns one of the following constants:
  - `Indigo.ABS` — "absolute" stereocenter
  - `Indigo.OR` — "or" stereocenter
  - `Indigo.AND` — "and" stereocenter
  - `Indigo.EITHER` — "any" stereocenter
  - zero — not a stereocenter
- `invertStereo` inverts the stereo configuration of an atom
- `resetStereo` resets the stereo configuration of an atom or a bond
- `changeStereocenterType(newType)` changes current stereocenter type to a specified type
- `addStereocenter(type, idx1, idx2, idx3, [idx4])` adds new stereocenter build on a atom pyramid with a specified atom indices
- `clearStereocenters` resets the chiral configurations of a molecule's atoms
- `clearAlleneCenters` resets the chiral configurations of a molecule's allene-like fragments
- `clearCisTrans` resets the cis-trans configurations of a molecule's bonds

The following methods are useful for keeping cis-trans stereochemistry intact when converting to/from SMILES:

- `resetSymmetricCisTrans` can be called on a molecule loaded from a Molfile or CML. After this call, the cis-trans configurations remain only on nonsymmetric cis-trans bonds. The method returns the number of bonds that have been reset.
- `markEitherCisTrans` can be called prior to saving a molecule loaded from SMILES to Molfile format. It guarantees that the bonds that have no cis-trans configuration in SMILES will not have a cis-trans configuration in the resulting Molfile.

Java:

```
IndigoObject mol = indigo.loadMolecule("chiral.mol");

System.output.println("%d chiral atoms\n", mol.countStereocenters());
for (IndigoObject atom : mol.iterateStereocenters())
{
   System.out.printf("atom %d -- stereocenter type %d\n", atom.index(), atom.stereocenterType());
   atom.invertStereo();
}

for (IndigoObject bond : mol.iterateBonds())
   if (bond.bondStereo() != 0)
      System.out.printf("bond %d -- stereo type %d\n", bond.index(), bond.bondStereo());

System.out.println(mol.smiles());
mol.clearStereocenters();
mol.clearCisTrans();
System.out.println(mol.smiles());
```

C#:

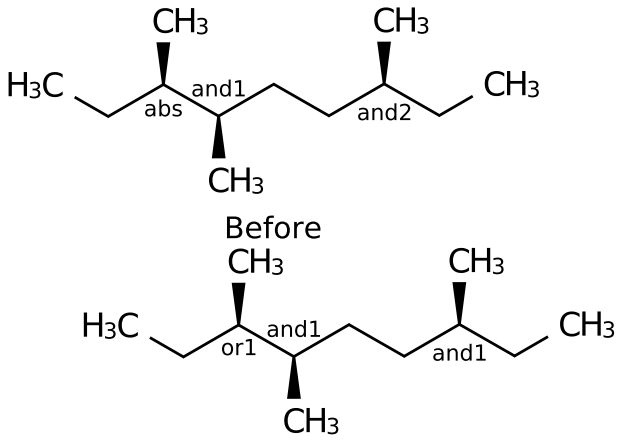
```
IndigoObject mol = indigo.loadMolecule("chiral.mol");

System.Console.WriteLine("{0} chiral atoms\n", mol.countStereocenters());
foreach (IndigoObject atom in mol.iterateStereocenters())
{
   System.Console.WriteLine("atom {0} -- stereocenter type {1}\n", atom.index(), atom.stereocenterType());
   atom.invertStereo();
}

foreach (IndigoObject bond in mol.iterateBonds())
   if (bond.bondStereo() != 0)
      System.Console.WriteLine("bond {0} -- stereo type {1}\n", bond.index(), bond.bondStereo());

System.Console.WriteLine(mol.smiles());
mol.clearStereocenters();
mol.clearCisTrans();
System.out.println(mol.smiles());
```

Python:

```
IndigoObject mol = indigo.loadMolecule("chiral.mol");

print mol.countStereocenters(), "chiral atoms"
for atom in mol.iterateStereocenters():
   print "atom", atom.index(), "-- stereocenter type", atom.stereocenterType()
   atom.invertStereo();

for bond in mol.iterateBonds():
   if bond.bondStereo() != 0:
      print "bond", bond.index(), "-- stereo type", bond.bondStereo()

print mol.smiles()
mol.clearStereocenters()
mol.clearCisTrans()
print mol.smiles()
```

`stereocenterGroup` and `setStereocenterGroup` method to get/set stereocenter group:

```
# Load structure
m = indigo.loadMoleculeFromFile('../release-notes/1.1.x/data/stereogroups.mol')
indigo.setOption('render-comment', 'Before')
indigoRenderer.renderToFile(m, 'result_1.png')

for s in m.iterateStereocenters():
    print "atom index =", s.index(), "group =", s.stereocenterGroup()

m.getAtom(1).changeStereocenterType(Indigo.OR)
m.getAtom(1).setStereocenterGroup(1)
m.getAtom(5).setStereocenterGroup(1)
indigo.setOption('render-comment', 'Stereocenter groups and types were changed')
indigoRenderer.renderToFile(m, 'result_2.png')
```

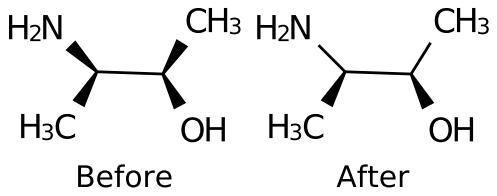


Output:

```
atom index = 1 group = 0
atom index = 2 group = 1
atom index = 5 group = 2
```

The `markStereobonds` method set up/down bond marks if a stereoconfiguration were changed manually, or if it should be reset:

```
m = indigo.loadMoleculeFromFile('../release-notes/1.1.x/data/stereobonds.mol')
indigo.setOption('render-comment', 'Before')
indigoRenderer.renderToFile(m, 'result_1.png')

m.markStereobonds()

indigo.setOption('render-comment', 'After')
indigoRenderer.renderToFile(m, 'result_2.png')
```



Enumeration of Submolecules

Indigo provides methods to enumerate submolecules of different kinds, namely:

- Smallest Set of Smallest Rings (SSSR) — `iterateSSSR`.
- All rings of size within a given interval — `iterateRings`. The method accepts the minimum and maximum amounts of

the rings' atoms.

- All subtrees of size within a given interval — `iterateSubtrees`. The method accepts the minimum and maximum amounts of the subtrees' atoms.
- All edge submolecules of size within a given interval — `iterateEdgeSubmolecules`. The method accepts the minimum and maximum amounts of the submolecules' edges.

The "submolecule" objects returned by these methods respond to the `countAtoms`, `countBonds`, `iterateAtoms`, and `iterateBonds` calls. However, they can not be used as molecules. If you want to have a separate molecule representing the obtained submolecule, you should `clone` it.

Java:

```
for (IndigoObject submol : mol.iterateEdgeSubmolecules(1, mol.countBonds())
{
  System.out.printf("submolecule #%d: %s\n", submol.index(), submol.clone().smiles());
  for (IndigoObject atom : item.iterateAtoms())
    System.out.printf("%d ", atom.index());
  System.out.printf("\n");
  for (IndigoObject bond : item.iterateBonds())
    System.out.printf("%d ", bond.index());
  System.out.printf("\n");
}
```

C#:

```
foreach (IndigoObject item in mol.iterateEdgeSubmolecules(1, mol.countAtoms()))
{
  System.Console.WriteLine("{0} {1}", item.index(), item.clone().smiles());
  foreach (IndigoObject atom in item.iterateAtoms())
    System.Console.Write("{0} ", atom.index());
  System.Console.WriteLine();
  foreach (IndigoObject bond in item.iterateBonds())
    System.Console.Write("{0} ", bond.index());
  System.Console.WriteLine();
}
```

Python:

```
for submol in mol.iterateEdgeSubmolecules(1, mol.countBonds()):
  print "submolecule", submol.index(), ":", submol.clone().smiles()
  print [atom.index() for atom in item.iterateAtoms()]
  print [bond.index() for bond in item.iterateBonds()]
```

Enumeration of Tautomers

Indigo provides a method to enumerate tautomers of a selected molecule. Currently there are two algorithms to enumerate tautomers: based on InChI code and based on a set of reaction SMARTS rules.

The `iterateTautomers` method returns an iterator for tautomers. It accepts a molecule and options as parameters. There are two possible options: `INCHI` to use method based on InChI code, and `RSMARTS` to use reaction SMARTS templates:

Java:

```
for (IndigoObject tautomer : indigo.iterateTautomers(molecule, "RSMARTS")
{
  System.out.printf("tautomer %d: %s\n", tautomer.index(), tautomer.clone().smiles());
}
```

C#:

```
foreach (IndigoObject item in indigo.iterateTautomers(molecule, "RSMARTS"))
{
  System.Console.WriteLine("tautomer {0}: {1}", item.index(), item.clone().smiles());
}
```

Python:

```
for tautomer in indigo.iterateTautomers(molecule, 'RSMARTS'):
  print "tautomer", tautomer.index(), ":", tautomer.clone().smiles()
```

Please see the Enumeration of Tautomers for detailed examples

SGroups

In a molecule loaded from a Molfile, arbitrary subsets of a its atoms and bonds can be joined in S-groups. There are many kinds of S-groups, Indigo supports all described in the format:

- generic SGroup (GEN)
- abbreviation (superatom) (SUP)
- structure repeating unit (SRU)
- multiple SGroup (MUL)
- data SGroup (DAT)
- monomer SGroup (MON)
- mer SGroup (MER)
- copolymer SGroup (COP)
- crosslink SGroup (CRO)
- modification SGroup (MOD)
- graft SGroup (GRA)
- component SGroup (COM)
- mixture SGroup (MIX)
- formulation SGroup (FOR)
- any polymer SGroup (ANY)

The following methods of `IndigoObject` are available for reading molecule's groups:

- `countGenericSGroups`
- `countSuperatoms`
- `countRepeatingUnits`
- `countMultipleGroups`
- `countDataSGroups`
- `iterateGenericSGroups`
- `iterateSuperatoms`
- `iterateRepeatingUnits`
- `iterateMultipleGroups`
- `iterateDataSGroups`
- `getSuperatom(index)`
- `getDataSGroup(index)`

The iterator methods return "group" objects. Each "group" object responds to `IndigoObject.iterateAtoms` and `IndigoObject.iterateBonds` methods.

Java:

```
for (IndigoObject dsg : mol.iterateDataSGroups())
{
   System.out.println("data sgroup " + dsg.index());
   for (IndigoObject atom : dsg.iterateAtoms())
      System.out.println("  atom " + atom.index());
}
```

C#:

```
foreach (IndigoObject dsg in mol.iterateDataSGroups())
{
   System.Console.WriteLine("data sgroup " + dsg.index());
   foreach (IndigoObject atom in dsg.iterateAtoms())
      System.Console.WriteLine("  atom " + atom.index());
}
```

Python:

```
for dsg in mol.iterateDataSGroups():
   print "data sgroup", dsg.index()
   for atom in dsg.iterateAtoms():
      print atom.index()
```

You can also add data SGroups to an existing structure using `IndigoObject.addDataSGroup()` method. It returns the added group. By default, the data SGroup is added in “attached” mode — that is, the data is attached to each of the group’s atoms. To make the data SGroup detached, you can call the `IndigoObject.setDataSGroupXY` method of the data SGroup object. To get a description of a data SGroup use method `IndigoObject.description()`.

To add a superatom use `IndigoObject.addSuperatom()` method:

```
mol.addSuperatom(list_with_atom_indices, "Abbreviation")
```

New set of methods is available for manipulation with S-groups. You can create new S-group using “mapping” object received from matcher:

- `IndigoObject.createSGroup`

Java:

```
IndigoObject match = indigo.substructureMatcher(mol).match(query);

if (match != null)
{
   IndigoObject sgroup = mol.createSGroup("SUP", match, "Asx");
}
```

C#:

```
IndigoObject match = indigo.substructureMatcher(mol).match(query);

if (match != null)
{
   IndigoObject sgroup = mol.createSGroup("SUP", match, "Asx");
}
```

Python:

```
match = indigo.substructureMatcher(mol).match(query)

if match:
   sgroup = mol.createSGroup("SUP", match, "Asx")
```

You can get and set different S-group's properties using the next methods:

- `getSGroupType` - returns S-group type
- `getSGroupIndex` - returns S-group index
- `setSGroupData` - accepts S-group data (for S-groups of "DAT" type)
- `setSGroupCoords` accepts S-group coordinates, x and y values (for S-groups of "DAT" type)
- `setSGroupDescription` - accepts S-group data field units or format (for S-groups of "DAT" type)
- `setSGroupFieldName` - accepts S-group data name (for S-groups of "DAT" type)
- `setSGroupQueryCode` - accepts S-group data query code (for S-groups of "DAT" type)
- `setSGroupQueryOper` - accepts S-group data query operation (for S-groups of "DAT" type)
- `setSGroupDisplay` - accepts S-group data display option (for S-groups of "DAT" type)
- `setSGroupLocation` - accepts S-group data display location (for S-groups of "DAT" type)
- `setSGroupTag` - accepts S-group data tag (for S-groups of "DAT" type)
- `setSGroupTagAlign` - accepts S-group data tag alignment (for S-groups of "DAT" type)
- `setSGroupDataType` - accepts S-group data type (for S-groups of "DAT" type)
- `setSGroupXCoord` - accepts S-group x coordinate (for S-groups of "DAT" type)
- `setSGroupYCoord` - accepts S-group y coordinate (for S-groups of "DAT" type)
- `getSGroupClass` - returns S-group class name (for S-groups of "SUP" type)
- `setSGroupClass` - accepts S-group class name (for S-groups of "SUP" type)
- `getSGroupName` - returns S-group label (for S-groups of "SUP" and "SRU" types)
- `setSGroupName` - accepts S-group label (for S-groups of "SUP" and "SRU" types)
- `getSGroupNumCrossBonds` - returns number of crossing bonds for sgroup
- `addSGroupAttachmentPoint` - accepts attachment point's description and creates it (for S-groups of "SUP" type)
- `deleteSGroupAttachmentPoint` - accepts attachment point's index and removes it (for S-groups of "SUP" type)
- `getSGroupDisplayOption` - returns display option for sgroup (for S-groups of "SUP" type)
- `setSGroupDisplayOption` - accepts display option for sgroup (for S-groups of "SUP" type)
- `getSGroupMultiplier` - returns multiplier value for sgroup (for S-groups of "MUL" type)
- `setSGroupMultiplier` - accepts multiplier value for sgroup (for S-groups of "MUL" type)
- `setSGroupBrackets` - accepts bracket style and brackets coordinates for sgroup (for S-groups of "GEN","MUL" and "SRU" types)

Note: All properties and its values correspond to Molfile format.

Python examples:

```
sg.setSGroupClass("AA")
print(sg.getSGroupName())
print(sg.getSGroupClass())
print(sg.getSGroupNumCrossBonds())
print(sg.getSGroupDisplayOption())
sg.setSGroupName("As")
sg.setSGroupDisplayOption(0)

mp = sg.getSGroupMultiplier()
sg.setSGroupMultiplier(mp + 1)
sg.setSGroupBrackets(1, 1.0, 1.0, 1.0, 2.0, 3.0, 1.0, 3.0, 2.0)

sg.setSGroupData("Test Data S-group")
sg.setSGroupCoords(1.0, 1.0)
sg.setSGroupDescription("SGroup Description (FIELDINFO)")
sg.setSGroupFieldName("SGroup (FIELDNAME)")
sg.setSGroupQueryCode("SGroup (QUERYTYPE)")
sg.setSGroupQueryOper("SGroup (QUERYOP)")
sg.setSGroupDisplay("attached")
sg.setSGroupLocation("relative")
sg.setSGroupTag("G")
sg.setSGroupTagAlign(9)
sg.setSGroupDataType("T")
sg.setSGroupXCoord(4.0)
sg.setSGroupYCoord(5.0)
```

You can find in the `IndigoObject` S-groups using different criteria with corresponding values in S-group properties by type, by name, by class, by atoms or crossing bonds included in S-group:

- `findSGroups` - it accepts key and value for search and retuns iterator for collection of found sgroups

Currently available keys are:

- `SG_TYPE` - find sgroups by type
- `SG_CLASS` - find sgroups by class
- `SG_LABEL` - find sgroups by name
- `SG_DISPLAY_OPTION` - find sgroups by display option
- `SG_BRACKET_STYLE` - find sgroups by bracket style
- `SG_DATA` - find sgroups which contains corresponding string in data field
- `SG_DATA_NAME` - find sgroups by data name
- `SG_DATA_TYPE` - find sgroups by data type
- `SG_DATA_DESCRIPTION` - find sgroups by data description
- `SG_DATA_DISPLAY` - find sgroups by data display option
- `SG_DATA_LOCATION` - find sgroups by data location
- `SG_DATA_TAG` - find sgroups by data tag
- `SG_QUERY_CODE` - find sgroups by query code
- `SG_QUERY_OPER` - find sgroups by query operation
- `SG_PARENT` - find sgroups by parent sgroup
- `SG_CHILD` - find sgroups by children sgroup
- `SG_ATOMS` - find sgroups containing list of atoms
- `SG_BONDS` - find sgroups contating list of crossing bonds

Please see the Sgroups search for detailed examples

S-group can be removed from the `IndigoObject` using `remove` method. In that case just description of corresponding S-group will be removed (the atoms and bonds remain).

TGroups

Indigo supports the hybrid representation (SCSR) for a molecule loaded from a V3000 Molfile. SCSR uses TEMPLATE blocks to represent residues and this representation is widely used for biological sequences.

There are methods for transformation SCSR into full CTAB form and vise versa:

- `transformSCSRtoCTAB` - transforms SCSR into full CTAB representation (templates are transformed into S-groups)
- `transformCTABtoSCSR` - transforms CTAB into SCSR (accepts templates collection and replaces matched fragments by pseudoatoms and corresponding templates)

Examples of usage these methods are in corresponding Examples section.

Alignment of Atoms

The `IndigoObject.alignAtoms` method can be used to calculate and apply the transformation (scale + rotation + translation) of a molecule so that some subset of its atoms will become as close as possible to the desired positions. The method accepts an integer array of atom indices and a float array of desired coordinates (three times bigger that the array of indices, storing resired x,y,z coordinates for each atom).

Java:

```
IndigoObject query = indigo.loadSmarts("[#7]1~[#6]~[#6]~[#7]~[#6]~[#6]2~[#6]~[#6]~[#6]~[#6]~[#6]~1~2");
int[] atoms = new int[query.countAtoms()];
float[] xyz = new float[query.countAtoms() * 3];

for (IndigoObject structure : indigo.iterateSDFile("structures.sdf.gz"))
{
   IndigoObject match = indigo.substructureMatcher(structure).match(query);
   int i = 0;

   if (structure.index() == 0)
      for (IndigoObject atom : query.iterateAtoms())
         System.arraycopy(match.mapAtom(atom).xyz(), 0, xyz, i++ * 3, 3);
   else
   {
      for (IndigoObject atom : query.iterateAtoms())
         atoms[i++] = match.mapAtom(atom).index();

      structure.alignAtoms(atoms, xyz);
   }
}
```

C#:

```
IndigoObject query = indigo.loadSmarts("[#7]1~[#6]~[#6]~[#7]~[#6]~[#6]2~[#6]~[#6]~[#6]~[#6]~[#6]~1~2");
int[] atoms = new int[query.countAtoms()];
float[] xyz = new float[query.countAtoms() * 3];

foreach (IndigoObject structure in indigo.iterateSDFile("structures.sdf.gz"))
{
  IndigoObject match = indigo.substructureMatcher(structure).match(query);
  int i = 0;

  if (structure.index() == 0)
    foreach (IndigoObject atom in query.iterateAtoms())
      Array.Copy(match.mapAtom(atom).xyz(), 0, xyz, i++ * 3, 3);
  else
  {
    foreach (IndigoObject atom in query.iterateAtoms())
    atoms[i++] = match.mapAtom(atom).index();

    structure.alignAtoms(atoms, xyz);
  }
}
```

Python:

```
query = indigo.loadSmarts("[#7]1~[#6]~[#6]~[#7]~[#6]~[#6]2~[#6]~[#6]~[#6]~[#6]~[#6]~1~2");
xyz = []
for structure in indigo.iterateSDFile("structures.sdf.gz"):
  match = indigo.substructureMatcher(structure).match(query)
  if structure.index() == 0:
    for atom in query.iterateAtoms():
      xyz.extend(match.mapAtom(atom).xyz())
  else:
    atoms = []
    for atom in query.iterateAtoms():
      atoms.append(match.mapAtom(atom).index())
    structure.alignAtoms(atoms, xyz);
```

InChI

InChI support is done via IndigoInchi plugin. To work with the InChI plugin in C#/Java/Python wrappers, one needs to create an instance of IndigoInchi, passing an existing Indigo instance to it.

Use `IndigoInchi.loadMolecule` method to convert InChI strings to a molecule, and `IndigoInchi.getInchi` method for the reverse operation.

Python:

```
indigo_inchi = IndigoInchi(indigo);

print(indigo_inchi.version())
m = indigo_inchi.loadMolecule("InChI=1S/C10H20N2O2/c11-7-1-5-2-8(12)10(14)4-6(5)3-9(7)13/h5-10,13-14H,1-4,11-12H2")
print(m.canonicalSmiles())
print(indigo_inchi.getInchi(m))
print(indigo_inchi.getWarning())

m = indigo.loadMolecule("C1CCCCCCC1")
print(indigo_inchi.getInchi(m))
```

OPTIONS

You can pass any options supported by the official InChI library via `inchi-options` option:

```
indigo.setOption("inchi-options", "/DoNotAddH /SUU /SLUUD")
```

One can use both `-` and `/` prefix for them:

```
indigo.setOption("inchi-options", "-DoNotAddH /SUU -SLUUD")
```

Standardize of Molecule

The `IndigoObject.standardize` method can be used to the "standardizing" of the molecule or query (stereo, charges, geometry, valences, atoms and bonds properties) in accordance with requirements. The list of applied modifications is defined by options activated in Indigo (full list of available standardize options is described in the corresponding Options section).

Note: in the case of activation multiple options the order of applied modifications corresponds to the order of the options in the list of available options

Java, C#, and Python:

```
indigo.setOption("standardize-stereo", true);
indigo.setOption("standardize-charges", true);
mol.standardize();
```

Ionize of Molecule

The `IndigoObject.ionize` method can be used for building protonated/deprotonated form of the molecule in accordance with pH and pH tolerance. pKa model for pKa estimation can be defined using corresponding Options section).

Java, C#, and Python:

```
indigo.setOption("pKa-model", "advanced")
indigo.setOption("pKa-model-level", 5)
indigo.setOption("pKa-model-min-level", 2)

mol.ionize(pH, ph_toll)
```

pKa

The `IndigoObject.getAcidPkaValue` and `IndigoObject.getBasicPkaValue` method can be used for estimation pKa values for individual atoms in a molecule. pKa model for pKa estimation can be defined using corresponding Options section).

The `IndigoObject.buildPkaModel` method is used for building pKa model based on custom structures set.

Java, C#, and Python:

```
level = indigo.buildPkaModel(10, 0.5, 'molecules/PkaModel.sdf')

a_pka = mol.getAcidPkaValue(atom, 5, 2)
b_pka = mol.getBasicPkaValue(atom, 5, 2)
```

CIP Stereo Descriptors

Descriptors calculation is activated by corresponding Indigo option `molfile-saving-add-stereo-desc` and descriptors are added into generated mol file as data S-groups with special name field `INDIGO_CIP_DESC`. Setting Indigo option `molfile-saving-add-stereo-desc` to 0 (or false) (the default value) disables descriptors calculation and removes all such data S-groups during corresponding mol file generation.

Please see the CIP Descriptors for detailed examples.

IO

Reading SDF, RDF, CML, multiline SMILES files, CDX (binary)

The following methods of the `Indigo` class can be used to enumerate files with multiple molecules/reactions:

- `iterateSDFile`
- `iterateRDFile`
- `iterateSmilesFile`
- `iterateCMLFile`
- `iterateCDXFile`

Java:

```
for (IndigoObject item : indigo.iterateSDFile("structures.sdf"))
  System.out.println(item.grossFormula());
```

C#:

```
foreach (IndigoObject item in indigo.iterateSDFile("structures.sdf"))
  System.Console.WriteLine(item.grossFormula());
```

Python:

```
for item in indigo.iterateSDFile("structures.sdf"):
  print item.grossFormula()
```

Indexed access

You can access items from SDF/RDF/CML/SMILES files by index, using the `at` method. The numbering of the items is zero-based.

Java, C#:

```
IndigoObject reader = indigo.iterateSDFile("structures.sdf");
String s = reader.at(15).smiles()
```

Python: the same with the `IndigoObject` and `String` omitted.

Accessing Molecule and Reaction Properties

NAMES

Molecule or reaction name is associated with the first line of the Molfile/Rxnfile the molecule/reaction was loaded from. In case it was loaded from SMILES string, the word immediately following the SMILES code is taken.

`IndigoObject.name` and `IndigoObject.setName` are the getter and setter of the molecule/reaction name.

Java, C#:

```
String name = mol2.name();
mol2.setName("another name");
mol2.saveMolfile("mol2.mol");
```

Python: the same with the `IndigoObject` and `String` omitted.

SDF/RDF/CDX PROPERTIES

The following methods are used to get the named properties of the molecule/reaction loaded from an SDF or RDF or CDX file:

- `IndigoObject.hasProperty` (boolean) — checks if the object has the given property.
- `IndigoObject.getProperty` (string) — returns the value of the given property, or raises an error in case the object does not have it.
- `IndigoObject.removeProperty` — removes the given property from the object, or does nothing in case the object does

not have it.

- **Note:** currently CDX annotations are used as CDX properties

Java:

```
for (IndigoObject item : indigo.iterateSDFile("structures.sdf"))
  if (item.hasProperty("cdbregno"))
    System.out.println(item.getProperty("cdbregno"));
```

C#:

```
foreach (IndigoObject item in indigo.iterateSDFile("structures.sdf"))
  if (item.hasProperty("cdbregno"))
    System.Console.WriteLine(item.getProperty("cdbregno"));
```

Python:

```
for item in indigo.iterateSDFile("structures.sdf"):
  if item.hasProperty("cdbregno"):
    print item.getProperty("cdbregno")
```

It is also possible to iterate over all properties of a particular record in SDF or RDF or CDX file.

Java:

```
for (IndigoObject item : indigo.iterateRDfile("reactions.rdf"))
  for (IndigoObject prop : item.iterateProperties())
    System.out.println(prop.name() + " : " + prop.rawData())
```

C#:

```
foreach (IndigoObject item in indigo.iterateRDfile("reactions.rdf"))
  foreach (IndigoObject prop in item.iterateProperties())
    System.Console.WriteLine(prop.name() + " : " + prop.rawData())
```

Python:

```
for item in indigo.iterateRDFile("reactions.rdf"):
  for prop in item.iterateProperties():
    print prop.name(), ":", prop.rawData()
```

Writing SDF Files

It is possible to write molecules to SDF files, along with arbitrary properties assigned.

Java and C#:

```
IndigoObject saver = indigo.writeFile("structures.sdf");

IndigoObject mol = indigo.loadMolecule("C1CCC1");
mol.setName("cyclobutane");
mol.setProperty("id", "8506");
saver.sdfAppend(mol);

mol = indigo.loadMolecule("C(NNN)C");
mol.setProperty("id", "42");
saver.sdfAppend(mol);
```

Python: the same with the `IndigoObject` omitted.

Writing RDF Files

Molecules and reactions can be written to an RDF file with the `rdfAppend` method. However, the special `rdfHeader` method must be called exactly once before anything is written into the RDF file.

Java and C#:

```
IndigoObject f = indigo.writeFile("test.rdf");
f.rdfHeader();
mol.setProperty("WHAT", "a molecule");
rxn.setProperty("WHAT", "a reaction");
f.rdfAppend(mol);
f.rdfAppend(rxn);
```

Python: the same with the `IndigoObject` omitted.

Writing CML Files

Molecules can be written to a CML file with the `cmlAppend` method. However, the `cmlHeader` method must be called exactly once before anything is written into the CML file, and the `cmlFooter` method must be called after all the molecules has been written.

Java and C#:

```
IndigoObject f = indigo.writeFile("structures.cml");
f.cmlHeader();
f.cmlAppend(mol1);
f.cmlAppend(mol2);
f.cmlFooter();
```

Python: the same with the `IndigoObject` omitted.

Writing Multiline SMILES Files

Java and C#:

```
IndigoObject f = indigo.writeFile("test.smi");
f.smilesAppend(mol);
f.smilesAppend(rxn);
```

Python: the same with the `IndigoObject` omitted.

Closing Files

The file output stream is closed on the object's destructor. However, you may need to close it explicitly. You can use the `close` method:

```
f.close();
```

You may need to be sure that the file will be closed after some block of code has finished executing. In Java, you can call the `close` method in `finally` block:

```
IndigoObject f = null;
try
{
  f = indigo.writeFile("test.sdf");
  f.sdfAppend(mol);
}
finally
{
  if (f != null)
    f.close();
}
```

In C#, the “using” statement can make the job easier:

```
using (IndigoObject f = indigo.writeFile("test.sdf"))
{
  f.sdfAppend(mol);
}
```

In Python, the “with” syntax serves the same purpose:

```
with indigo.writeFile("test.sdf") as f:
  f.sdfAppend(mol)
```

Generic Interface to Writing Files

There is also a generic interface to writing multiple molecules/reactions in a file. You can create a generic file saver (`IndigoObject.createFileSaver`), specifying the desired format to it, and then append items to it (`IndigoObject.append`), not bothering about header and footer — the saver will write them for you automatically.

The `IndigoObject.createFileSaver` method accepts the output file name and a string with the desired format, which is “sdf”, “rdf”, “cml”, or “smi”.

Java:

```
IndigoObject saver = null;
try
{
  saver = indigo.createFileSaver("reactions.cml", "cml");
  saver.append(rxn1);
  saver.append(rxn2);
}
finally
{
  if (saver != null)
    saver.close();
}
```

C#:

```
using (IndigoObject saver = indigo.createFileSaver("reactions.cml", "cml"))
{
   saver.append(rxn1);
   saver.append(rxn2);
}
```

Python:

```
with indigo.createFileSaver("reactions.cml", "cml") as saver:
   saver.append(rxn1);
   saver.append(rxn2);
```

Writing into a Memory Buffer

You can write the SDF or RDF data into a memory buffer instead of a file. `Indigo.writeBuffer` method creates and returns a memory output stream. After you have finished writing, you can call the `toString` method to obtain the written data.

Java, C#:

```
IndigoObject buf = indigo.writeBuffer();
mol.setProperty("name", "cyclobutane");
buf.sdfAppend(mol);
String s = buf.toString();
```

Python: the same with the `IndigoObject` and `String` omitted.

You can also create a generic saver on top of an existing buffer writer

Java, C#:

```
IndigoObject saver = indigo.createSaver(buf, "cml");
```

Python: the same with the `IndigoObject` omitted.

Serialization

You can use the `IndigoObject.serialize` method to serialize a molecule or a reaction into a binary byte array. All molecule and reaction features are serialized, including atom coordinates, stereochemistry, bond orientations, highlighting, AAM numbers, etc. To restore the molecule/reaction back, use the `Indigo.unserialize` method.

- **Note:** groups are not serialized (i.e. polymer brackets and data s-groups will be lost during serialization)
- **Note:** Molecule/reaction name and SDF properties ate not serialized

Java, C#:

```
byte[] data = mol.serialize();
...
IndigoObject mol2 = indigo.unserialize(data);
```

Python: the same with the `IndigoObject` omitted.

Match and Similarity

Fingerprints

The `IndigoObject.fingerprint` method works for molecules and reactions (including query molecules and query reactions) and returns a `fingerprint` object. `IndigoObject.toString` and `IndigoObject.toBuffer` methods of that object can be called to obtain a hex-string representation or a byte array of a fingerprint, respectively.

The `fingerprint` method accepts a string with a requested fingerprint type. The following fingerprint types are available:

- `sim` — “Similarity fingerprint”, useful for calculating similarity measures (the default)
- `sub` — “Substructure fingerprint”, useful for substructure screening
- `sub-res` — “Resonance substructure fingerprint”, useful for resonance substructure screening
- `sub-tau` — “Tautomer substructure fingerprint”, useful for tautomer substructure screening
- `full` — “Full fingerprint”, which has all the mentioned fingerprint types included

The size of the fingerprint can be controlled via a number of fingerprinting options.

Java:

```
fp1 = mol1.fingerprint("sim");
fp2 = mol2.fingerprint("sim");
String fp1string = fp1.toString();
byte[] fp2bytes = fp1.toBuffer();
```

Python: the same with `IndigoObject` and `String` omitted.

The `IndigoObject.countBits` method calculates the number of nonzero bits in a fingerprint. The `Indigo.commonBits` method calculates the number of coincident nonzero bits in two fingerprints.

Java and C#:

```
int bits1 = fp1.countBits();
int bits2 = fp2.countBits();
int bits12 = indigo.commonBits(fp1, fp2);
```

Python: the same with the `int` omitted.

Molecule and Reaction Similarity

The `Indigo.similarity` method accepts two molecules, two reactions, or two fingerprints. It also accepts a "metrics" string and returns a float value — the similarity measure between the two given items. There are three available metrics, which are calculated in the following way:

- `tanimoto` (the default) : c / (a + b - c)
- `tversky <alpha> <beta>` : c / ((a - c) * alpha + (b - c) * beta) (if alpha and beta are omitted, they are taken for alpha = beta = 0.5)
- `euclid-sub` : c / a

Where:

- "a" is the number of nonzero bits in the first fingerprint
- "b" is the number of nonzero bits in the second fingerprint
- "c" is the number of coincident bits in the two fingerprints

Java and C#:

```
float sim1 = indigo.similarity(fp1, fp2, "tanimoto");
float sim2 = indigo.similarity(rxn1, rxn2, "tversky");
float sim2 = indigo.similarity(mol1, mol2, "tversky 0.1 0.9");
float sim4 = indigo.similarity(mol1, mol2, "euclid-sub");
```

Python: the same with the `float` omitted.

Exact Match

The `Indigo.exactMatch` method accepts two molecules or reactions and returns a "mapping" object — the result of the exact match of two given molecules or reactions. If match is not possible, it returns `null`. If you are interested only in does the match exist or not, you can just check if the result of the method is `null`. Otherwise, please check below how to work with the mapping objects.

Note: You can not pass query molecules or reactions to `exactMatch`.

EXACT MATCHING FLAGS FOR MOLECULES

The `Indigo.exactMatch` method accepts an optional string. In this string, you can pass the following flags to the matching procedure:

Flag	Comment
`ELE`	Distribution of electrons: bond types, atom charges, radicals, valences
`MAS`	Atom isotopes
`STE`	Stereochemistry: chiral centers, stereogroups, and cis-trans bonds
`FRA`	Connected fragments: disallows match of separate ions in salts
`ALL`	All of the above (the most permissive matching)
`TAU`	Tautomer matching (not compatible to other flags)
`NONE`	No conditions (the most flexible kind of search)
`$number`	Maximum allowed RMS value for affine transformation match

The flags should go in the parameters string separated by space. Each flag specifies a constraint for the matching procedure. The more flags you set, the more restrictive the matching is. You can write the minus sign before the flag to exclude it from the 'ALL' flag. For example, 'ALL -MAS' means that all the described features except the isotopes must match.

The `TAU` flag stands for tautomer matching. It can not be combined with any other flags. You can see examples of exact matching, affine transformation matching, and exact tautomer matching in the Bingo User Manual.

Note: When no flags are specified, the behavior is equivalent to the `ALL` flag.

The `$number`, if present, should go after the flags; for example: `ALL 0.1`. The given `number` is the maximum allowed root-mean-square deviation between the atoms of the two molecules, measured in angstroms. Prior to the calculation of the RMS value, the optimal affine transformation (translation+rotation+scale) is applied to one of the molecules to make it as close as possible to the other molecule.

Note: Affine transformation matching is not possible when the molecule's atoms do not have coordinates (e.g. the molecule was loaded from SMILES).

Java and C#:

```
IndigoObject match = indigo.exactMatch(mol1, mol2, "ALL 0.1");

if (match == null) // affine matching has failed?
  match = indigo.exactMatch(mol1, mol2);

if (match == null) // exact match has failed, trying to ignore stereochemistry
  match = indigo.exactMatch(mol1, mol2, "ALL -STE");

if (match == null) // failed again; trying the most permissive matching
  match = indigo.exactMatch(mol1, mol2, "NONE");
```

Python:

```
match = indigo.exactMatch(mol1, mol2, "ALL 0.1")
if not match:
  match = indigo.exactMatch(mol1, mol2) # equivalent to ALL
if not match:
  match = indigo.exactMatch(mol1, mol2, "ALL -STE") # equivalent to "ELE MAS FRA"
if not match:
  match = indigo.exactMatch(mol1, mol2, "NONE")
```

EXACT MATCHING FLAGS FOR REACTIONS

When calling the `Indigo.exactMatch` method for reactions, you can pass it the "ELE", "MAS", and "STE" flags, which have the same meaning as in molecule exact matching. You can also pass two reaction-specific flags:

Flag	Comment
`AAM`	Atom-atom mapping
`RCT`	Reacting centers

Note: `FRA`, `TAU`, and affine matching are not available for reactions.

Molecule Substructure Matching

The `Indigo.substructureMatcher` method accepts a molecule (but not a query molecule) and returns a "substructure matcher" object. The given molecule is going to be the "target" molecule for the substructure match.

MOLECULE SUBSTRUCTURE MATCHING FLAGS

The `Indigo.substructureMatcher` method accepts an optional string, in which you can specify one of the following flags:

- `RES` — chemical resonance matching mode (`C=N` matches `CN=O` etc). You can read more about the resonance search and see examples on the Bingo User Manual page.
- `TAU` — tautomer matching mode. You can read more about the tautomer substructure search and see examples on the Bingo User Manual page.
- `TAU INCHI` — tautomer matching mode based on InChI code. Read more about the tautomer substructure search and see examples on the Bingo User Manual page.
- `TAU RSMARTS` — tautomer matching mode based on RSMARTS rules. Read more about the tautomer substructure search and see examples on the Bingo User Manual page.

METHODS OF SUBSTRUCTURE MATCHER

- `IndigoObject.ignoreAtom` method accepts an atom of the target molecule and marks it as "ignored" for the substructure matcher. No atom of any query will be mapped to this atom.
- `IndigoObject.unignoreAtom` method cancels the effect of the `ignoreAtom` method for the given atom.
- `IndigoObject.unignoreAllAtoms` method cancels the effect of previous `ignoreAtom` calls.
- `IndigoObject.match` accepts a query molecule and returns a "mapping" object if the matching has succeeded. If

multiple matches are possible, the first one is taken. If no match is possible, it returns `null`.

- `IndigoObject.countMatches` accepts a query molecule and returns the number or unique matches.
- `IndigoObject.countMatchesWithLimit` works like `countMatches`, but also accepts an additional integer parameter for limiting the number of matches.
- `IndigoObject.iterateMatches` accepts a query molecule and returns an iterator for unique matches ("mapping" objects).

Also, please take a look on some options available for substructure matchers.

Reaction Substructure Matching

You can pass a reaction to the `Indigo.substructureMatcher` method to obtain a "reaction substructure matcher" object. It responds to the `IndigoObject.match` method similarly to the molecule matcher. This method returns a "reaction match" object, or `null` if the matching is not possible.

You can specify optional `DAYLIGHT-AAM` flag to the matcher. When this flag is present, the matching of the atom-atom mapping (reaction AAM) is following the Daylight's rules for reaction SMARTS. Otherwise, it follows ordinary rules that are normal for matching Rxnfiles.

Java and C#:

```
IndigoObject rxn1 = indigo.loadReactionSmarts("[C:1][C:1]>>[C:1]");
IndigoObject rxn2 = indigo.loadReaction("[CH3:7][CH3:8]>>[CH3:7][CH3:8]");
IndigoObject match = indigo.substructureMatcher(rxn2, "DAYLIGHT-AAM").match(rxn1);
```

Python: the same with the `IndigoObject` omitted.

Note: `ignoreAtom`, `unignoreAtoms`, `unignoreAllAtoms`, `countMatches`, `countMatchesWithLimit`, and `iterateMatches` methods are not supported by reaction substructure matcher.

Converting the Substructure Query to the Aromatic Form

If your query is loaded from a Molfile or SMILES and its aromatic rings are represented in Kekule form, you should transform it to aromatic form prior to matching. Otherwise, it will not match the aromatic rings in the target structure.

Java and C#:

```
IndigoObject query = indigo.loadMoleculeFromFile("kekule.mol");
query.aromatize();
```

Python: the same with the `IndigoObject` omitted.

Note: `aromatize` does not have any effect for SMARTS queries (and should not be called on them).

Optimizing the Substructure Query

If you have a complex molecule or reaction SMARTS query, you can "optimize" it in-memory so that the matching performs faster.

Java and C#:

```
IndigoObject query = indigo.loadSmarts("[$([a;r4,!R1&r3])]1:[$([a;r4,!R1&r3])]:[$([a;r4,!R1&r3])]:[$([a;r4,!R1&r3])]:1");
query.optimize();
```

Python: the same with the `IndigoObject` omitted.

Methods of "mapping" object

You can get the detailed information about the succeeded matching via the returned "mapping" `IndigoObject`. It responds to the following method calls:

- `IndigoObject.mapAtom` method accepts an atom of the query molecule/reaction and returns the corresponding atom from the target molecule/reaction. In case the query atom is unmapped (this can happen only with explicit hydrogens), it returns `null`. You can use `IndigoObject.index` method to get mapped atom index.

- `IndigoObject.mapBond` method is similar to `mapAtom`, but accepts a bond of the query and returns a bond of the target. It can also return `null` if the bond is unmapped (this can happen only with bonds to explicit hydrogens).
- `IndigoObject.mapMolecule` method is similar to `mapAtom`, but accepts a molecule of the query reaction and returns a molecule of the target reaction. It can also return `null` if the molecule is unmapped (this can happen if a molecule within a query reaction represents explicit hydrogen).
- `IndigoObject.highlightedTarget` method returns a copy of the target molecule/reaction where the matched atoms and bonds are highlighted.

Note: The `IndigoObject.mapAtom` and `IndigoObject.ignoreAtom` methods accepts only atoms, and not atom indices. Similarly, the `IndigoObject.mapBond` and `IndigoObject.mapMolecule` method accepts only bonds and molecules respectively, and not their indices.

EXAMPLE 1 (MAPPING QUERY MOLECULE ATOMS)

Java:

```
IndigoObject match = indigo.substructureMatcher(mol).match(query);

if (match != null)
{
   match.highlightedTarget().saveMolfile("highlighted.mol");
   for (IndigoObject atom : qmol2.iterateAtoms())
      System.out.printf("atom %d mapped to atom %d\n", atom.index(), match.mapAtom(atom).index());
}
```

C#:

```
IndigoObject match = indigo.substructureMatcher(mol).match(query);

if (match != null)
{
   match.highlightedTarget().saveMolfile("highlighted.mol");
   foreach (IndigoObject atom in qmol2.iterateAtoms())
      System.Console.WriteLine("atom {0} mapped to atom {1}", atom.index(), match.mapAtom(atom).index());
}
```

Python:

```
match = indigo.substructureMatcher(mol).match(query)

if match:
   match.highlightedTarget().saveMolfile("highlighted.mol")
   for atom in qmol2.iterateAtoms():
      print "atom", atom.index(), "mapped to atom", match.mapAtom(atom).index()
```

EXAMPLE 2 (GETTING THE HIGHLIGHTED TARGET MOLECULE)

Java:

```
IndigoObject matcher = indigo.substructureMatcher(mol);
IndigoObject query = indigo.loadSmarts("O=[!C;R]");

System.out.println(matcher.countMatches(query);
for (IndigoObject match : matcher.iterateMatches(query))
   System.out.println(match.highlightedTarget().smiles());
```

C#:

```
IndigoObject matcher = indigo.substructureMatcher(mol);
IndigoObject query = indigo.loadSmarts("O=[!C;R]");

System.Console.WriteLine(matcher.countMatches(query);
foreach (IndigoObject match in matcher.iterateMatches(query))
  System.Console.WriteLine(match.highlightedTarget().smiles());
```

Python:

```
matcher = indigo.substructureMatcher(mol)
query = indigo.loadSmarts("O=[!C;R]")

print matcher.countMatches(query)
for match in matcher.iterateMatches(query):
  print match.highlightedTarget().smiles()
```

EXAMPLE 3 (EXACT MATCH BETWEEN REACTIONS)

Java:

```
IndigoObject match = indigo.exactMatch(rxn1, rxn2, "ALL -AAM");

if (match != null)
{
  for (IndigoObject mol : rxn1.iterateMolecules())
  {
    System.out.println("molecule #" + mol.index());

    for (IndigoObject atom : mol.iterateAtoms())
      if (match.mapAtom(atom) != null)
        System.out.printf("atom #%d matched\n", atom.index());

    for (IndigoObject bond : mol.iterateBonds())
      if (match.mapBond(atom) != null)
        System.out.printf("bond #%d matched\n", bond.index());
  }
}
```

C#:

```
IndigoObject match = indigo.exactMatch(rxn1, rxn2, "ALL -AAM");

if (match != null)
{
  foreach (IndigoObject mol in rxn1.iterateMolecules())
  {
    System.Console.writeLine("molecule #" + mol.index());

    foreach (IndigoObject atom in mol.iterateAtoms())
      if (match.mapAtom(atom) != null)
        System.Console.WriteLine("atom #{0} matched\n", atom.index());

    foreach (IndigoObject bond in mol.iterateBonds())
      if (match.mapBond(bond) != null)
        System.Console.WriteLine("bond #{0} matched\n", bond.index());

  }
}
```

Python:

```
match = indigo.exactMatch(rxn1, rxn2, "ALL -AAM")

if match:
  foreach mol in rxn1.iterateMolecules():
    print "molecule #" + mol.index();
    for atom in mol.iterateAtoms():
      if match.mapAtom(atom):
        print "atom #", atom.index(), "matched"

    for bond in mol.iterateBonds():
      if match.mapBond(bond):
        print "bond #", bond.index(), "matched"
```

Tautomer Matching Rules

- By default, any chains that satisfy basic conditions of tautomerism, are considered tautomeric by he tautomer matcher. You can restrict the tautomer matching by enabling conditions for boundary atoms in tautomeric chains. Each rule consists of two conditions, for two boundary atoms of the chain. The rules are numbered (1,2,…), and the numbers can be used in matcher's parameter string after the `TAU` specifier (`TAU R1 R2`). The more rules you specify, the more flexibility you receive in the search; *but* when you specify no rules at all (`TAU`), you get the most flexible search because no rules are checked. Any tautomeric chain is acceptable in this case. You can also use `TAU R*` to specify all rules at once.
- Some metal bonds and atom charges can replace hydrogen in tautomeric chains. You can add the HYD word to disable such hydrogen replacements (`TAU HYD`).
- Ring-chain tautomerism is disabled by default. You can add `R-C` to enable it (`TAU R-C`).

The following three rules are recommended and are used in examples:

- Each boundary atom in the tautomeric chain must be one of N, O, P, S, As, Se, Sb, Te
- Carbon not from the aromatic ring at one end of the tautomeric chain, and one of N, O, P, S at the other end
- Carbon from the aromatic ring at one end of the tautomeric chain and one of N, O at the other end

CUSTOMIZING THE RULES

`Indigo` provides thee methods to customize the rules: `clearTautomerRules`, `setTautomerRule`, and `removeTautomerRule`. There are no rules by default. The `Indigo.setTautomerRule` method accepts an ID number of the rule (must be from 1 to 32), and two strings (for two bound atoms of the chain), containing allowed elements are separated by commas. '1' at the beginning means an aromatic atom, and '0' means an aliphatic (non-aromatic) atom.

The three rules defined above can be set in the following way:

Java and C#:

```
indigo.setTautomerRule(1, "N,O,P,S,As,Se,Sb,Te", "N,O,P,S,As,Se,Sb,Te");
indigo.setTautomerRule(2, "0C", "N,O,P,S");
indigo.setTautomerRule(3, "1C", "N,O");

IndigoObject mol1 = indigo.loadQueryMolecule("OC1=CC=CNC1");
IndigoObject mol2 = indigo.loadMolecule("O=C1CNCC2=CC=CC=C12");

IndigoObject matcher = indigo.substructureMatcher(mol2, "TAU R2");
IndigoObject match = matcher.match(mol1);
```

Python: the same with `IndigoObject` omitted

Highlighting Atoms and Bonds

The `IndigoObject.highlight` and `IndigoObject.unhighlight` methods can be applied to an atom or a bond. Also, the `IndigoObject.unhighlight` method can be applied to a molecule or reaction to dismiss the highlighting on the whole molecule or reaction. You can check if object is highlighted using `IndigoObject.isHighlighted` method.

The following example shows how to highlight the matched atoms and bonds on a substructure matcher's target, along

with explicit hydrogens attached to the matched atoms.

Java:

```
IndigoObject matcher = indigo.substructureMatcher(target);
IndigoObject match = matcher.match(query);

for (IndigoObject item : query.iterateAtoms())
{
   IndigoObject atom = match.mapAtom(item);
   atom.highlight();

   for (IndigoObject nei : atom.iterateNeighbors())
   {
      if (!nei.isPseudoatom() && !nei.isRSite() && nei.atomicNumber() == 1)
      {
         nei.highlight();
         nei.bond().highlight();
      }
   }
}

for (IndigoObject bond : query.iterateBonds())
   match.mapBond(bond).highlight();

System.out.println(target.smiles());
target.unhighlight();
System.out.println(target.smiles());
```

C#:

```
IndigoObject matcher = indigo.substructureMatcher(target);
IndigoObject match = matcher.match(query);

foreach (IndigoObject qatom in query.iterateAtoms())
{
   IndigoObject atom = match.mapAtom(qatom);
   atom.highlight();

   foreach (IndigoObject nei in atom.iterateNeighbors())
   {
      if (!nei.isPseudoatom() && !nei.isRSite() && nei.atomicNumber() == 1)
      {
         nei.highlight();
         nei.bond().highlight();
      }
   }
}

foreach (IndigoObject bond in query.iterateBonds())
   match.mapBond(bond).highlight();

System.Console.WriteLine(target.smiles());
target.unhighlight();
System.Console.WriteLine(target.smiles());
```

Python:

```
matcher = indigo.substructureMatcher(target)
match = matcher.match(query)

for qatom in query.iterateAtoms():
  atom = match.mapAtom(qatom)
  atom.highlight()

  for nei in atom.iterateNeighbors():
    if not nei.isPseudoatom() and not nei.isRSite() and nei.atomicNumber() == 1:
      nei.highlight()
      nei.bond().highlight()

for bond in query.iterateBonds():
  match.mapBond(bond).highlight()

print target.smiles()
target.unhighlight()
print target.smiles()
```

Rendering

Rendering in Indigo is done by the `IndigoRenderer` plugin which is included in the Indigo distribution. The plugin is available for C (as a separate dynamic library), and has wrappers for all supported languages.

To work with the rendering plugin in C#/Java/Python wrappers, one needs to create an instance of `IndigoRenderer`, passing an existing `Indigo` instance to it.

Java and C#:

```
IndigoRenderer renderer = new IndigoRenderer(indigo);
```

A large number of options can be set for the rendering plugin. They are set via the ordinary `Indigo` instance. One of the options is obligatory: `render-output-format` tells which format the image should be rendered to.

The `IndigoRenderer.renderToBuffer` accepts an `IndigoObject` and returns a byte buffer with PNG, SVG, or PDF image (on the Windows platform, EMF format is supported too). Similarly, the `IndigoRenderer.renderToFile` method saves the image to a file.

Java and C#:

```
indigo.setOption("render-output-format", "png");
indigo.setOption("render-margins", 10, 10);
mol1.layout();
indigo.setOption("render-comment", "N-Hydroxyaniline")
renderer.renderToFile(mol1, "mol.png");

indigo.setOption("render-output-format", "svg");
byte[] svg = renderer.renderToBuffer(rxn);
```

Python: the same with the `byte[]` omitted.

Rendering Multiple Items to Grid

`IndigoRenderer.renderGridToFile` and `IndigoRenderer.renderGridToBuffer` methods can be used to render multiple molecules or reactions at once. The rendered items are placed in grid specified by its number of columns. Some options are available for placing titles under (or above) the items.

Java:

```
IndigoObject arr = indigo.createArray();

for (IndigoObject item : indigo.iterateSDFile("structures.sdf"))
  arr.arrayAdd(item);

renderer.renderGridToFile(collection, null, 4, "structures.png");
```

C#:

```
IndigoObject arr = indigo.createArray();

foreach (IndigoObject item in indigo.iterateSDFile("structures.sdf"))
  arr.arrayAdd(item);

renderer.renderGridToFile(collection, null, 4, "structures.png");
```

Python:

```
arr = indigo.createArray();

for item in indigo.iterateSDFile("structures.sdf"):
  arr.arrayAdd(item);

renderer.renderGridToFile(collection, None, 4, "structures.png")
```

The second parameter of the `IndigoRenderer.renderGridToFile` and `IndigoRenderer.renderGridToBuffer` methods is optional. You can specify it only for molecules, not for reactions. If this parameter is not `null`, it has to be an integer array whose number of elements is equal to the number of given molecules. Each element of this array is the index of the "reference atom" in the corresponding molecule. The rendering is then done in such a way that the reference atoms are grid-aligned. You can see the examples of this at CTR.

Win32 and .NET Support

In the .NET API, additional `IndigoRenderer` methods are available for working with Win32 HDC and `System.Drawing` objects directly:

- `void renderToHDC (IndigoObject item, IntPtr hdc, bool printing)`
- `Bitmap renderToBitmap (IndigoObject item)`
- `Metafile renderToMetafile (IndigoObject item)`

Scaffold Detection and R-Group Decomposition

Scaffold Detection

The `Indigo.extractCommonScaffold(mode [max_iterations])` method searches MCS scaffold for given molecules array and returns an extraction result. The method accepts a string parameter called `mode`. The following modes are available:

- `exact` : searches MCS using the exact algorithm.
- `approx` : searches MCS using the approximate algorithm

There is an optional integer parameter (iteration limit) followed by the mode. The default value for the exact algorithm is 0 (infinite) and for the approximate algorithm is 1000.

Indigo searches MCS in terms of a largest induced molecule1 subgraph isomorphic to an induced molecule2 subgraph. Largest means that a result subgraph can not be extended by the adding a vertex (atom). E.g. there are three possible mcs for the molecules 'NCOCCCS' and 'NCCOCCS' which can not be extended: 'CCOC', 'NC' and 'CCS'. Indigo stores all the intermediate subgraphs, which can be a result MCS. Thus, after a searching is finished, there is an array (all the solutions) of submolecules available. In the end, the result molecules array is sorted by the following rule: maximize rings number; if two molecules contain equal ring number, then maximize bonds number. The result sorted molecules array (all the maximum common submolecules) can be iterated by the `allScaffolds()` method. The `IndigoObject.allScaffolds()` method returns an array object.

Java:

```
IndigoObject arr = indigo.createArray();

for (IndigoObject item : indigo.iterateSDFile("structures.sdf"))
  arr.arrayAdd(item);

IndigoObject scaf = indigo.extractCommonScaffold(arr, "exact");

if (scaf != null) {
  System.out.println("max scaffold: " + scaf.smiles());
  for(IndigoObject scaffold: scaf.allScaffolds().iterateArray())
    System.out.println("current scaffold: " + scaffold.smiles());
}
```

C#:

```
IndigoObject arr = indigo.createArray();

foreach (IndigoObject item in indigo.iterateSDFile("structures.sdf"))
  arr.arrayAdd(item);

IndigoObject scaf = indigo.extractCommonScaffold(arr, "approx 2000");

if (scaf != null) {
  System.Console.WriteLine("max scaffold: " + scaf.smiles());
  for(IndigoObject scaffold: scaf.allScaffolds().iterateArray())
    System.Console.WriteLine("current scaffold: " + scaffold.smiles());
}
```

Python:

```
arr = indigo.createArray();

for item in indigo.iterateSDFile("structures.sdf"):
  arr.arrayAdd(item);

scaf = indigo.extractCommonScaffold(arr, "exact 1000");

if scaf:
  print "max scaffold:" + scaf.smiles()

  for scaffold in scaf.allScaffolds().iterateArray():
    print "current scaffold:" + scaffold.smiles()
```

R-Group Decomposition

The R-Group decomposition algorithm separates given molecule into a scaffold (matches the given query molecule) and R-Groups. To start working with the algorithm you need to create a `Decomposer` object. The `Indigo.createDecomposer` method accepts a query molecule and returns the `Decomposer` object. During a decomposition the `Decomposer` object builds a common scaffold with all R-Groups gathered together(so named `full scaffold`). The scaffold matches every passed molecule and together with calculated R-Groups restores initial molecules. Once the `Decomposer` object is created, you can pass molecules into and perform the decomposition. The method `IndigoObject.decomposeMolecule` returns a `DecompositionItem` object. The method accepts molecule. User can get the calculated results by accessing the `DecompositionItem` object. The following methods are used to get the results:

The `IndigoObject.decomposedMoleculeWithRGroups` returns molecule separated into a scaffold with R-Groups (`DecompositionItem` object). The `IndigoObject.decomposedMoleculeHighlighte` returns molecule with a highlighted scaffold (`DecompositionItem` object). The `IndigoObject.decomposedMoleculeScaffold` returns the full scaffold (`Decomposer` object) or a submolecule which matches the given query (`DecompositionItem` object).

Java, C#:

```
IndigoObject deco = indigo.createDecomposer(scaf);

IndigoObject deco_item1 = decomposer.decomposeMolecule(mol1);
deco_item1.decomposedMoleculeWithRGroups().saveMolfile("molrgoups1.mol");
deco_item1.decomposedMoleculeHighlighted().saveMolfile("highlighted1.mol");
deco_item1.decomposedMoleculeScaffold().saveMolfile("scaffold1.mol");

IndigoObject deco_item2 = decomposer.decomposeMolecule(mol2);
deco_item2.decomposedMoleculeWithRGroups().saveMolfile("molrgoups2.mol");
deco_item2.decomposedMoleculeHighlighted().saveMolfile("highlighted2.mol");
deco_item2.decomposedMoleculeScaffold().saveMolfile("scaffold2.mol");

IndigoObject scaffold = deco.decomposedMoleculeScaffold();
scaffold.saveMolfile("full_scaffold.mol");
```

Python:

```
deco = indigo.createDecomposer(scaf);

deco_item1 = decomposer.decomposeMolecule(mol1);
deco_item1.decomposedMoleculeWithRGroups().saveMolfile("molrgoups1.mol");
deco_item1.decomposedMoleculeHighlighted().saveMolfile("highlighted1.mol");
deco_item1.decomposedMoleculeScaffold().saveMolfile("scaffold1.mol");

deco_item2 = decomposer.decomposeMolecule(mol2);
deco_item2.decomposedMoleculeWithRGroups().saveMolfile("molrgoups2.mol");
deco_item2.decomposedMoleculeHighlighted().saveMolfile("highlighted2.mol");
deco_item2.decomposedMoleculeScaffold().saveMolfile("scaffold2.mol");

scaffold = deco.decomposedMoleculeScaffold();
scaffold.saveMolfile("full_scaffold.mol");
```

Note: There are possible situations when a given query molecule does not match a next decomposition molecule. An exception is raised in that case.

The `DecompositionItem` objects supports iterating matches since a given query scaffold can have several embeddings. The `IndigoObject.iterateDecompositions` returns the `DecompositionItem` iterator. The full scaffold can be completed by an appropriate match. The `IndigoObject.addDecomposition` (`Decomposer` object)method accepts a `DecompositionItem` object and adds the current match to the full scaffold. The following example shows the usage with iterating all matches:

Python:

```
...

# iterate over all the structures
for smiles in structures:
  # handle current molecule and handle exceptions
  try:
    item = deco.decomposeMolecule(indigo.loadMolecule(smiles))
    # iterate over all the decompositions
    for q_match in item.iterateDecompositions()
      # get decomposed molecules (current match)
      rg_mol = q_match.decomposedMoleculeWithRGroups()
      # add current match
      deco.addDecomposition(q_match)

  except Exception,e:
    # error handlers
```

Note: There are two different types of input queries: - user defined molecule with RGroups - simple query molecule, which can be passed from the Scaffold Detection

In the first case the full scaffold equals to the user defined molecule itself and it can not be changed during the RGroup Decomposition. In the second case the full scaffold should be generated by the library, and it will be returned by the `decomposedMoleculeScaffold` method. You can load scaffold from a molfile. The SMILES format is not supported at the

moment.

The following example shows the usage user-defined scaffold:

Python:

```
# prepare query scaffold (e.g. '(R1)C1CCCC(R2)C1')
scaffold = indigo.loadQueryMoleculeFromFile("query_mol")

# init decomposition
deco = indigo.createDecomposer(scaffold)

# load molecule
mol = indigo.loadMolecule('NC1CCCC(O)C1')

# create deco item
item = deco.decomposeMolecule(indigo.loadMolecule(smiles))

# iterate over all the decompositions
for q_match in item.iterateDecompositions()
  # get decomposed molecule (current match)
  rg_mol = q_match.decomposedMoleculeWithRGroups()

  # print molfile
  print(rg_mol.molfile())
```

In the example above there will be two matches: (R1)C1CCCC(R2)C1, R1=OH, R2=NH2 (R1)C1CCCC(R2)C1, R1=NH2, R2=OH

Note: the following code is deprecated:

Java:

```
IndigoObject deco = indigo.decomposeMolecules(scaf, arr);
deco.decomposedMoleculeHighlighted().saveMolfile("highlighted.mol");
IndigoObject scaffold = deco.decomposedMoleculeScaffold();
for (IndigoObject item : deco.iterateDecomposedMolecules())
  System.out.println(item.decomposedMoleculeWithRGroups().molfile());
```

C#:

```
IndigoObject deco = indigo.decomposeMolecules(scaf, arr);
deco.decomposedMoleculeHighlighted().saveMolfile("highlighted.mol");
IndigoObject scaffold = deco.decomposedMoleculeScaffold();
foreach (IndigoObject item in deco.iterateDecomposedMolecules())
  System.Console.WriteLine(item.decomposedMoleculeWithRGroups().molfile());
```

Python:

```
deco = indigo.decomposeMolecules(scaf, arr);
deco.decomposedMoleculeHighlighted().saveMolfile("highlighted.mol");
scaffold = deco.decomposedMoleculeScaffold();
for item in deco.iterateDecomposedMolecules():
  print item.decomposedMoleculeWithRGroups().molfile()
```

Reaction Products Enumeration

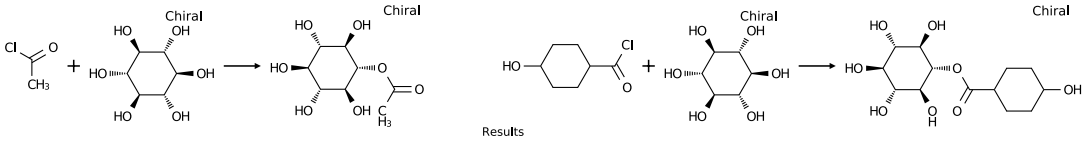
General API for the combinatorial chemistry capabilities of Indigo. More examples can be found here. Also some examples available at options page.

Java:

```
IndigoObject reaction = indigo.loadQueryReaction("Cl[C:1]([*:3])=O.[OH:2][*:4]>>[*:4][O:2][C:1]([*:3])=O");
IndigoObject monomers_table = indigo.createArray();

monomers_table.arrayAdd(indigo.createArray());
monomers_table.at(0).arrayAdd(indigo.loadMolecule("CC(Cl)=O"));
monomers_table.at(0).arrayAdd(indigo.loadMolecule("OC1CCC(CC1)C(Cl)=O"));

monomers_table.arrayAdd(indigo.createArray());
monomers_table.at(1).arrayAdd(indigo.loadMolecule("O[C@H]1[C@H](O)[C@@H](O)[C@H](O)[C@@H](O)[C@@H]1O"));

IndigoObject output_reactions = indigo.reactionProductEnumerate(reaction, monomers_table);
```

Python:

```
reaction = indigo.loadQueryReaction("Cl[C:1]([*:3])=O.[OH:2][*:4]>>[*:4][O:2][C:1]([*:3])=O")

monomers_table = indigo.createArray()

monomers_table.arrayAdd(indigo.createArray())
monomers_table.at(0).arrayAdd(indigo.loadMolecule("CC(Cl)=O"))
monomers_table.at(0).arrayAdd(indigo.loadMolecule("OC1CCC(CC1)C(Cl)=O"))

monomers_table.arrayAdd(indigo.createArray())
monomers_table.at(1).arrayAdd(indigo.loadMolecule("O[C@H]1[C@H](O)[C@@H](O)[C@H](O)[C@@H](O)[C@@H]1O"))

output_reactions = indigo.reactionProductEnumerate(reaction, monomers_table)

indigo.setOption("render-comment", "Results")
rxn_array = indigo.createArray();
for elem in output_reactions.iterateArray():
    rxn = elem.clone();
    rxn_array.arrayAdd(rxn)

indigoRenderer.renderGridToFile(rxn_array, None, 2, 'result_rpe.png')
```



Reaction-based Molecule Transformations

Usage examples are available here.

Java:

```
IndigoObject reaction = indigo.loadQueryReaction("[*+:1][*-:2]>>[*:2]=[*:1]");
IndigoObject molecule = indigo.loadMolecule("[O-][C+]1CCCC1[N+]([O-])=O");
indigo.transform(reaction, molecule);
System.out.println(molecule.smiles());
```

Python:

```
reaction = indigo.loadQueryReaction("[*+:1][*-:2]>>[*:2]=[*:1]")
molecule = indigo.loadMolecule("[O-][C+]1CCCC1[N+]([O-])=O")
indigo.transform(reaction, molecule)
print(molecule.smiles())
```

Output:

```
O=N(C1CCCC1=O)=O
```